Funktionale Programmierung mit Clojure

Dominikus Herzberg*
Hochschule Heilbronn

8. April 2013

Einige Leser und Leserinnen haben mir geholfen, kleinere Mängel im Text zu beseitigen. Mein besonderer Dank geht an Marc Hesenius.

Inhaltsverzeichnis

Herzlich Willkommen

Clojure ist das neue Lisp. Mit Lisp verbindet sich die Faszination einer unglaublich flexiblen und anpassungsfähigen Programmiersprache mit grenzenlosen Möglichkeiten — mit ihr begann vor über 55 Jahren die Zukunft. Lisp war seiner Zeit weit voraus. Clojure setzt die Zukunft fort. Moderne Sprachkonzepte machen Clojure zu einer der spannendsten funktionalen Programmiersprachen dieser Zeit. Wer funktional programmieren kann, schreibt bessere Programme. Und wer mit Clojure programmiert, kann sich den besten Weg zum besseren Programm aussuchen!

Obwohl Clojure einfach zu verstehen ist, Anfängern und insbesondere Wechslern von einer traditionellen Sprache wie Java oder C# fällt der Umstieg oft sehr schwer. Es dauert eine Weile, ehe der Einstieg in eine gänzlich andere Denkkultur der Programmierung gefunden ist. In der funktionalen Programmierwelt gibt es keine `for`- oder `while`-Schleifen, es gibt keine Variablenzuweisungen und eine `print`-Ausgabe ist schlechter Stil. Symbole, Funktionen, Rekursion, unveränderliche Datenstrukturen, verzögerte Sequenzen, Makros und vieles mehr sind die neuen Strukturmittel, aus dem neue Programmwelten entstehen.

Dieses Buch will Sie mitnehmen auf eine Erkundungstour in die funktionale Programmierung mit Clojure. Schritt für Schritt nähern wir uns der Sprache und der funktionalen Denkweise. Fallstudien zeigen Ihnen interessante Möglichkeiten funktionaler Programmierung mit Clojure auf. Viele Übungen leiten Sie zum Ausprobieren an. Mit Clojure kann man interaktiv programmieren; über ein Konsolenfenster können Sie direkt und ohne komplizierte Entwicklungsumgebung in Kontakt mit Clojure treten. Das macht das Programmieren einfach und spontan und gibt Ihnen sofort Rückmeldung, ob Sie etwas verstanden haben. So macht das Programmieren Spaß!

Das Buch will kurzweilig und praktisch orientiert sein und möchte Ihnen einen Weg in die funktionale Programmierung und Denke eröffnen. Das Buch versteht sich als Starthilfe und verzichtet auf eine vollständige Abdeckung aller Features von Clojure. Wenn Sie „Blut geleckt“ haben, dann hat das Buch seinen Zweck erfüllt, und Sie werden die sehr gute Dokumentation auf clojure.org für sich entdecken oder auf das ein oder andere, weiterführende Buch zu Clojure zurückgreifen.

Wenn Sie mithelfen wollen, dieses Buch zu verbessern: Ihre Korrekturen und Anregungen sind willkommen! Sie können gerne auch am Text mitarbeiten, Kapitel umarbeiten oder hinzufügen. Aus diesem Grund steht dieses Buch unter der Creative-Commons-Lizenz CC BY-NC-SA 3.0. Die Quelltexte finden Sie auf Github.

Sie dürfen dieses Buch gerne verbreiten. Eine kommerzielle Nutzung ist allerdings nicht erlaubt. Wenn Ihnen das Buch gefällt, ich freue mich, von Ihnen zu hören.

Nun viel Spaß mit Clojure!

Herzlichst,

Ihr Dominikus Herzberg, `dominikus.herzberg@gmail.com`

1 Am Anfang war die REPL

Wenn Sie Clojure über die Konsole starten, dann interagieren Sie mit Clojure über die sogenannte REPL, die Read-Evaluate-Print-Loop, die Lese/Auswerte/Ausgabe-Schleife. Nach dem Start über die Kommandozeile meldet sich die REPL mit `user=>` und wartet mit einem blinkenden Cursor auf Eingaben:

```
java -jar clojure.jar
Clojure 1.4.0
user=>
```

Alles, was Sie an der Konsole eingeben und mit einem ENTER abschließen, analysiert als erstes der Reader. Der Reader entscheidet, ob Ihre Eingabe vollständig ist und einen gültigen Ausdruck darstellt. Ist das der Fall, wertet der Evaluator den vom Reader übergebenen Ausdruck aus. Den Vorgang nennt man auch Evaluation (*evaluation*). Das Ergebnis der Auswertung wird an den Printer, die Ausgabeeinheit, weitergereicht. Der Printer wandelt die an ihn übergebenen Daten in Zeichen um, die dann wieder auf der Konsole erscheinen. Anschließend können Sie wieder eine Eingabe machen und den Vorgang von Read, Evaluate und Print in die nächste Runde schicken.

Wenn Sie ein Programm als Datei von Clojure abarbeiten lassen, dann können Sie sich den Ablauf vereinfacht so vorstellen, als würde die Programmdatei Zeile für Zeile von der REPL abgearbeitet werden.

1.1 Der Reader

Der Reader – dahinter verbirgt sich die Funktion `read` – erhält seinen Input von dem an `*in*` gebundenen Zeichenstrom; im Normallfall ist das die sogenannte Standard-Eingabe (*standard input*, oft als *stdin* abgekürzt). Der Reader nimmt eine Analyse der eingehenden Textzeichen vor; es handelt sich dabei um eine syntaktische Analyse. Die Analyse stellt beispielsweise fest, dass die Zeichenfolge „`23.1`“ eine Zahl, die Zeichenfolge „`square`“ ein Symbol und „`(square 23.1)`“ eine Liste mit einem Symbol und einer Zahl ist. Um Zahlen von Symbolen zu unterscheiden und um eine Liste erkennen zu können, müssen sich die Zeichenfolgen in ihrer (Erscheinungs)Form eindeutig voneinander unterscheiden. Sie müssen unterschiedlich kodiert sein. Ein Symbol darf z.B. nicht mit einer runden Klammer „`(`“ beginnen, da damit der Anfang einer Liste kodiert ist.

1.1.1 Formen für Daten

Die Formen, die der Reader unterscheidet (*reader forms*), sind nichts anderes als Kodierungen für Ausdrücke, die Datentypen repräsentieren. Es gibt Formen für Symbole und Literale, die einfache Datentypen (*primitive datatypes*) kodieren, und für Listen, Vektoren, Maps und Sets, die zusammengesetzte Datentypen (*composite datatypes*) kodieren. Als Literale kodiert sind Zeichenketten, Zahlen, Einzelzeichen und Schlüsselwörter.

Die Formen lassen sich mit wenigen Ausnahmen bereits am ersten Zeichen voneinander unterscheiden. Beginnt eine Zeichenfolge mit einem der

folgenden Zeichen, dann markiert das Zeichen den Anfang der beschriebenen Form. Die Bedeutung und der Sinn und Zweck der Datenstrukturen wird an anderer Stelle beschrieben; hier geht es nur um die Erkennung durch den Reader.

Zeichenkette `"` – Ein doppeltes Anführungszeichen eröffnet und schließt eine Zeichenkette (*string*); viele Programmiersprachen kodieren Zeichenketten so. Ein Beispiel ist der String `"I love Clojure!"`.

Einzelzeichen `\` – Ein umgekehrter Schrägstrich (*backslash*) kodiert ein Einzelzeichen (*character*). Zum Beispiel steht `\a` für das Zeichen „a“ und `\newline` für das Zeilenumbruch-Zeichen.

Keyword `:` – Ein Doppelpunkt (*colon*) markiert ein Schlüsselwort (*keyword*). Ein Schlüsselwort macht in der Regel nur Sinn im Zusammenhang mit sogenannten Maps. Ein Beispiel für ein Keyword ist `:monday`.

Zahl `0-9` – Eine Ziffer leitet die Kodierung für Zahlen ein. Beispiele für Zahlen sind `7`, `42`, `13.5` und `1.35E1`. Einer Zahl kann ein Plus- oder Minus-Zeichen als Vorzeichen voranstehen, was die Form für Zahlen um die Anfangszeichen `+` und `-` erweitert. Somit sind auch `-7` und `+13.5` gültige Zahl-Formen. Clojure kennt übrigens auch Bruchzahlen.

Liste `(` – Eine öffnende runde Klammer markiert den Beginn einer Liste, eine schließende runde Klammer `)` ihr Ende. Die Elemente der Liste sind ihrerseits durch Reader-Formen kodiert, was z.B. verschachtelte Listen erlaubt. Die Listenelemente werden durch Leerzeichen oder Kommas voneinander getrennt. Üblich ist die Trennung durch Leerzeichen. Ein Listen-Beispiel ist `(1 2 3)`.

Vektor `[` – Eine öffnende eckige Klammer zeichnet den Beginn eines Vektors aus, eine schließende eckige Klammer `]` sein Ende. Wie bei den Listen trennen Leerzeichen oder Kommas die Elemente des Vektors, und es sind beliebige Verschachtelungen mit anderen zusammengesetzten Formen möglich. Ein Beispiel für einen Vektor ist `[1 (2 3) "hey"]`.

Map `{` – Geschweifte Klammern kodieren eine Map, was man mit „Abbildung“ übersetzen kann. Die Klammern umschließen eine beliebige Anzahl von Paaren beliebiger Reader-Formen. Die Formen sind entweder durch Leerzeichen oder Kommas voneinander zu trennen. Beispiel: `{"mo" 1, "tue" 2}`.

Menge `#{` – Ein Hash-Zeichen gefolgt von einer öffnenden geschweiften Klammer markiert den Beginn einer Menge (*set*), eine schließende geschweifte Klammer `}` das Ende. In einer Menge kann es beliebige weitere Formen geben, die durch Leerzeichen oder Kommas getrennt sind. Ein Beispiel ist `#{1 2 3}`.

Der Reader erkennt außerdem die Zeichenfolgen `true` und `false` als die Boolschen Werte für „wahr“ und „falsch“ sowie `nil` für „nichts“.

Sie werden bei der Interaktion mit der REPL feststellen, dass für den Reader weniger das ENTER wichtig ist als vielmehr gewartet wird auf eine vollständige, gültige Form. Ein ENTER bei der Eingabe eines Vektors wird wie ein Leerzeichen behandelt.

```
user=> [1
2
3]
[1 2 3]
```


1.1.2 Makrozeichen und Reader-Makros

Neben den gelisteten Zeichen gibt es noch weitere Zeichen, die ein besonderes Verhalten des Readers kodieren. Diese Zeichen heißen Makrozeichen (*macro characters*) und sind in einer unveränderlichen Tabelle (*read table*) vermerkt. Das mit ihnen assoziierte Sonderverhalten des Readers bezeichnet man als Reader-Makros (*reader macros*).

Die Makrozeichen sind `'` (*quote*), `\` (Einzelzeichen sind also über ein Reader-Makro realisiert), `;` (*comment*), `@` (*deref*), `^` (*metadata*), `#` (*dispatch*), `` ` `` (*syntax quote*), `~` (*unquote*) und `~@` (*unquote-splicing*). Lisp-Programmierer(innen) werden einige der Zeichen anhand der englischen Bezeichnungen zu interpretieren wissen. Wir werden jedoch an anderer Stelle auf die Bedeutung dieser Zeichen zurückkommen. Zum Einstieg genügt es zu wissen, dass diese Zeichen eine Sonderbehandlung durch den Reader auslösen.

1.1.3 Symbole

Es bleibt die Frage, wie die Reader-Form für Symbole (*symbols*) aussieht. Prinzipiell könnte jede Zeichenfolge ein Symbol kodieren, die nicht mit einem Startzeichen für eine Reader-Form oder mit einem Makrozeichen beginnt. Tatsächlich ist der Reader ein wenig restriktiver bei der Kodierung von Symbolen. Ein Symbol darf mit einem Zeichen aus dem Alphabet samt Umlauten von `a` bis `Z` beginnen, einschließlich `+`, `-`, `*`, `/`, `!`, `?`, `.` und `_`. In der Folge dürfen sich beliebig viele weitere derartige Zeichen anschließen, inklusive der Ziffern von `0` bis `9` – sofern damit keine Zahl kodiert wird. Beispiele für Symbole sind `x`, `set-name` (in Clojure ist Kleinschreibung und die Trennung durch `-` statt durch `_` üblich), `name?` und `U2`.

Es gibt ein paar Besonderheiten bei der Kodierung von Symbolen. Die Zeichen `.` und `/` haben eine weitergehende Bedeutung: kommen diese Zeichen vor, so spricht man von einem „qualifizierten“ Symbol, einem *qualified symbol*. Details zu der Bedeutung von `.` und `/` in der Symbol-Kodierung folgen in einem späteren Kapitel.

Ein Tipp für Java-Programmierer(innen): Die Reader-Formen für Strings und Zahlen sind identisch mit den syntaktischen Konventionen in Java. Soll in einer Zeichenkette selbst ein Anführungszeichen vorkommen, so ist dem Anführungszeichen ein Backslash voranzustellen, wie z.B. in `"I say \"Hey\"!"`. Möchte man den Backslash selbst in einem String verwenden, so sind zwei Backslashes zu setzen: `"\\"`; das Backslash-Zeichen innerhalb eines Strings wird aufgrund seiner Sonderfunktion oft als Ausstiegszeichen (*escape character*) bezeichnet. Zahlen können mit einer Basis von 2 bis 36 angegeben werden, z.B. `2r101` und `4r11` für 5. Und Ihnen sei verraten, dass `nil` mit `null` in Java identisch ist.

1.2 Der Evaluator

Der Reader verarbeitet syntaktische Formen und gibt die erkannten Ausdrücke (*expressions*) als Daten an den Evaluator weiter. Der Evaluator nimmt eine Auswertung (*evaluation*) der übergebenen Ausdrücke (Daten) vor.

1.2.1 Die Evaluationsregeln

Die Regeln für die Auswertung sind einfach. Die Funktion `eval` führt die Evaluation durch.

1. Evaluationsregel : Alle Ausdrücke evaluieren zu sich selbst, ausgenommen Symbole und Listen. Die Bestandteile der zusammengesetzten Ausdrücke (Vektoren, Maps und Mengen) werden gemäß den Evaluationsregeln evaluiert.

   Das heißt, dass Zeichenketten, Einzelzeichen, Zahlen, Keywords, `true`, `false` und `nil` zu sich selbst evaluieren. Vektoren, Mengen und Maps evaluieren zwar wieder zu Vektoren, Mengen bzw. Maps, allerdings sind dann die Teilausdrücke, sprich die Elemente dieser zusammengesetzten Datentypen evaluiert.

   Auch wenn Sie mit den verschiedenen Formen bzw. Ausdrücken noch nicht vertraut sind, probieren Sie es an der Konsole aus: Die Zahl `7` evaluiert zur `7` und der Vektor `[1 2 3]` zum Vektor mit den evaluierten Zahlen `1`, `2` und `3`.

   ```
   user=> 7
   7
   user=> [1 2 3]
   [1 2 3]
   ```

   Symbole haben eine tragende Bedeutung in Clojure. Ein Symbol wird evaluiert zu dem an ihn gebundenen Wert – auf nähere Details der Bindungsauflösung gehen wir an dieser Stelle nicht ein.

2. Evaluationsregel : Ein Symbol evaluiert zu dem an ihn gebundenen Wert. Ist das Symbol nicht gebunden, liefere eine Fehlermeldung zurück. Ist der Bindungswert ein Makro, so ist dies ebenfalls ein Fehler.

   Man kann also in Clojure Bindungen von Symbolen zu Werten aufbauen. Einige Symbole sind in Clojure bereits an Werte gebunden, wie z.B. `+`; andere, wie z.B. `x`, nicht. In dem einen Fall gibt Clojure eine Ausgabe für die mit dem Symbol `+` verbundene Funktion zurück, im anderen Fall vermeldet Clojure, dass für das Symbol keine Bindung vorliegt.

   ```
   user=> +
   #<core$_PLUS_ clojure.core$_PLUS_@1f4bcf7>
   user=> x
   java.lang.Exception: Unable to resolve symbol: [...]
   ```

   Man kann Bindungen mit einer speziellen Form, mit `(def ...)` erzeugen. Das ungebundene Symbol `x` kann z.B. an den Wert `7` gebunden werden. Im nächsten Schritt kann Clojure dann das Symbol evaluieren.

   ```
   user=> (def x 7)
   #'user/x
   user=> x
   7
   ```

   Wir werden uns später noch genauer damit befassen, wie Bindungen über Namensräume und Umgebungen organisiert sind.

Programme sind Listen

Listen sind eine Datenstruktur, die eine geordnete Ansammlung von Werten (Daten) repräsentieren, ähnlich wie es auch Vektoren tun. Allerdings

– und das ist allen Lisp-basierten Sprachen eigen – werden Programme in Clojure als Listen notiert. Damit kommt den Listen eine Doppelbedeutung zu: Listen können entweder ein zusammengesetztes Datum darstellen oder ein Programm; man nennt diese Eigenschaft auch Homoikonizität (Selbst-Abbildbarkeit). Das hat interessante Konsequenzen: Ein Programm kann als Liste über Listenoperationen manipuliert und anschließend zur Ausführung gebracht werden. Das ist die Grundlage für selbstveränderliche Programme, Programme, die Programme schreiben (Programmgeneratoren) und sogenannte Makros. Außerhalb der Lisp-basierten Sprachwelt sind diese Features nahezu unbekannt. Dort erreicht man ähnliches deutlich umständlicher nur, wenn man Zugriff auf den Abstrakten Syntaxbaum (AST, *Abstract Syntax Tree*) einer Sprache erhält. In Clojure sind insbesondere Programmm-Manipulationen via Makros an der Tagesordnung und eine Leichtigkeit.

Clojure interpretiert Listen grundsätzlich als Programme. Darum evaluiert eine Liste nicht zu einer Liste, sondern folgt einer gesonderten Evaluationsregel.

3. Evaluationsregel : Die leere Liste `()` evaluiert zu einer leeren Liste. Eine nicht leere Liste wird gemäß der folgenden Fallunterscheidung entweder als Spezialform, als Makro oder als Funktionsaufruf evaluiert:

   1. Ist der Ausdruck ganz links in der Liste ein Symbol und identifiziert dieses Symbol eine sogenannte Spezialform (*special form*), so wende die Evaluationsregeln der Spezialform an. Zu den Spezialformen gehören u.a. `def`, `if`, `do`, `let`, `quote` und `fn`.
   2. Ist der Ausdruck ganz links in der Liste ein Symbol und ist dieses Symbol an ein Makro gebunden (das ist eine Funktion, die als Makro gekennzeichnet ist), so rufe das Makro auf und übergebe die restlichen Listenausdrücke unevaluiert an das Makro.
   3. Evaluiert der Ausdruck ganz links in der Liste zu einer Funktion, so gehe wie folgt vor: (1) Evaluiere von links nach rechts alle weiteren Ausdrücke in der Liste, d.h. befolge für jeden Ausdruck die Evaluationsregeln. (2) Wende anschließend die Funktion auf die Ergebnisse aus (1) an. Man sagt auch: Rufe die Funktion mit den Ergebnissen aus (1) als Argumente auf.

   Trifft keiner dieser Fälle zu, so ist die Liste kein gültiger Programmausdruck.

Sie werden in Kürze sehen (siehe S. 13 in Kap. 1.2.5), dass Clojure noch einen vierten Fall zu unterscheiden scheint: Evaluiert der Ausdruck ganz links in einer Liste zu einem Keyword, zu einem Symbol, zu einem Vektor oder einer Map, dann gilt jeweils ein Sonderfall. Ein Blick hinter die Kulissen verrät jedoch, dass Clojure tatsächlich nur eine Spielvariante des Funktionsfalls ausführt.

1.2.2 Funktionen

Wenn Sie die Symbole für die arithmetischen Grundoperationen wie `+`, `*` etc. an der Konsole eingeben, bekommen Sie als Ergebnis der Auswertung die über das Symbol gebundene Funktion zurück, also die Funktion zur Durchführung der Addition, der Multiplikation usw. Die Ausgabe erklärt sich, wenn wir über den Printer reden.


```
user=> +
#<core$_PLUS_ clojure.core$_PLUS_@1d381d2>
user=> *
#<core$_STAR_ clojure.core$_STAR_@afa68a>
```

Die Anwendung einer Funktion bedarf des Kontexts einer Liste – dann ist es ein Programm, in dem die Additionsfunktion und die Argumente für die Funktion vorkommen.

```
user=> (+ 2 3)
5
```

Das erste Element in der Liste, das Symbol `+`, evaluiert zur Additionsfunktion; das zweite und dritte Element, die Zahlen `2` und `3`, evaluieren zu sich selbst. Die Anwendung der Additionsfunktion auf die Zahlen liefert das Ergebnis `5`. Anders ausgedrückt: Der Aufruf der Additionsfunktion mit den Argumenten `2` und `3` liefert `5`.

Die Auswertungsregeln erlauben die Verschachtelung von Ausdrücken:

```
user=> (+ (* 2 3) (- 4 2))
8
```

Bevor die durch `+` gebundene Additionsfunktion angewendet werden kann, werden zuvor und der Reihe nach die weiteren Listenausdrücke ausgewertet. Das Programm `(* 2 3)` liefert `6`, das Programm `(- 4 2)` liefert `2`. Die Addition führt also `(+ 6 2)` aus, was `8` ergibt.

1.2.3 Spezialformen

Es ist unmöglich, Clojure vollständig nach dem Evaluationsschema für Funktionen aufzubauen. Warum? Nehmen Sie den obigen Ausdruck

```
(def x 7)
```

Wenn `def` eine Funktion wäre, dann müsste vor der Anwendung der über `def` gebundenen Funktion die Evaluation der Argumente `x` und `7` erfolgen. Die Auswertung des Symbols `x` würde zu einem Fehler führen, da es ursprünglich nicht gebunden ist. Die Intention, über `def` eine Bindung an das Symbol `x` einzuführen, scheitert daran, dass `x` zuvor nach seinem Bindungswert befragt wird. Darum muss `def` eine Spezialform sein. Spezialformen realisieren ihre eigene Auswertungsstrategie. Die Spezialform `def` zum Beispiel erwartet als erstes Argument ein Symbol, ohne nach dem an das Symbol gebundenen Wert zu fragen, evaluiert das zweite Argument und bindet dieses Ergebnis an das Symbol.

Eine andere Spezialform namens `quote` hilft, die Evaluierung einer Liste zu unterdrücken und lässt die Liste eine Datenstruktur sein.

```
user=> (quote (+ 2 3))
(+ 2 3)
```

Auf diese Weise können Listen als Daten und nicht als Programme behandelt werden. Eine Kurzform für `quote` ist das Makrozeichen `'`, auch Quotierungszeichen (*quote character*) genannt. Das Reader-Makro löst das Quotierungszeichen in die `quote`-Form auf. Gleichwertig zu oben ist also die Schreibweise

```
user=> '(+ 2 3)
(+ 2 3)
```


Mit der Funktion `eval` kann eine Liste explizit wieder als Programm interpretiert werden.

```
user=> (eval '(+ 2 3))
5
```

Die Quotierung unterdrückt ebenso die Evaluierung anderer Formen. Insbesondere bei den zusammengesetzten Formen (Vektoren, Maps und Mengen) verhindert die Quotierung die Auswertung der inneren Ausdrücke. Beachten Sie bei dem Beispiel, dass wir eine bestehende Bindung von `x` zu `7` haben.

```
user=> [+ x 3]
[#<core$_PLUS_ clojure.core$_PLUS_@197bb7> 7 3]
user=> '[+ x 3]
[+ x 3]
```

Sie haben mit `def` und `quote` zwei Spezialformen kennengelernt. Insgesamt gibt es rund ein Dutzend solcher Spezialformen in Clojure. Vollständig aufgelistet sind dies `def`, `if`, `do`, `let`, `quote`, `var`, `fn`, `loop`, `recur`, `throw`, `try` und `monitor`. Für die Interoperabilität mit Java gibt es außerdem die Spezialformen `.` (den Punkt, engl. *dot*), `new` und `set!`; `set!` hat sowohl im Zusammenhang mit Java als auch mit Var-Referenzen eine Bedeutung.

1.2.4 Makros

Sie erahnen vielleicht, wie das Spiel mit den Listen in der Behandlung als Datum oder als Programm sehr gut genutzt werden kann zur Manipulation von Programmen. Lassen Sie uns als Anschauungsbeispiel ein Makro namens `twice` definieren; beachten Sie bei der Eingabe das Backquote-Zeichen `‘`. Das Makro verdoppelt einen Zahlenwert, wie an der Beispielanwendung zu sehen ist.

```
user=> (defmacro twice [expr] `(+ ~expr ~expr))
#'user/twice
user=> (twice (+ 2 3))
10
```

Da `twice` ein Makro ist, wird das Makro mit dem nicht ausgewerteten Teilausdruck `(+ 2 3)` aufgerufen. Das Makro liefert einen neuen Ausdruck zurück, der durch die Makrodefinition beschrieben ist – und wertet diesen erzeugten Ausdruck aus. Mit Hilfe von `macroexpand` kann die Auflösung des Makros betrachtet werden.

```
user=> (macroexpand '(twice (+ 2 3)))
(clojure.core/+ (+ 2 3) (+ 2 3))
```

Der Aufruf von `(twice (+ 2 3))` löst sich auf zu `(+ (+ 2 3) (+ 2 3))`, was dann evaluiert wird und `10` ergibt.

1.2.5 Sonderfälle: Keywords, Symbole, Vektoren, Maps

Es gibt bei der Interpretation einer Liste als Programm eine Besonderheit zu berücksichtigen, die auftritt, wenn der Ausdruck ganz links in der Liste zu einem Keyword, einem Symbol, einem Vektor oder einer Map evaluiert. In diesem Fall verhalten sich die evaluierten Ausdrücke ähnlich wie

Funktionen, die eine bestimmte Anzahl an Argumenten erwarten. Letztlich dient das dazu, die Abfragen von Vektoren und Maps komfortabel zu gestalten. Gehen wir die Sonderfälle kurz durch.

Keywords und Symbole

Hat ein Keyword oder Symbol als erstes Argument eine Map und als optionales zweites Argument einen Defaultwert, dann wird der mit dem Keyword bzw. Symbol assoziierte Wert der Map zurückgegeben. Gibt es die Assoziation nicht, ist der Defaultwert das Ergebnis falls er gegeben wurde; sonst ist der Wert `nil`.

```
user=> (:tue {:mon 1 :tue 2 :wed 3 :thu 4 :fri 5})
2
user=> (:sat {:mon 1 :tue 2 :wed 3 :thu 4 :fri 5} 0)
0
user=> ('mon '{mon 1 tue 2 wed 3 thu 4 fri 5} 0)
1
user=> (:mon '{mon 1 tue 2 wed 3 thu 4 fri 5})
nil
```

Vektoren

Steht ein Vektor an erster Stelle in einer Liste und folgt ein einziges weiteres Argument, dann wird dieser Wert als Index interpretiert. Es wird der entsprechende Wert im Vektor zurückgegeben. Dabei ist zu beachten, dass das erste Element ganz links in einem Vektor mit dem Index 0, das folgende mit dem Index 1 assoziiert ist usw.

```
user=> ([5 7 3] 0)
5
```

Maps

Ähnlich ist der Zugriff auf Werte einer Map. Das der Map folgende Argument wird als Zugriffsschlüssel interpretiert, ein optionales weiteres Argument als Defaultwert. Das Verhalten ist analog zu dem obigen Fall, wenn das Keyword oder das Symbol an erster Stelle in der Liste steht.

```
user=> ({:mon 1 :tue 2 :wed 3 :thu 4 :fri 5} :tue)
2
user=> ({:mon 1 :tue 2 :wed 3 :thu 4 :fri 5} :sat 0)
0
```

1.3 Der Printer

Das Ergebnis eines evaluierten Ausdrucks ist ein Wert: ein Datum, das intern in irgendeiner, in der Regel maschinennahen Art kodiert ist. Der Printer wandelt diese interne Kodierung um in eine Folge von Zeichen, die dem Zeichenstrom für die Ausgabe hinzugefügt werden. Hierzu verwendet der Printer den an das Symbol `*out*` gebundenen Ausgabestrom. Im Normalfall ist das die Standard-Ausgabe (*standard output*, oft als *stdout* abgekürzt). Die Standard-Ausgabe wird bei der Interaktion mit der REPL auf der Konsole zur Anzeige gebracht.

1.3.1 Ausgaben im Reader-Format

Der Printer – dahinter steckt die Funktion `pr` – wählt für alle Datenwerte eine Darstellung, die kompatibel mit dem Reader ist: der Reader kann die Darstellung wieder in den gleichen Datenwert wandeln. Wer die Formen für die Eingabe kennt, wird keine Mühe haben, die Ausgabeformen zu verstehen. Das gilt für alle Datenstrukturen mit der Ausnahme von Funktionen.

```
user=> 42
42
```

Die Funktion `pr` macht die Art ihrer Ausgabe abhängig vom Evaluationswert des Symbols `*print-readably*`: „Drucke so, dass es durch die Reader-Funktion `read` lesbar, *readably*, ist“. Das ist der Fall in der Grundeinstellung (`true`).

Einzig Funktionen werden nicht in der Form ausgegeben, mit der sie definiert wurden (zur Definition von Funktionen in einem späteren Kapitel mehr), sondern in einer Kurzform. Die Kurzform stellt eine Funktion als Datenobjekt dar. Nehmen wir als Beispiel die `+`-Funktion.

```
user=> +
#<core$_PLUS_ clojure.core$_PLUS_@1b0bdc8>
```

Die Ausgabe innerhalb der Marker `#<` und `>` besteht aus zwei Anteilen und gibt die Bindung eines Symbols an eine Funktion wieder: Links ist das Symbol `_PLUS_` im Namensraum `core` angegeben; Clojure bildet einige Zeichen, wie etwa `+`, bei der Selbstauskunft von Funktionen auf sprechende Namen ab. Rechts ist die als `_PLUS_` vermerkte Funktion mit ihrer hexadezimalen Speicheradresse angegeben; die Funktion ist dem Namensraum `clojure.core` zugeordnet.

1.3.2 Ausgaben an den Anwender

Neben den Funktionen `pr` und `prn` (`prn` entspricht der Funktionsfolge von `pr` und `newline`; `newline` fügt dem Ausgabestrom einen Zeilenumbruch (*newline*) hinzu) mit denen der Printer arbeitet, gibt es die Ausgabefunktionen `print` und `println` (ein `print` mit `newline` in Folge). Diese Funktionen sind zur Ausgabe für Menschen gedacht, nicht für Ausgaben, die der Reader prinzipiell wieder verarbeiten könnte. Wenn Sie also Ausgaben in ihren Programmen machen möchten, dann werden Sie typischerweise `println` oder auch `print` verwenden. Beide verwenden eine beliebige Anzahl von Argumenten und fügen eine Zeichendarstellung der übergebenen Objekte an den Ausgabestrom an.

```
user=> (println "The result is:\t" 42)
The result is:   42
nil
```

Für `print` bzw. `println` wird `*print-readably*` an `nil` gebunden. Zeichenketten werden dann ohne Anführungszeichen ausgegeben und Sonderzeichen werden umgesetzt (z.B. als Tabulator, daher der Abstand) und nicht über Ausstiegszeichen (z.B. `\t`) dargestellt. Im Vergleich dazu die Ausgabe mit `prn`:

```
user=> (prn "The result is:\t" 42)
"The result is:\t" 42
nil
```

Alle diese Funktionen, die letztlich auf `pr` und `newline` zurückgehen und über `*print-readably*` gesteuert werden, haben eines gemeinsam: ihr Rückgabewert ist `nil`! Die Ausgabe einer Zeichenkette wird als sogenannter Seiteneffekt bezeichnet. Seiteneffekte sind ein heikles Thema in der funktionalen Programmierung. Der Rückgabewert einer Funktion darf nur von dem Eingabewert abhängen; Ausgaben passen nicht in das Schema. Wir werden uns damit noch ausführlich beschäftigen müssen.

1.3.3 String-Rückgaben statt Ausgaben

Es gibt zu den Funktionen der `pr`-Familie ein rein funktionales Pendant: `pr-str`, `prn-str`, `print-str` und `println-str`. Der Vollständigkeit halber sei auch die Funktion `with-out-str` erwähnt. In all diesen Fällen wird `*out*` sozusagen „vorübergehend“ an eine Stringausgabe (*string writer*) gebunden. Die sich ergebende Zeichenkette ist der Rückgabewert der `-str`-Funktionen. Ein Beispiel:

```
user=> (println-str "The result is" 42)
"The result is 42\r\n"
```

1.4 Aufgaben

Die meisten Aufgaben verstehen sich als „Kontrollfragen“ zum Textverständnis. Sie finden viele der Antworten im voranstehenden Text „versteckt“.

▷ **Aufgabe 1.1** Was ist `*in*`?

▷ **Aufgabe 1.2** Was ist eine Reader-Form?

▷ **Aufgabe 1.3** Zählen Sie die primitiven Datentypen auf, die der Reader erkennt.

▷ **Aufgabe 1.4** Zählen Sie die zusammengesetzten Datentypen auf, die der Reader erkennt.

▷ **Aufgabe 1.5** Für welche Datentypen gibt es keine Reader-Forms?

▷ **Aufgabe 1.6** Wie sind die aufgezählten Datentypen, für die es Reader-Forms gibt, kodiert?

▷ **Aufgabe 1.7** Welche Bedeutung hat ein Enter für den Reader?

▷ **Aufgabe 1.8** Was sind Reader-Macros?

▷ **Aufgabe 1.9** Was ist ein „qualifiziertes“ Symbol?

▷ **Aufgabe 1.10** Der Reader verarbeitet Formen. Was gibt er an den Evaluator weiter? Erkannte Formen?

▷ **Aufgabe 1.11** Fassen Sie die Evalutionsregeln zusammen.

▷ **Aufgabe 1.12** Erläutern Sie die Evaluation von `+`.

▷ **Aufgabe 1.13** Erläutern Sie, die Evaluationen der Ausdrücke `[1 2 3]`, `[+ 2 3]`, `(1 2 3)`, `(+ 2 3)`, `[(+ 1 2) (+ 3 4)]`, `(first [1 2 3])`; raten Sie beim letzten Ausdruck.

▷ **Aufgabe 1.14** Die Eingabe des Ausdrucks `(1 2 3)` liefert einen Fehler. Wer ist für die Erkennung des Fehlers (nicht für die Ausgabe) verantwortlich? Der Reader, der Evaluator oder der Printer? Begründen Sie Ihre Antwort!

▷ **Aufgabe 1.15** Wie kann eine Liste als Datum behandelt werden?

▷ **Aufgabe 1.16** Wie kann eine als Datum behandelte Liste wieder evaluiert werden?

▷ **Aufgabe 1.17** Was ist ein Seiteneffekt?

▷ **Aufgabe 1.18** Was ist der Rückgabewert von `println` und `prn`?

▷ **Aufgabe 1.19** In welchem Format macht der Printer seine Ausgaben?

1.5 Lösungen

Aufgabe 1.1 `*in*` ist ein Symbol, das assoziiert ist mit einer Variablen, die als Wert einen Zeichenstrom hat.

Aufgabe 1.2 Eine Reader-Form ist eine Kodierung für Ausdrücke, die Datentypen repräsentieren. Was heißt Kodierung? Es muss eindeutig unterscheidbar sein, welche Zeichenfolge welche Arten von Daten repräsentiert.

Aufgabe 1.3 Symbole und Literale (Schlüsselwörter, Einzelzeichen, Zeichenketten, Zahlen, `true`, `false` und `nil`).

Aufgabe 1.4 Listen, Vektoren, Abbildungen (Maps) und Mengen (Sets).

Aufgabe 1.5 Zum Beispiel Namensräume und Funktionen.

Aufgabe 1.6 Siehe Auflistung in Kapitel 1.1.1.

Aufgabe 1.7 Ein ENTER schickt einen „Schwung“ neuer Zeichen zur Verarbeitung an den Reader. Der Reader entscheidet, ob eine Form unvollständig ist oder nicht und wartet entweder auf weiteren Input oder reicht erkannte Formen an den Evaluator weiter. Ungültige Kodierungen für Formen weist der Reader als fehlerhaft ab.

Aufgabe 1.8 Bei Erkennung eines Makrozeichens wird ein Reader-Makro aktiviert. Beispiele sind das Quote-Zeichen und das Backslash-Zeichen. Das aktivierte Reader-Makro ersetzt das Makrozeichen durch andere Zeichen. Im Falle des Quote-Zeichens werden sogar hinter der dem Quote folgenden Form Zeichen eingefügt, aus `'<form>` wird `(quote <form>)`. Ein Reader-Makro ist ein Mechanismus zur Zeichenersetzung.

Aufgabe 1.9 Das sind Symbole, die in ihrem Namen ein „`/`“ oder „`.`“ beinhalten. Ohne weitere Details: Damit kann auf Namensräume bzw. Java-Klassen und -Methoden Bezug genommen werden.

Aufgabe 1.10 Der Reader verarbeitet Formen (Kodierungen) und wandelt sie in Daten um. Zum Beispiel wird mit Hilfe der Form für Zahlen, die `42.0` vom Reader als gültige Zahlenkodierung erkannt und dann intern als Zahl `42.0` weiter verarbeitet. Der Reader reicht Daten und keine Formen an den Evaluator.

Aufgabe 1.11 Sie finden die Evaluationsregeln in Kapitel 1.2.1. Beachten Sie bei Ihrer Zusammenfassung, dass Sie keine Details vernachlässigen.

Aufgabe 1.12 Nutzen Sie Clojure, um Ihre Antwort zu überprüfen.

Aufgabe 1.13 Nutzen Sie Clojure, um Ihre Antwort zu überprüfen.

Aufgabe 1.14 Es ist nicht der Reader. Der Reader erkennt sowohl die Liste als auch die Elemente der Liste als gültige Formen. Es ist der Evaluator, der bei der Anwendung der Evaluationsregeln eine Verletzung des Regelwerks feststellt und meldet. Der Printer kommt nur bei erfolgreicher Evaluation zum Einsatz. Die Tatsache, dass der Evaluator eine Fehlerausgabe dem Ausgabestrom anfügt, heißt nicht, dass der Printer benötigt wird. Der Printer ist für die String-Repräsentation von Daten zuständig.

Aufgabe 1.15 Durch Quotierung: `(quote (1 2 3))` oder äquivalent mit dem Makrozeichen `'(1 2 3)`.

Aufgabe 1.16 Mit eval: `(eval '(+ 2 3))`.

Aufgabe 1.17 Der Aufruf einer Funktion hat immer einen Rückgabewert zur Folge, sofern Sie nicht unbeabsichtigt eine Art „Endlosschleife“ programmiert haben. Wenn sich abseits von dieser Rückgabe ein Zustand in Clojure

verändert hat, dann spricht man von einem Seiteneffekt. Ein Beispiel ist die Verwendung von `println` in einer Funktion. Ausgaben sind Seiteneffekte.

Aufgabe 1.18 In beiden Fällen ist die Rückgabe `nil`. Der Seiteneffekt ist, dass eine Zeichenfolge dem Ausgabestrom hinzugefügt wird.

Aufgabe 1.19 Der Printer macht seine Ausgaben in einem Format, das vom Reader wieder erkannt und verarbeitet werden kann. Es gibt wenige Ausnahmen von dieser Regel. Zu diesen Ausnahmen gehören beispielsweise Funktionen und Namensräume.

Historie

22. Mai 2012	Schreibkorrekturen; mit Dank an Katharina Tetkowski und Manuel Wöhr; Lösungskapitel ist nun in Inhaltsverzeichnis aufgeführt; Update auf Clojure 1.4.0 in der Consolen-Anzeige
21. Mrz. 2012	Kleine Umstellung im Unterkapitel zu Spezialformen; ergänzende Auflistung aller Spezialformen in Clojure. Kleinere Klarstellung zur Erläuterung der 1. Evaluationsregel. Ergänzung der 2. Evaluationsregel; ein Symbol darf nicht zu einem Makro evaluieren. Überarbeitung der 3. Evaluationsregel; die Fälle werden nun deutlicher unterschieden. Ergänzender Hinweis um die Sonderfälle Keyword, Symbol, Vektor und Map. Einleitung zu den Sonderfällen (Kap. 1.2.5) vereinfacht, indem die Details zum IFn nicht mehr erwähnt werden.
7. Okt. 2011	Korrekturen an Lösungen vorgenommen. Dank an Marc Hesenius.
6. Okt. 2011	Lösungen zu Aufgaben hinzugefügt. Behebung von Inkonsistenzen in Schreibweise.
6. Okt. 2011	Einige Schreibkorrekturen. Mit Dank an Marc Hesenius.
10. Juli 2011	Drei Schreibkorrekturen. Mit Dank an Thomas Düllmann.
23. Mrz. 2011	Update der Copyright-Info.
18. Mrz. 2011	Erste Version.

2 Einfache Funktionen

Clojure ist eine funktionale Programmiersprache. In einer funktionalen Programmiersprache steht das Konzept der Funktion im Mittelpunkt. Und in der Tat ist damit der Funktionsbegriff aus der Mathematik gemeint.

Im Mathematik-Unterricht haben Sie Funktionen z.B. geschrieben als $f(x) = x^2$. Die Berechnung von Funktionswerten haben Sie dann notiert als $f(3) = 9$. In der Schule geht man in der Regel ein wenig darüber hinweg, sauber zu unterscheiden zwischen der Definition einer Funktion und der Anwendung einer Funktion.

- Eine **Funktionsdefinition** wie $f(x) = x^2$ legt fest, dass die Funktion f heißt, ein Argument namens x hat und ihr „Verhalten“ durch x^2 beschrieben ist.
- Eine **Funktionsanwendung** wie $f(9)$ wendet die Funktion f auf den Wert 9 an. Der Wert 9 ist nun der konkrete Wert für das Argument x. In dem Ausdruck x^2 wird x durch den Wert 9 ersetzt (der Fachausdruck ist „substituiert“). Das Ergebnis von $9^2 = 81$ ist das Resultat der Funktionsanwendung.

2.1 Definition und Anwendung von Funktionen

Clojure behandelt den Namen einer Funktion und die eigentliche Funktion als zwei verschiedene Dinge. Definieren wir beispielsweise die Quadrat-Funktion; wir nennen die Quadratfunktion hier nicht `f`, sondern `sq` für das englische „square“ (quadriere).

```
user=> (def sq (fn [x] (* x x)))
#'user/sq
```

Das Symbol `sq` wird über die Spezialform `def` (für *define*, definieren) gebunden an das Ergebnis der Evaluation des Ausdrucks `(fn [x] (* x x))`. Das Symbol `sq` übernimmt die Aufgabe eines Funktionsnamens.

Der Ausdruck `(fn [x] (* x x))` (ebenfalls eine Spezialform) evaluiert zu einer Funktion, die ein Argument namens `x` hat und über den sogenannten Funktionsrumpf `(* x x)` definiert ist. Dieser Gedanke ist für viele Programmierer neu und gewöhnungsbedürftig: Eine Funktion ist ebenso ein Datenwert wie es z.B. eine Zahl oder ein Vektor ist.

Wenn Sie die Funktionsform alleine über die REPL eingeben, erhalten Sie als Ergebnis eine Funktion.

```
user=> (fn [x] (* x x))
#<user$eval4$fn__5 user$eval4$fn__5@1963b3e>
```

Die Rückgabe des Printers auf der Konsole repräsentiert zwar eine Funktion, aber sie ist zugegebenermaßen sehr kryptisch. Clojure verrät Ihnen einige interne Details wie z.B. die hexadezimale Speicheradresse (hier `1963b3e`), unter der die Funktion abgespeichert ist. Das sind Details, die uns eigentlich nicht kümmern brauchen.

Darf ich Ihnen einen kleinen Trick verraten, wie Sie sich die Evaluation einer Funktion gedanklich vorstellen können? Nehmen Sie an, dass die `fn`-Spezialform zu einer fast identischen Form evaluiert, nur ersetzen wir die äußeren runden Klammern durch spitze Klammern, um den Unterschied zwischen der `fn`-Funktionsform und der Funktion an sich deutlich zu machen. Das macht das Evaluationsergebnis anschaulich und lässt Sie nicht vergessen, wie Argumentliste und Funktionsrumpf definiert sind.

Stellen Sie sich also vor, dass der obige Funktionsausdruck zur dieser Funktionsnotation evaluiert; gedanklich sieht die Ausgabe auf der Konsole dann wie folgt aus:

```
user=> (fn [x] (* x x))
<fn [x] (* x x)>
```

Wir haben eingangs mit der `def`-Spezialform das Symbol `sq` an diese Funktion gebunden. Folglich evaluiert fortan das Symbol zu dem an ihn gebundenen Wert, die Funktion. Gedanklich sehen wir also auf der Konsole als Ergebnis der Evaluation von `sq`:

```
user=> sq
<fn [x] (* x x)>
```

Tatsächlich liefert Clojure über die Konsole wiederum eine Ausgabe, die – damit sie nicht ganz so sinnfrei ist – einen Hinweis enthält, über welches Symbol die Funktion gebunden ist; das „`sq`“ taucht in der Repräsentation des Rückgabewerts auf.

```
user=> sq
#<user$sq user$sq@15f7107>
```

Wenn Sie gedanklich den Rückgabewert als Funktionsform im Kopf behalten, wird Ihnen die Funktionsanwendung (*function application*), oft auch Funktionsaufruf genannt (*function call*), sofort zugänglich sein. Der Funktionsaufruf geschieht in der für Clojure spezifischen Programmform einer Liste. Nach unserem gedanklichen Modell entspricht dem die Form `(<fn ...> ...)`.

Betrachten wir die Evaluation des Ausdrucks `(sq 3)` genauer. Wir drücken über den Pfeil „`=>`“ das Verhältnis „evaluiert zu“ aus.

Rekapitulieren wir die Evaluationsregeln für einen Listenausdruck. Ist der linkeste Ausdruck in der Liste keine Spezialform (wie z.B. `def` oder `fn`) und kein Makro, so wird der Wert evaluiert; er *muss* zu einer Funktion evaluieren. Da wir das Symbol `sq` entsprechend gebunden haben, evaluiert es zu einer Funktion. Anschließend werden alle weiteren Teilausdrücke des Listenausdrucks evaluiert; die Zahl `3` evaluiert zu sich selbst. Es ergibt sich also nach unserem Vorstellungsmodell:

```
(sq 3)
=> (<fn [x] (* x x)> 3)
```

Nun greift die Funktionsanwendung. Die evaluierten Teilausdrücke bilden die Argumentwerte und werden gemäß der Funktionsform im Funktionsrumpf an den entsprechenden Stellen eingesetzt. In unserem Beispiel sorgt die Ersetzung von `x` mit `3` dafür, dass aus dem Ausdruck `(* x x)` der Ausdruck `(* 3 3)` wird. Die Evaluation von `(* 3 3)` liefert `9` und damit das Ergebnis der Funktionsanwendung.

```
(sq 3)
```

```
=>  (<fn [x] (* x x)> 3)
=> (* 3 3)
=> 9
```

Clojure liefert das Ergebnis ebenso, ohne die Zwischenschritte transparent zu machen.

```
user=> (sq 3)
9
```

Das Ganze mag Ihnen auf den ersten Eindruck kompliziert vorkommen, Sie werden sich daran aber schnell gewöhnen. Denn ohne Funktionen können Sie in Clojure nicht programmieren. Der Umgang mit Funktionen gehört zum „Alltagsgeschäft“ in einer funktionalen Sprache.

▷ **Aufgabe 2.1** Definieren Sie eine Funktion `cube`, die nicht das Quadrat, sondern den Kubikwert einer gegebenen Zahl ausrechnet.

▷ **Aufgabe 2.2** Zu was evaluiert `('(fn [x] x) 3)`?

2.2 Funktionen sind Abstraktionen

Die `fn`-Form liefert für sich alleine genommen eine Funktion, zu der es kein Symbol gibt, das an die Funktion gebunden ist. Man spricht deshalb bei der `fn`-Form von einer „anonymen“ (namenlosen) Funktion.

Da die Definition von Funktionen so zentral für eine Sprache wie Clojure ist und eine Symbolbindung der Funktion einen Namen gibt und damit ihrer Anonymität enthebt, gibt es mit `defn` (*define function*, definiere Funktion) eine Kurzform für den kombinierten Einsatz von `def` und `fn`.

Mit `defn` schreibt sich die Definition der Quadrat-Funktion als

```
user=> (defn sq [x] (* x x))
#'user/sq
```

`defn` liefert eine Variable zurück. Die vom Printer mit einem „`#'`“ beginnende Ausgabe ist, würde sie vom Reader wieder gelesen werden, ein Reader-Makro, das gleichbedeutend ist mit `(var sq)`. Doch das nur am Rande. Machen Sie bitte nicht den Fehler, Variablen mit Symbolen durcheinander zu bringen. Variablen werden später im Kapitel über Referenzen behandelt. Hier genügt es zu wissen, dass nach einem `defn` das Symbol an die definierte Funktion gebunden ist.

Die Definition von Funktionen über die `fn`-Form wird in funktionalen Sprachen als Abstraktion (*abstraction*) bezeichnet. Noch deutlicher wird das, wenn die Funktion per `def` bzw. `defn` an ein Symbol gebunden ist. Das Symbol ist ein Stellvertreter für eine Funktion, die von den Details der Programmausführung abstrahiert. So wird ein (möglichst aussagekräftiger) Name für das Symbol zum Repräsentanten einer Abstraktion.

Zur Laufzeit, bei der Evaluation von Programmausdrücken, werden die Abstraktionen ersetzt durch den definierten Funktionsrumpf bei entsprechender Besetzung der Argumentwerte. Diese Ersetzung, die Auflösung der Abstraktion, wird auch als Verfeinerung (*refinement*) bezeichnet.

Dieser Verfeinerungsprozess kann nicht beliebig fortgesetzt werden. Es muss grundlegende Funktionen geben, die nicht über einen Clojure-Ausdruck definiert sind und somit nicht mittels anderer Funktionen aufgebaut sind. Solche Funktionen kapseln eine fest eingebaute Funktionalität von

Clojure; sie nennt man primitive (*primitive*) Funktionen. In Clojure ist es allerdings schwer, konkret von primitiven Funktionen zu reden, da Clojure einen Mechanismus hat, auf sämtliche Funktionen der zur JVM gehörigen Java-API zuzugreifen. Aus der Sicht sieht jeder Aufruf einer Java-Methode wie eine primitive Funktion aus.

Ob ein Wert wirklich eine Funktion ist (primitiv oder nicht), lässt sich mit `fn?` erfragen. Die Antwort darauf ist entweder ein „wahr“ (*true*) oder ein „falsch“ (*false*).

```
user=> (fn? sq)
true
user=> (fn? (fn [x] (* x x)))
true
user=> (fn? 3)
false
```

▷ **Aufgabe 2.3** Beschreiben Sie die Evaluation von `(fn? (fn [x] (* x x)))` Schritt für Schritt.

2.3 Mehr Argumente

Natürlich kann eine Funktion auch mehr als ein Argument haben. Nicht selten erwartet eine Funktion zwei, drei, vier Argumente oder auch mehr. Es gibt aber auch Funktionen, die kein Argument erwarten. Funktionen, die bei ihrem Aufruf immer das gleiche Ergebnis liefern, da es keine Abhängigkeit von einem Eingabeparameter gibt, heißen „konstante Funktionen“.

Die folgende konstante Funktion liefert immer eine `7`. Da der Funktionsaufruf ohne weitere Parameter erfolgt, ist in der Liste außer der `fn`-Form kein weiterer Ausdruck zu finden. Daher die Doppelklammern; ohne Doppelklammern kommt nur die Funktion als Evaluationsergebnis zurück.

```
user=> ((fn [] 7))
7
user=> (fn [] 7)
#<user$eval33$fn__34 user$eval33$fn__34@173ec72>
```

Funktionen, die mehrere Argumente entgegen nehmen sollen, sind mit mehreren (namentlich unterschiedlichen) Argumenten in der Argumentenliste zu versehen. Man ist in der Namensgebung der Argumente weitgehend frei.

```
user=> (defn myf [fst snd thrd] (list fst snd thrd))
#'user/myf
user=> (myf 3 7 9)
(3 7 9)
```

Die Funktion `myf` gibt die Argumentwerte in einer Liste verpackt zurück.

Soll neben einer fixen Anzahl an Argumenten die Angabe weiterer, optionaler Werte an die Funktion möglich sein, so kümmert sich ein `&`-Argument darum. Alles, was „zuviel“ an Werten an die Funktion übergeben wird, sammelt das `&`-Argument ein. Sind keine überzähligen Argumentwerte gegeben, wird das `&`-Argument auf `nil` (engl. für „nichts“) gesetzt.

```
user=> (defn g [x y & z] (list x y z))
#'user/g
user=> (g 1 2 3 4 5)
(1 2 (3 4 5))
user=> (g 1 2)
(1 2 nil)
```

Zu wenige Argumentwerte sorgen für eine Fehlermeldung; Clojure beschwert sich über eine falsche Anzahl an Argumenten.

```
user=> (g 1)
ArityException Wrong number of args (1) passed to: user$g ...
user=> (myf)
ArityException Wrong number of args (0) passed to: user$myf ...
```

Man kann die Fehlermeldung verhindern, indem man `partial` nutzt. Man drückt damit aus, dass man die Funktion nur mit den teilweise (*partial*) vorhandenen Argumenten aufruft. Als Ergebnis liefert `partial` eine anonyme Funktion zurück, die bereit ist, noch fehlende Argumentwerte entgegen zu nehmen.

```
user=> (partial myf 1)
#<core$partial$fn__3794 clojure.core$partial$fn__3794@fd918a>
user=> ((partial myf 1) 2 3)
(1 2 3)
```

Die Funktion `partial` erwartet also eine Funktion als erstes Argument und dann beliebig viele, weitere (partielle) Argumente für die Funktion. Die zurück gegebene anonyme Funktion kann dann mit den noch fehlenden Argumenten aufgerufen werden, die ihrerseits die an `partial` übergebene Funktion mit allen bisherigen Argumentwerten aufruft.

▷ **Aufgabe 2.4** Erläutern Sie die Evaluation von `((partial myf 1) 2 3)`.

Clojure hat eine Vielzahl von Funktionen fertig im Gepäck wie z.B. die arithmetischen Funktionen für das Rechnen mit Zahlen. Sie verstehen nun die Angabe der Dokumentation, beispielsweise für die Funktion `+`, wenn es dort heißt:

```
user=> (doc +)
-------------------------
clojure.core/+
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'
nil
```

Die Zeile nach `clojure.core/+` sagt aus, wie viele Argumentwerte die Additionsfunktion erwartet: entweder kein Argument (`[]`), ein Argument (`[x]`), zwei Argumente (`[x y]`) oder mehr (`[x y & more]`). Ohne Argumente aufgerufen, gibt `+` das neutrale Elemente der Addition, eine `0` zurück.

```
user=> (+)
0
user=> (+ 2)
2
user=> (+ 2 3)
```



```
5
user=> (+ 2 3 7 4)
16
```

Eine Funktion wie `+`, die für verschiedene Anzahlen von Argumenten definiert ist, heißt „variadische Funktion“ (*variadic function*).

Mit `doc` können Sie über die Console die Clojure-Dokumentation zur einer Funktion anfordern; gleiches gilt für Makros und Spezialformen, die wir an anderer Stelle behandeln.

▷ **Aufgabe 2.5** Die Taylorreihenentwicklung für die Sinus-Funktion lautet

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Angenähert auf den dritten Term reduziert sich die Berechnung zu

$$\sin(x) \approx \frac{x}{1} - \frac{x^3}{6} + \frac{x^5}{120}$$

Schreiben Sie eine Funktion `mysin`, die diese reduzierte Berechnungsformel umsetzt.

▷ **Aufgabe 2.6** Die Normalform der quadratischen Funktion $f(x) = x^2 + px + q$ hat zwei Lösungen für die Nullstelle $f(x) = 0$. Die Lösungen lauten

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$

Schreiben Sie eine Funktion `x1` und eine Funktion `x2` zur Berechnung der Nullstellen bei gegebenem `p` und `q`. Die Quadratwurzel kann mit der Funktion `Math/sqrt` berechnet werden; z.B. liefert `(Math/sqrt 4)` den Wert `2`.

▷ **Aufgabe 2.7** Schreiben Sie eine Funktion `f`, die ohne Argumente aufgerufen den Wert `3` zurück gibt, d.h. `(f)` ergibt `3`.

▷ **Aufgabe 2.8** Schreiben Sie eine Funktion `g` für die gilt, dass `((g))` den Wert `3` ergibt. Verwenden Sie die Funktion `f` aus Aufgabe 2.7.

▷ **Aufgabe 2.9** Schreiben Sie eine Funktion `h` für die gilt, dass `(((h)))` ebenfalls `3` ergibt. Greifen Sie zur Lösung ausschließlich auf die Funktion `f` aus Aufgabe 2.7 zurück.

2.4 `#(`Funktionen in Kurzform)

Clojure-Programmierer(innen) lieben es kurz und bündig, vor allem, wenn es um Funktionen geht, die in eine Zeile passen. Für solche Einzeiler gilt selbst die `fn`-Form als zu aufwendig. Ein Reader-Makro wandelt jeden mit einem „`#(`“ beginnenden und mit einer schließenden Klammer „`)`“ endenden Ausdruck in eine Funktion um. Die Anzahl der Argumente leitet sich indirekt ab aus den Symbolen, die mit einem Prozentzeichen beginnen und möglicherweise eine positive Ganzzahl folgen lassen.

Ein paar Beispiele zeigen, welche `fn`-Form dem Reader-Makro für Kurzfunktionen entspricht. Der Pfeil `-->` meint „entspricht“. Das erste Beispiel ist die anonyme Quadratfunktion.

```
#(* % %)
--> (fn [%] (* % %))

user=> (#(* % %) 5)
25
```

Ein führendes Prozentzeichen markiert einen gültigen Symbolnamen. In der Kurzform weist das alleinige Prozentzeichen nur ein Argument aus. Sind mehrere Argumente gefragt, so sind sie – je nach benötigter Anzahl – durchzunummerieren: `%1`, `%2`, `%3` usw. Das nächste Beispiel entspricht dem Satz des Pythagoras $a^2 + b^2 = c^2$, es berechnet die Summe der Quadrate der Katheten eines rechtwinkligen Dreiecks.

```
#(+ (* %1 %1) (* %2 %2))
--> (fn [%1 %2] (+ (* %1 %1) (* %2 %2)))
```

Der Aufruf der Kurzform mit den Werten `2` und `3` liefert:

```
user=> (#(+ (* %1 %1) (* %2 %2)) 2 3)
13
```

Wie unschwer zu erraten ist, sammelt `%&` alle optionalen Argumente in einer Liste auf. Ein Beispiel und seine Anwendung:

```
#(list %1 %&)
--> (fn [%1 & %&] (list %1 %&))

user=> (#(list %1 %&) 1 2 3)
(1 (2 3))
```

Diese Kurzformen für Funktionen werden sehr oft im Zusammenhang mit Funktionen höherer Ordnung verwendet. Mit Funktionen höherer Ordnung werden wir uns in einem späteren Kapitel beschäftigen.

2.5 `eval` und `apply`

Clojure stellt Ihnen über die Funktionen `eval` und `apply` die Evaluation (*evaluation*) von Clojure-Ausdrücken und die Funktionsanwendung (*function application*) explizit zur Verfügung.

Die Funktion `eval` erwartet als Argument einen beliebigen Clojure-Ausdruck und liefert das Ergebnis der Evaluation des Arguments zurück.

```
user=> (eval '(+ 2 3))
5
```

▷ **Aufgabe 2.10** Warum wird im voranstehenden Beispiel der an `eval` übergebene Ausdruck quotiert? Genügt es nicht `(eval (+ 2 3))` zu schreiben?

▷ **Aufgabe 2.11** Was ist das Ergebnis von `(eval (list '+ 2 3))`?

▷ **Aufgabe 2.12** Was ergibt `(eval (fn [x] x))`?

Das Beachtliche an `eval` ist, dass Sie Listenausdrücke manipulieren und nachfolgend als Programm auswerten können. Zum Beispiel kann man mit `concat` zwei Listen verbinden (konkatenieren, *concatenate*). Die daraus entstandene Liste kann dann per `eval` als Programm behandelt werden.

```
user=> (concat '(+) '(2 3))
(+ 2 3)
user=> (eval (concat '(+) '(2 3)))
5
```

Ohne `eval` wäre das nicht möglich. Versuchen Sie, den aus `concat` hervorgehenden Listenausdruck `(+ 2 3)` *ohne* `eval` von Clojure evaluieren zu lassen – es wird Ihnen nicht gelingen! In `eval` liegt die Mächtigkeit von Clojure verborgen, zur Laufzeit Programme als Daten (sprich Listen) manipulieren und auf Wunsch als Programme interpretieren zu können. Das nennt man Meta-Programmierung. Eine Fähigkeit, mit der nur wenige Programmiersprachen ausgestattet sind und die in Clojure bestechend einfach zu handhaben ist.

Für die Anwendung einer Funktion auf ihre Argumente gibt es die Funktion `apply`. Als erstes Argument ist eine Funktion zu übergeben. Es folgen beliebig viele Argumente, mit denen die Funktion aufgerufen wird. Als letztes Argument muss eine Liste übergeben werden; die Liste entspricht den „überzähligen“ `&`-Argumenten bei „normalen“ Funktionsaufrufen, siehe Kap. 2.3 (Seite 24). Alle voranstehenden Argumente werden der Liste unter Beibehaltung der Reihenfolge hinzugefügt. `apply` wendet die Funktion auf die so erzeugte Argumentliste an.

```
user=> (apply sq 3 ())
9
user=> (apply sq '(3))
9
user=> (apply + '(1 2 3))
6
```

▷ **Aufgabe 2.13** Warum erzeugt der Ausdruck `(apply sq (3))` eine Fehlermeldung?

▷ **Aufgabe 2.14** Schauen Sie sich mit `(source partial)` die Definition von `partial` an. Versuchen Sie, die Arbeitsweise des Codes zu erklären.

Grundsätzlich lässt sich jeder `apply`-Ausdruck schematisch in einen entsprechenden `eval`-Ausdruck umwandeln. Damit scheint `apply` überflüssig zu sein. Tatsächlich ist die Funktionsanwendung ein Unterpunkt in den Evaluationsregeln. Clojure legt diesen Aspekt der Evaluation mit `apply` offen und stellt ihn für den Anwender bzw. die Anwenderin zur Verfügung. Eine Wandlung eines `apply`-Ausdrucks in einen `eval`-Ausdruck würde die Funktionsanwendung nur „covern“ – was nichts an der Tatsache ändert, dass ein `eval` logisch auf ein `apply` angewiesen ist, wenn ein Funktionsausdruck vorliegt.

Es ist vielmehr so, dass Sie – wenn `apply` verfügbar ist – die Funktion `eval` zur Auswertung von Ausdrücken nachprogrammieren können. Wenn Sie möchten, können Sie auch eine neue, andere Evaluationsstrategie umsetzen. Sie können sich auf diese Weise einen Meta-Interpreter bauen, der die Evaluationsregeln von Clojure nachempfindet oder abändert und damit Ihre Programme, die ja Listenausdrücke sind, auswerten. Auch das ist im Vergleich mit fast allen anderen Programmiersprachen eine ungewöhnliche Eigenschaft. Mit einem Meta-Interpretierer lässt sich das Ausführungsmodell einer Sprache sehr unkompliziert verändern.

Die Funktionen `eval` und `apply` werden gerne als das Yin und Yang Lisp-basierter Sprachen bezeichnet, da sie wechselseitig die beachtliche


Abbildung 2.1: Links das chinesische Yin-Yang-Symbol, rechts das Clojure-Logo (© Tom Hickey)

Mächtigkeit Lisp-basierter Sprachen (generell homoikonischer Sprachen) begründen. Sehr prominent hat das Buch *Stucture and Interpretation of Computer Programs* von ABELSON und SUSSMANN diesen mystischen Bezug zur chinesischen Denkkultur geprägt [AS96]; auf dem Titelbild des Buches ist ein Zauberer und das Bildsymbol für Yin und Yang zu sehen.[1]

Clojure greift den Bezug zum Yin-Yang-Symbol in seinem Logo auf, siehe Bild 2.1. Eingearbeitet ist der griechische Buchstabe λ (sprich „Lambda“), da die funktionale Programmierung auf den Lambda-Kalkül zurück geht.

2.6 Funktional gedacht: keine Zeit für Seiteneffekte

Die funktionale Programmierung baut auf dem mathematischen Konzept der Funktion auf – und damit ist eine sehr wichtige Eigenschaft verbunden, die ich gerne als das *„Gesetz der funktionalen Programmierung“* bezeichne:

- Gleicher Input, gleicher Output!

Damit ist folgendes gemeint: Rufe ich eine Funktion wiederholt mit den gleichen Argumentwerten auf (das ist der „gleiche Input“), dann liefert die Funktion immer den gleichen Ergebniswert zurück (das ist der „gleiche Output“).

Das klingt äußerst trivial, ist aber von entscheidender Bedeutung. Darin drückt sich die Unabhängigkeit von der Zeit und von sonstigen Umständen aus. Es macht keinen Unterschied, ob ich `(sq 3)` gestern, heute oder morgen in der Clojure-Konsole ausführe. Das Ergebnis beträgt immer `9`.

Mathematisch und aus softwaretechnischer Sicht ist das äußerst hilfreich. Funktionen sind meist einfach zu verstehen, der Programmcode liest sich oft wie eine Spezifikation, bisweilen kann man sogar Eigenschaften beweisen, wie z.B. das Laufzeitverhalten oder die Terminierung einer Funktion. Das sind Qualitäten, die die sogenannten imperativen Programmiersprachen wie z.B. Java und C# vermissen lassen und zunehmend zum guten Ruf der funktionalen Sprachen beigetragen haben.

Allerdings hat die Unabhängigkeit von der Zeit und sonstigen Umständen einen Haken. Formal kann man das wie folgt ausdrücken. Zu zwei beliebigen Zeitpunkten t_1 und t_2 führt die Anwendung einer Funktion f auf die Argumente $args$ stets zum selben Ergebnis; man nennt das „zeitliche Invarianz“:

$$f_{t_1}(args) = f_{t_2}(args)$$

[1] Das Buch ist frei zugänglich unter http://mitpress.mit.edu/sicp/ (2012-03-31).

Man kann die Zeit explizit in einer Funktion als Argument mit berücksichtigen, etwa in der Form $f(t, args)$ – wir werden das später in einem Kapitel tun, um den Zeitverlauf von Geldtransaktionen auf einem Konto abzubilden. Die explizite Einbeziehung der Zeit als Argument erlaubt nur die *Simulation* von Zeit, jedoch keine Interaktion mit der *Echtzeit*, der Welt, in der Sie und ich leben. Kurzum, aus funktionaler Sicht ist keine Interaktion mit der echten Welt möglich.

Das ist mehr als unbrauchbar und disqualifiziert eine funktionale Programmiersprache für den praktischen Einsatz. Es gilt, einen Kompromiss zu finden. Verschiedene funktionale Programmiersprachen haben darauf unterschiedliche Antworten gefunden.

Clojure schlägt einen sehr pragmatischen Weg ein. Die Daumenregel ist: Programmiere so viel wie möglich rein funktional und lagere Interaktionen mit der „Echtwelt“ in möglichst wenige „unreine“ Funktionen aus. Solche „unreinen“ Funktionen (*impure functions*) werden auch als Funktionen mit Seiteneffekt (*side effect*) bezeichnet.

Ein Seiteneffekt beeinflusst zum Beispiel die Ausgabe auf der Konsole. So ist zum Beispiel `println` eine Funktion mit einem Seiteneffekt.

```
user=> (println "Hello World")
Hello World
nil
```

Der Rückgabewert von `println` ist stets `nil`, wie es die letzte Zeile zeigt. Als Seiteneffekt gibt `println` jedoch den String `"Hello World"` über die Konsole aus.

Ein Seiteneffekt ganz anderer Art ist die Eingabe z.B. per `read`.

```
user=> (* (read) 5)
4
20
```

Clojure wartet in der REPL auf eine Eingabe (hier `4`) und setzt die Evaluation nach der Eingabe fort.

▷ **Aufgabe 2.15** Wie Sie wissen, fordern Sie mit `doc` die Dokumentation zu einer Funktion, einem Makro oder einer Spezialform an. Arbeitet `doc` mit Seiteneffekten?

Sie werden feststellen, dass ich im ganzen Buch Funktionen so gut wie nie mit Seiteneffekten ausstatte. Ansonsten wäre der Sinn verfehlt, eine Einführung in die funktionale Programmierung mit Clojure zu bieten. Außerdem werden Sie im Laufe der Zeit merken, wie klar und verständlich funktionale Programme sind.

2.7 Lösungen

Aufgabe 2.1 Vielleicht haben Sie die Definition von `cube` korrekt erraten können.

```
(def cube (fn [x] (* x x x)))
```

Auch die Funktionsrümpfe `(* x (* x x))` bzw. `(* (* x x) x)` stellen eine richtige Lösung dar.

Aufgabe 2.2 Die Quotierung verhindert die Evaluation des Funktionsausdrucks. Ohne Funktion kann die äußere Klammerform jedoch nicht evaluiert werden. Clojure meldet einen Fehler. Mit der Substitutionstechnik sieht das wie folgt aus.

```
('(fn [x] x) 3)
=> ((fn [x] x) 3)
=> Fehler
```

Begehen Sie nicht den Irrtum, die `fn`-Form in Gedanken zweimal auszuwerten, nur damit es passt und die Evaluation erfolgreich sein kann. Halten Sie sich strikt an die Evaluationsregeln! Clojure tut es auch.

```
user=> ('(fn [x] x) 3)
ClassCastException ...
```

Aufgabe 2.3 Die Evaluationsschritte gestalten sich wie folgt:

```
(fn? (fn [x] (* x x)))
=> (fn? <fn [x] (* x x)>)
=> true
```

Aufgabe 2.4 Die Evaluation eines Listenausdrucks beginnt mit dem Element ganz links. Der Ausdruck `(partial myf 1)` evaluiert zu einer anonymen Funktion, die weitere Argumente erwartet. Hätte die Evaluation keine Funktion ergeben, hätte Clojure die Evaluation abgebrochen. Die anonyme Funktion wird mit den Werten `2` und `3` aufgerufen, was nun dem Aufruf von `(myf 1 2 3)` gleich kommt.

Aufgabe 2.5 Die Lösung trifft für kleine Werte sehr gut den mittels `Math/sin` berechneten Sinuswert. Für größere Werte ist die Abweichung auffallend. (Zur Erinnerung: Der Sinus ist eine über 2π periodische Funktion.)

```
user=> (defn mysin [x]
         (+ x (/ (* x x x) -6) (/ (* x x x x x) 120)))
#'user/mysin
user=> (mysin 0.3)
0.29552025
user=> (Math/sin 0.3)
0.29552020666133955
user=> (mysin 2.3)
0.8085285833333337
user=> (Math/sin 2.3)
0.7457052121767203
```

Die deutsche Wikipediaseite zur Taylorreihe[2] veranschaulicht Ihnen sehr schön den Grund für diese Abweichung am Beispiel der Sinus-Funktion und in einer Animation für die Cosinus-Funktion: Mit jedem weiteren Term in der Taylorreihe schmiegt sich die angenäherte Funktion stückweise an den gewünschten Funktionsverlauf an. Nahe Null erfassen die ersten

[2] http://de.wikipedia.org/wiki/Taylorreihe

drei Terme der Reihenentwicklung den Sinusverlauf auf mehrere Nachkommastellen genau, für größere Werte von x schießt die Annäherung regelrecht „ins Kraut“.

Aufgabe 2.6 Um die Funktionen zur Berechnung von `x1` und `x2` zu vereinfachen, erstellen wir die Hilfsfunktion `Dsqrt`, die die Wurzel aus der sogenannten Determinante $(p/2)^2 - q$ liefert.

```
(defn Dsqrt [p q] (Math/sqrt (- (/ (* p p) 4) q)))
(defn x1 [p q] (+ (/ p -2) (Dsqrt p q)))
(defn x2 [p q] (- (/ p -2) (Dsqrt p q)))
```

Die Richtigkeit der Gleichungen lässt sich an einem Beispiel überprüfen. Angenommen $p = 3$ und $q = -9$, dann muss gelten:

```
user=> (+ (* (x1 3 -9) (x1 3 -9)) (* 3 (x1 3 -9)) -9)
0.0
user=> (+ (* (x2 3 -9) (x2 3 -9)) (* 3 (x2 3 -9)) -9)
1.7763568394002505E-15
```

Im Rahmen der Genauigkeit von Fließkommazahlen, ist das zweite Ergebnis so klein, dass es praktisch Null ist.

Aufgabe 2.7 `(defn f [] 3)`

Aufgabe 2.8 `(defn g [] f)`

Aufgabe 2.9 `(defn h [] (fn [] f))`

Aufgabe 2.10 Beachten Sie, dass `eval` eine Funktion ist! Wird der an `eval` übergebene Ausdruck nicht quotiert, wird – gemäß den Evaluationsregeln – der Ausdruck bereits evaluiert, bevor(!) `eval` sich an die Evaluation macht. Das macht `eval` praktisch überflüssig, es sei denn, diese Form der doppelten Auswertung ist ausdrücklich gewollt.

Aufgabe 2.11 Diese Aufgabe ist ein Beispiel einer solchen gewollten „doppelten“ Auswertung, siehe auch Aufgabe 2.10. Zunächst evaluiert der innere Programmausdruck, der seinerseits dann von `eval` evaluiert wird. Das lässt sich schrittweise über die REPL nachvollziehen.

```
user=> (list '+ 2 3)
(+ 2 3)
user=> (eval (list '+ 2 3))
5
```

Ist das Argument zu `eval` quotiert, dann ergibt sich:

```
user=> (eval '(list '+ 2 3))
(+ 2 3)
```

Aufgabe 2.12 Der Ausdruck liefert den Beweis, dass eine Funktion zu sich selbst evaluiert. Der innere `fn`-Ausdruck evaluiert zu einer Funktion, die anschließend von `eval` evaluiert wird. Ein Test bestätigt die Evaluation einer Funktion zu sich selbst.

```
user=> ((eval (fn [x] x)) 3)
3
```

Übrigens nennt man die Funktion `(fn [x] x)` „Identitätsfunktion“ (*identity funtion*), da Eingabe und Ausgabe „identisch“ sind.

Aufgabe 2.13 Da `apply` eine Funktion ist, werden vor der Anwendung der Funktion

die Argumente zu `apply` ausgewertet. Der Ausdruck `(3)` ist gemäß den Evaluationsregeln keine gültige Programmform.

Aufgabe 2.14 Die Funktion `partial` kann mit verschiedenen Anzahlen an Argumenten umgehen; sie ist eine variadische Funktion. Sie erwartet immer eine Funktion `f` als Argument. Für jede Variante – ob ein, zwei, drei plus optionale Argumente geliefert werden – gibt es eine eigene Implementierung. Die Struktur der Implementierungen ist immer gleich.

`partial` liefert in allen Fällen eine Funktion zurück, die die übrigen Argumente mit `& args` erwartet. Wenn sie mit konkreten Werten aufgerufen wird, nutzt sie `apply`, um dann die Funktion `f` mit allen Argumenten auszuführen. Machen wir uns das am Beispiel aus dem Text klar und wenden die Substitutionstechnik an.

```
(partial myf 1)
=> (partial <fn [fst snd thrd] (list fst snd thrd)> 1)
=> (fn [& args] (apply <fn [fst snd thrd] ...> 1 args))
=> <fn [& args] (apply <fn [fst snd thrd] ...> 1 args)>
```

Ich habe mir erlaubt, den Rumpf der inneren Funktion mittels „`...`“ abzukürzen. Es kommt also eine Funktion zurück, die die Ausführung der inneren Funktion mittels `apply` verzögert. Das ist eine typisch funktionale Technik, die Evaluation eines Ausdrucks zu verzögern, indem man ihn in eine Funktion einbettet und die einbettende Funktion zurück gibt.

Evaluieren wir nun `((partial myf 1) 2 3)`. Der Ausdruck ganz links evaluiert zur eben ermittelten Funktion.

```
((partial myf 1) 2 3)
=> (<fn [& args] (apply <fn [fst snd trd] ...> 1 args)> 2 3)
=> (apply <fn [fst snd thrd] (list fst snd thrd)> 1 '(2 3))
=> (list 1 2 3)
=> (1 2 3)
```

Aufgabe 2.15 Mit `(doc doc)` erfahren Sie, dass der Rückgabewert von `doc` ein `nil` ist. Alles, was `doc` darüber hinaus auf der Konsole darstellt, ist ein Seiteneffekt.

Historie

22. Mai 2012 Schreibkorrektur; mit Dank an Johannes Hummel, Nicolas de Klerk und Katharina Tetkowski

10. Apr. 2012 Neues Unterkapitel, das das Reader-Makro `#(` als Kurzform für Funktionen einführt

3. Apr. 2012 Einarbeitung eines Kapitels „Funktionen sind Abstraktionen“ mit neuen Textanteilen; Aufgabe hinzugefügt; überflüssig gewordenen Absatz aus Kapitel „Mehr Argumente“ entfernt; Aufgabe zu `partial` mit ausführlicher Antwort; den Begriff der „primitiven“ Funktion eingebracht

2. Apr. 2012 Überarbeitung der beiden Eingangskapitel, um die Evaluation einer `fn`-Funktionsform besser zu veranschaulichen.

30. Mrz. 2012 Fehlenden Text zu `apply` ergänzt

29. Mrz. 2012 Ergänzung eines Kapitels zu `eval` und `apply`

28. Mrz. 2012 Kleinere Korrekturen

26. Mrz. 2012 Kleinen Schreibfehler korrigiert

20. Mrz. 2012 Ergänzung des Ausdrucks „zeitliche Invarianz“

19. Mrz. 2012 Erste Fassung fertig

16. Mrz. 2012 Schreibbeginn; weitgehender Verwurf einer älteren Fassung

3 Rekursive Funktionen

In der funktionalen Programmierung gibt es einzig die Rekursion (*recursion*) als Konstrukt, um die Idee der Wiederholung umzusetzen.

3.1 Die Fakultätsfunktion

Die Fakultätsfunktion ist so etwas wie das „Hello World“ der funktionalen Programmierung. Es hat sich eingebürgert, imperative Sprachen oft mit einem kleinen Programm einzuführen, das einen Gruß an die Welt per print-Anweisung auf der Konsole oder in einem Fenster erscheinen lässt. Der Wert dieses trivialen Programms ist zwar fraglich, ändert aber nichts an dieser Tradition.

Bei den funktionalen Sprachen verbietet sich eine Ausgabe per `print` oder `println`. Diese Funktionen haben Seiteneffekte. Sie geben `nil` zurück und erzeugen „nebenbei“ eine Ausgabe auf der Konsole. Der funktionale Programmierstil duldet Seiteneffekte nur als unabdingbare Notwendigkeit der Interaktion mit einer zustandsbehafteten Welt, nicht als Demonstration funktionaler Fähigkeiten.

Die Fakultätsfunktion ist ein idealer Kandidat zur Veranschaulichung funktionaler Konzepte. Sie ist leicht zu verstehen, und sie ist rekursiv definiert. Die Mathematik definiert die Fakultät $n!$ einer natürlichen Zahl $n \in \mathbb{N}_0$ als

$$n! = \begin{cases} 1 & \text{für } n \leq 1 \\ n \cdot (n-1)! & \text{für } n > 1 \end{cases}$$

Die Fakultät 0! ist 1, ebenso wie die Fakultät 1! den Wert 1 hat – das ist der Abbruchfall. Die Fakultät 2! berechnet sich nach der Gleichung zu $2 \cdot (2-1)! = 2 \cdot 1!$. Das Ergebnis ist $2 \cdot 1 = 2$. Die Fakultät 3! ist $3 \cdot 2! = 3 \cdot 2 \cdot 1 = 6$. Entsprechend liefert $4! = 4 \cdot 3! = 4 \cdot 6 = 24$ usw.

3.2 Rekursion in Clojure

Die mathematische Definition lässt sich praktisch eins zu eins als Clojure-Funktion hinschreiben – und zwar als Einzeiler. Sie ist ein direktes Abbild der mathematischen Definition.

```
(defn fact [n] (if (<= n 1) 1 (* n (fact (- n 1)))))
```

Diese Funktion – `fact` steht für *factorial* – vereint prägnant die entscheidenden Eigenschaften einer rekursiven, d.h. sich selbst aufrufenden Funktion: (1) Eine rekursive Funktion benötigt ein Abbruchkriterium, hier den Vergleich `(<= n 1)` in der `if`-Form. Ansonsten würde sich die Rekursion ins Unendliche, *ad infinitum*, fortsetzen. Eine Funktion, die niemals zu einem Ende kommt, ist unbrauchbar. (2) Der rekursive Aufruf muss so gestaltet sein, dass eine Annäherung an den Abbruchfall stattfindet und der Abbruch auch erreicht werden kann. Bei der Fakultätsfunktion sorgt das beständige Dekrementieren `(- n 1)` des Arguments dafür, dass der Abbruchfall angestrebt und eintreffen wird.

Man kann mathematische Verfahren bemühen, wie z.B. den Beweis per Induktion, um den Nachweis der kontinuierlichen Annäherung und des Erreichens des Abbruchfalls anzutreten. Bei der Fakultätsfunktion genügt die Methode des „scharfen Hinsehens“. Die beiden Eigenschaften rekursiver Funktionen sind offensichtlich erfüllt.

Die Möglichkeit des Einsatzes mathematischer Beweisverfahren ist ein deutlicher Vorteil funktionaler Programmiersprachen gegenüber imperativen Sprachen. Gut geschriebene Funktionen sind so aufgebaut, dass der Einsatz mathematischer Verfahren zwar eine Option ist, jedoch „scharfes Hinsehen“ und wenige Testfälle oftmals genügen. Mit Blick auf den Code und einigen Tests kann eine Funktion nur korrekt sein – oder die Tests schlagen unmittelbar fehl.

Nehmen Sie als Beispiel die Fakultätsfunktion. Wenn Sie wenige Fälle an der Konsole durchspielen und korrekte Ergebnisse erhalten, können Sie sicher sein, dass die Fakultätsfunktion für jede natürliche Zahl wie gewünscht arbeitet, sogar arbeiten muss – ein anderes Verhalten läßt die Rekursion gar nicht zu! Probieren Sie die zwei Abbruchfälle und einen beliebigen anderen Fall aus. Die Resultate sind korrekt. Unausweichlich wird auch jede andere Fakultät korrekt berechnet.

```
user=> (fact 0)
1
user=> (fact 1)
1
user=> (fact 4)
24
```

Beachten Sie: Der Aufruf von `fact` mit einer negativen Zahl oder mit einer Kommazahl, wie z.B. `(fact 3.5)` ist unzulässig. Die Fakultätsfunktion ist nur für $\mathbb{N}_0$, für natürliche Zahlen definiert. Man kann solche Annahmen durch eine sogenannte Vorbedingung (*precondition*) und/oder Typannotationen explizit machen. Es ist jedoch unnötig, missbräuchliche Aufrufe der Fakultätsfunktion abzufangen oder zu testen. Der Aufrufer ist selber für „abnorme“ Ergebnisse und Verhaltensweisen einer Funktion verantwortlich, wenn er sich nicht daran hält, gültige Argumente zu übergeben. Das ist ein wichtiges softwaretechnisches Prinzip.

3.3 Veranschaulichung der rekursiven Abarbeitung

Es gibt eine sehr einfache Technik, um sich die Abarbeitung einer Funktion zu veranschaulichen: durch das Ersetzen von Ausdrücken, meist Substitution bezeichnet, kann man sich schrittweise verdeutlichen, wie das Ergebnis einer rekursiven Funktionsevaluation zustande kommt.

Nehmen wir die Evaluation von `(fact 4)`. Gemäß den Evaluationsregeln wird die an das Symbol `fact` gebundene Funktion auf die Zahl `4` angewendet. Oder anders ausgedrückt: Die Fakultätsfunktion wird mit dem Wert `4` aufgerufen. Ein Funktionsaufruf bedeutet, dass der in der Funktionsdefinition angegebene Funktionsrumpf ausgeführt wird, wobei die Argumente mit dem übergebenen Argumentwert ersetzt werden. In unserem Fall evaluiert `(fact 4)` zu

```
(fact 4)
=> (if (<= 4 1) 1 (* 4 (fact (- 4 1))))
```

Der Pfeil `=>` hat die Bedeutung von „evaluiert zu“. In der `if`-Spezialform evaluiert der erste Ausdruck zu `false` (`4` ist nicht kleiner oder gleich `1`),

so dass der dritte Ausdruck innerhalb des `if` zur Auswertung kommt. Vor der erneuten Anwendung der Fakultätsfunktion wird `(- 4 1)` zu `3` evaluiert.

```
(fact 4)
=> (if (<= 4 1) 1 (* 4 (fact (- 4 1))))
=> (if false 1 (* 4 (fact (- 4 1))))
=> (* 4 (fact (- 4 1)))
=> (* 4 (fact 3))
```

Die äußere Multiplikation des letzten Ausdrucks kann nicht ausgeführt werden, da die Evaluationsregeln zunächst eine vollständige Auflösung der inneren Ausdrücke einfordern – solange ist Ausführung der Multiplikation anhängig, die Multiplikation „hängt“.

Die weitere Auflösung von `(fact 3)` läuft nach demselben Schema ab, wie die Evaluation von `(fact 4)`.

```
(* 4 (fact 3))
=> (* 4 (if (<= 3 1) 1 (* 3 (fact (- 3 1)))))
=> (* 4 (* 3 (fact 2)))
```

Dies setzt sich solange fort, bis mit `(fact 1)` der Abbruchfall der Rekursion erreicht ist. Für diesen Fall evaluiert die `if`-Form zu `1`.

```
(* 4 (* 3 (fact 2)))
=> (* 4 (* 3 (* 2 (fact 1))))
=> (* 4 (* 3 (* 2 1)))
```

Wir beobachten eine Zunahme an „hängenden“ Multiplikationen bis zum Abbruchfall. Mit dem Erreichen des Abbruchsfalls wird die Multiplikation von innen nach außen abgearbeitet.

```
(* 4 (* 3 (* 2 1)))
=> (* 4 (* 3 2))
=> (* 4 6)
=> 24
```

Mit dem letzten Schritt sind wir zum Ende der Evaluation von `(fact 4)` gekommen. Es ist das Endergebnis, was uns auch Clojure über die REPL anzeigt:

```
user=> (fact 4)
24
```

Mit Hilfe der Substitutionstechnik kann man sich die Evaluation beliebig komplizierter Ausdrücke verdeutlichen. Einzige Voraussetzung ist, dass die Funktion rein ist, d.h. Seiteneffekte keine Rolle spielen. Da Clojure jedoch sehr „geordnete“ Regeln für die Evaluation von Ausdrücken vorgibt, funktioniert die Substitutionstechnik sogar weitgehend bei Seiteneffekten, sofern Nebenläufigkeit keine Rolle spielt.

Clojure arbeitet intern ähnlich zur Substitutionstechnik, bearbeitet jedoch jeden einzelnen Funktionsaufruf (*function call*) separat und bildet die wachsende Verschachtlung der Ausdrücke über einen sogenannten Stapel ab. Ein Stapel (*stack*) ist eine einfache Datenstruktur auf der man, gleich einem Tellerstapel, Dinge ablegen und wieder entfernen kann.

Die Abarbeitung ist im Grunde die gleiche wie bei der reinen Subsitutionstechnik. Die Stapeltechnik vermeidet nur das Anwachsen des zu evaluierenden Ausdrucks und nutzt den Stapel dazu, um Funktionen jeweils

für sich auszuwerten und das Ergebnis im aufrufenden Ausdruck einzusetzen. Der Vorgang sei beispielhaft nachfolgend dargestellt; wir verzichten dabei auf die Zwischenschritte zur Auflösung eines Funktionsaufrufs, es sind dieselben wie oben. Die führende Zahl zeigt die Stapeltiefe an.

```
0: (fact 4)
0: => (* 4 (fact 3))
1: (fact 3)
1: => (* 3 (fact 2))
2: (fact 2)
2: => (* 2 (fact 1))
3: (fact 1)
3: => 1
```

Bei der nun folgenden Rückabwicklung des Stapelaufbaus wird der jeweilige Aufruf durch den Evaluationswert ersetzt (substituiert). Das Ergebnis von `(fact 1)` der Stapelebene 3 wird auf der Stapelebene 2 ersetzt durch den Wert `1`. Die Evaluation des sich ergebenden Ausdrucks `(* 2 1)` kann fortgesetzt werden.

```
2: (* 2 1)
2: => 2
1: (* 3 2)
1: => 6
0: (* 4 6)
0: => 24
```

▷ **Aufgabe 3.1** Die Fibonacci-Folge ist ebenfalls ein sehr beliebtes einführendes Beispiel in die funktionale Programmierung. Für $n \in \mathbb{N}_0$ ist $f_n = f_{n-1} + f_{n-2}$ mit $f_0 = 0$ und $f_1 = 1$. Setzen Sie die Berechnung rekursiv in Clojure um! Nennen Sie die Funktion `fib`. (Wenn Sie Ihre Lösung überprüfen möchten: `(fib 7)` ergibt `13`, `(fib 15)` ergibt `610`.)

3.4 Die Limitierung durch den Callstack

Das Fassungsvermögen dieses Aufrufstapels (*call stack*) ist begrenzt, weshalb zu hohe Rekursionstiefen zu einem Abbruch samt Fehlermeldung führen. Clojure beklagt in einem solchen Fall einen Überlauf (*overflow*) des Stapels als Stapelüberlauffehler (*stack overflow error*).

Wenn Sie ein wenig mit der Fakultätsfunktion experimentieren, werden Sie feststellen, dass Clojure für Werte größer `20` mit der Meldung einer Ausnahme (*exception*, ein Begriff der auch eingedeutscht genutzt wird) abbricht.

```
user=> (fact 20)
2432902008176640000
user=> (fact 21)
ArithmeticException integer overflow  [...]
```

Hier ist kein Stapelüberlauf die Ursache, sondern die berechnete Zahl ist zu groß für den zugedachten Zahlentypen, einer Ganzzahl vom Typ `java.lang.Long`. Es ist der Zahlentyp, der durch die Zahl beim Aufruf von `fact` gegeben ist, hier `20` bzw. `21`.

```
user=> (type 20)
java.lang.Long
```


Der Typ `java.lang.Long` stellt in 8 Bytes (64 Bits) Ganzzahlen im Zweierkomplement dar. Der Wertebereich dieses Typs reicht von -2^{63} bis $+2^{63}-1$. Man kann den Wertebereich bei Kenntnis der Java-Schnittstelle mit Clojure erfragen:[1]

```
user=> (. java.lang.Long MAX_VALUE)
9223372036854775807
user=> (. java.lang.Long MIN_VALUE)
-9223372036854775808
```

Für Argumentwerte größer `20` müssen Sie auf den Zahlentypen `BigInt` umsteigen, um weiterhin Ergebnisse von der Fakultätsfunktion zu erhalten. Der Zahlentyp `BigInt` realisiert Ganzzahlen beliebiger Größe. Der Zahlentyp passt im Bedarfsfall den benötigten Speicher zu Erfassung der Zahl an. Hängen Sie einer Ganzzahl ein `N` an, erkennt der Reader sie als `BigInt`. Tun Sie das nicht, sprengt die Fakultätsfunktion – wie Sie es oben im Beispiel sehen – den Wertebereich für `Long`-Integer.

```
user=> (fact 21N)
51090942171709440000N
```

Nun gibt die Fakultätsfunktion auch bei relativ großen Argumentwerten wie z.B. `(fact 1000N)` ein korrektes Ergebnis zurück. Ein Problem mit der Fakultätsfunktion gibt es dennoch. Wird das Argument zu groß, steigt Clojure mit einer Exception aus. Der Aufrufstapel (*call stack*) ist bei der Auflösung der Rekursion an seine Grenzen gekommen.

```
user=> (fact 50000N)
java.lang.StackOverflowError (NO_SOURCE_FILE:0)
```

Falls Ihr Rechner bei `50000N` nicht aussteigt, müssen Sie eventuell einen größeren Wert nehmen. Die maximale Stapeltiefe ist abhängig von individuellen Konfigurationswerten der JVM, der Java Virtual Maschine, die sich von Rechner zu Rechner unterscheiden kann.

Diese zunehmende, durch die Rekursion bedingte Aufruftiefe der `fact`-Funktion lässt den Aufrufstapel anwachsen und konsumiert mehr und mehr Speicher. Die JVM schiebt hier einen Riegel vor und bricht die Ausführung ab einer bestimmten Aufruftiefe ab; der Callstack darf nur bis zu einer fixen (aber veränderbaren) Größe anwachsen.

Dahinter steckt neben dem Speicherverbrauch eine einfache Heuristik. Eine Funktion, die sich so oft wiederholt aufruft, ist eventuell fehlerhaft programmiert, da sie das Abbruchkriterium bis zu dieser Tiefe nicht erreicht hat. In dem Sinne ist der Overflow des Callstacks ein sinnvoller Schutzmechanismus. Andererseits bleibt uns damit verwehrt, die Fakultätsfunktion für große Zahlen zu berechnen. Eine Alternative muss her!

3.5 Endrekursion als Alternative

In der funktionalen Programmierung ist man oft bestrebt, Funktionen so zu formulieren, dass die Rekursion ohne zunehmende Verschachtelung oder ein Anwachsen des Callstacks realisiert werden kann; das nennt sich *tail recursion*, auf deutsch auch als „Endrekursion“ bezeichnet. Dazu muss die Fakultätsfunktion umgeschrieben und dem Clojure-Compiler ein Hinweis gegeben werden. Die Rekursion kann dann vom Compiler in

[1] http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Long.html (2012-04-01)

eine Iteration, eine Schleife, umgewandelt werden. Das beschleunigt die Ausführung, da die Verwaltung des Callstacks entfällt.

Der Trick ist, die Fakultätsfunktion so umzubauen, dass es keine „hängende“ Multiplikation mehr gibt, die zur Verschachtelung bzw. zum Anwachsen des Callstacks führt. Das kann man mit einem Gedächtnis umsetzen, oft Akkumulator genannt, der das bisher erreichte Ergebnis der Fakultätsberechnung mitführt.

```
(defn fact
  ([n] (fact n 1))
  ([n acc]
    (if (<= n 1)
      acc
      (fact (- n 1) (* n acc)))))
```

Die „neue“ Fakultätsfunktion macht Gebrauch von dem Clojure-Feature, eine Funktion für verschiedene Anzahlen von Argumenten mit unterschiedlichen Funktionsrümpfen auszustatten (*variadic functions*). Der Aufruf von `(fact 4)` mit einem Argument führt zum Selbstaufruf `(fact 4 1)` mit zwei Argumenten. Das ist eine übliche Technik in Clojure, um das ursprüngliche Interface zu bewahren; die Fakultätsfunktion soll nachwievor mit nur einem Argument aufgerufen werden. Intern wird jedoch mit zwei Argumenten gearbeitet. Das zweite Argument ist der Akkumulator, der mit dem Wert `1` über den Aufruf mit einem Argument initialisiert wird.

Veranschaulichen wir uns die schrittweise Abarbeitung am Beispiel `(fact 4)`. Wieder wird der Funktionsaufruf durch den definierten Funktionsrumpf mit den ersetzten Argumentwerten realisiert. Wir notieren mit jedem Funktionsaufruf die Ebene des Callstacks.

```
0: (fact 4)
0: => (fact 4 1)
0: => (if (<= 4 1) 1 (fact (- 4 1) (* 4 1)))
0: => (fact 3 4)
```

Der Funktionsaufruf führt auf die nächste Ebene des Callstacks.

```
1: (fact 3 4)
1: => (if (<= 3 1) 4 (fact (- 3 1) (* 3 4)))
1: => (fact 2 12)
```

Das setzt sich fort bis zur 3. Ebene des Callstacks.

```
2: (fact 2 12)
2: => (if (<= 2 1) 12 (fact (- 2 1) (* 2 12)))
2: => (fact 1 24)
3: (fact 1 24)
3: => (if (<= 1 1) 24 (fact (- 1 1) (* 1 24)))
3: => 24
```

Nun muss der Callstack noch abgebaut werden – ein trivialer Vorgang, da es nirgends etwas einzusetzen und weiter zu evaluieren gibt.

```
2: => 24
1: => 24
0: => 24
```


Wie Sie sehen, kann auf den Aufbau des Callstacks vollständig verzichtet werden, er ist überflüssig. Immer wieder wiederholt sich einzig der Funktionsaufruf der Form `(fact n acc)`. Da dieser Ausdruck nicht mehr eingebettet ist in einen anderen Ausdruck, gibt es keine Notwendigkeit, einen Callstack aufzubauen. Genau das ist das Anliegen der Endrekursion: der rekursive Aufruf bleibt ohne Erhöhung des Callstacks.

Clojure braucht in einer endrekursiven Funktion einen expliziten Hinweis darauf, dass Sie den Callstack nicht anwachsen lassen wollen. Ohne diesen Hinweis ist auf meinem Rechner nachwievor bei `5000N` Schluß, da Clojure für jeden rekursiven Funktionsaufruf (unnötigerweise) den Callstack erhöht.

```
user=> (fact 5000N)
StackOverflowError ...
```

Ersetzen wir in unserem Programm den rekursiven `fact`-Aufruf durch `recur` (eine Spezialform), hat Clojure den benötigten Hinweis auf Endrekursion.

```
(defn fact
  ([n] (fact n 1))
  ([n acc]
    (if (<= n 1)
      acc
      (recur (- n 1) (* n acc)))))
```

Jetzt kann die Fakultät von `5000N` oder von noch viel größeren Zahlen berechnet werden, wenn Sie gewillt sind, ein paar Sekunden auf das Ergebnis zu warten.

Der Clojure-Compiler überprüft bei Gebrauch von `recur`, ob tatsächlich Endrekursion vorliegt. Die Analyse, die Clojure dazu vornimmt, ist relativ einfach: Immer wenn der rekursive Aufruf das Letzte ist, was es „am Ende“ zu tun gilt, liegt Endrekursion (*tail recursion*) vor – daher der Name.

Ist das nicht der Fall, bricht die Compilierung mit einer Fehlermeldung ab. Wenn Sie versuchen, die ursprüngliche Umsetzung der Fakultätsfunktion mit `recur` zu definieren, dann verweigert sich der Compiler.

```
user=> (defn fact [n] (if (<= n 1) 1 (* n (recur (- n 1)))))
CompilerException ...
```

Die Fehlermeldung enthält die Anmerkung „*Can only recur from tail position*“. Ein Aufruf, der sich in „Endposition“ (*tail position*) befindet, erlaubt Endrekrusion, sonst nicht.

▷ **Aufgabe 3.2** Wandeln Sie die rekursive Berechnung der Fibonacci-Zahlen in eine Funktion mit Endrekursion um! (Hinweis für Fortgeschrittene: Verwenden Sie kein `loop`.)

Es gibt funktionale Programmiersprachen, deren Compiler die Endrekursion automatisch erkennen und bei der Compilierung berücksichtigen. Die Auszeichnung der Endrekursion mit `recur` ist gewollt und hat Vorteile.

Erstens muss sich eine Entwicklerin bzw. ein Entwickler ausdrücklich Gedanken um Endrekursion machen. Man muss die Idee der Endrekursion verstehen, um sie gezielt einsetzen zu können. Zweitens hilft `recur` als Marker unmittelbar zu erkennen, ob eine Funktion endrekursiv ist oder

nicht. Eine Leserin bzw. ein Leser eines Programms muss eine Funktion nicht notwendigerweise vollständig intellektuell durchdringen um zu wissen, ob Endrekursion zum Tragen kommt. Würde der Compiler die Erkennung der Endrekursion im Hintergrund automatisiert durchführen, wäre sicher vielen Programmierer(innen) und Leser(innen) von Funktionen nicht klar, ob ein Überlauf des Callstacks auftreten kann oder Endrekursion vorliegt.

▷ **Aufgabe 3.3** Kann zu jeder rekursiven Funktion, die den Callstack anwachsen lässt, eine alternative Funktion gefunden werden, die mit Endrekursion arbeitet?

▷ **Aufgabe 3.4** Für größere Zahlen kommt die Fakultätsfunktion ans Rechnen und benötigt ihre Zeit für ein Ergebnis. Es gibt eine Näherungsformel für die Fakultätsfunktion. Recherchieren Sie: Wie lautet die Näherungsformel?

3.6 Schleifen mit `loop`

Die Endrekursion realisiert im Grund nichts anderes als eine Schleife (*loop*) – eine Wiederholung ohne Notwendigkeit, den Callstack anwachsen zu lassen. In imperativen Sprachen ist diese Wiederholungsform auch als Iteration bekannt. Da dieser Sonderfall der Rekursion eine besondere Auszeichnung verdient, bietet Clojure die Spezialform `loop` an: `loop` verhält sich wie eine anonyme Funktion und erwartet als Rekursionsaufruf ein `recur`. Mit anderen Worten: `loop` wird vom Clojure-Compiler immer endrekursiv behandelt.

Folgender Ausdruck entspricht dem Rumpf der endrekursiven Fakultätsfunktion für den beispielhaften Wert $n = 4$. So beginnt der Ausdruck mit der Bindung von `n` an `4` und dem initialen Wert `1` des Akkumulators `acc`.

```
user=> (loop [n 4 acc 1]
              (if (<= n 1) acc (recur (- n 1) (* n acc))))
24
```

Dieser `loop`-Form entspricht nahezu 1:1 die folgende `fn`-Form; die Funktion wird lediglich mit dem Startwert für `n` bzw. dem Initialwert für `acc` aufgerufen.

```
user=> ((fn [n acc] (if (<= n 1) acc (recur (- n 1) (* n acc)))) 4 1)
24
```

▷ **Aufgabe 3.5** Definieren Sie die Funktion `fact` zur Berechnung der Fakultät unter Nutzung von `loop`.

▷ **Aufgabe 3.6** Realisieren Sie die endrekursive Fibonacci-Funktion mit Hilfe von `loop`!

Sie stoßen bei Recherchen im Internet auf alternative Implementierungen zur Fakultätsfunktion wie z.B.[2]

```
(defn factorial [n]
  (reduce * (range 1 (inc n))))
```

[2] http://stackoverflow.com/questions/1662336/clojure-simple-factorial-causes-stack-overflow (2012-04-01)

Diese Implementierung ist zweifellos elegant und kurz. Da `range` eine *lazy sequence* zurück gibt, ist dieses Programm sogar sehr effizient.

Sie sollten um eine solche alternative Umsetzung wissen. Im Zusammenhang mit diesem Kapitel geht bei dieser Implementierung der Punkt der Rekursion und das wichtige Thema der Endrekursion verloren. Die Fakultätsfunktion als „Hello World“ der funktionalen Programmierung will besonders den Aspekt der Rekursion hervorheben. Funktionen wie `map` und ihr Gegenstück `reduce` gehören zu einer anderen „Baustelle“ in der funktionalen Programmierung.

3.7 Wechselseitige Rekursion

Die direkte Form der Rekursion ist leicht erkennbar: eine Funktionsdefinition hat in ihrem Funktionsrumpf den Selbstaufruf bzw. verwendet dort `recur` oder gar `loop`. Bei der indirekten Form der Rekursion geht die Funktion einen Umweg über mindestens eine andere Funktion, die dann ihrerseits die aufrufende Funktion aufruft. Das wird als wechselseitige oder auch als verschränkte Rekursion (*mutual recursion*) bezeichnet.

Das folgende Beispiel ist ebenso ein Klassiker zur Demonstration verschränkter Rekursion, wie es die Fakultätsfunktion für die „normale“ Rekursion ist. Mit Hilfe der beiden Funktionen `is-even?` und `is-odd?` wird bestimmt, ob eine Ganzzahl (*integer*) gerade oder ungerade ist.[3] Erinnern Sie sich: Funktionen, die nur entweder `true` oder `false` zurück geben, heißen Prädikate und werden per Konvention mit einem Fragezeichen im Namen abgeschlossen.

Das einleitende `declare` ist notwendig, da `is-even?` im Rumpf `is-odd?` verwendet, ohne dass `is-odd?` bereits definiert ist. Das bringt den Clojure-Compiler dazu einen Fehler zu melden. Mit dem `declare`-Ausdruck wird das Symbol angelegt aber nicht gebunden. Das genügt dem Compiler an Information, um seine Arbeit fortsetzen zu können.

```
(declare is-odd?)
(defn is-even? [n]
  (if (zero? n) true (if (pos? n) (is-odd? (dec n)) (is-odd? (inc n)))))
(defn is-odd? [n]
  (if (zero? n) false (if (pos? n) (is-even? (dec n)) (is-even? (inc n)))))
```

Sie werden es wahrscheinlich korrekt vermuten: Das Prädikat `zero?` vergleicht eine Zahl auf Null; `pos?`, ob eine Zahl positiv, d.h. größer Null ist. Die Funktion `dec` dekrementiert eine Zahl um `1`, die Funktion `inc` inkrementiert eine Zahl um `1`. Dekrementieren heißt um Eins reduzieren (subtrahiere Eins), inkrementieren um Eins erhöhen (addiere Eins). Sollte Ihnen unklar sein, was z.B. `inc` und `dec` machen, probieren Sie z.B. `(inc 7)` auf der Konsole aus und lesen Sie die Dokumentation mittels `(doc inc)`.

Die Prädikate `is-even?` und `is-odd?` sind fast wie Klartext zu lesen: Eine Zahl `n` ist gerade (`is-even?`), wenn sie `zero?` ist. Ist sie nicht Null, dann befragt `is-even?` die Funktion `is-odd?`, ob `(dec n)` ungerade ist, sofern `n` positiv ist; ansonsten wird gefragt, ob `(inc n)` ungerade ist. Das Prädikat `is-odd?` ist entsprechend definiert.

[3] Clojure stellt diese Funktionen unter den Namen `even?` und `odd?` bereit. Unsere Implementierung verwendet diese leicht abgeänderten Bezeichnungen, um Namenskonflikte zu vermeiden.

Es ist unverkennbar, wie sich die beiden Funktion auf den Abbruchfall, die Null, hintreiben. Ebenfalls ist offensichtlich, dass die jeweiligen Aufrufe von `is-odd?` bzw. `is-even?` endrekursiv sind, da sie zum Aufrufzeitpunkt in keinem anderen Ausdruck eingebettet sind. Machen Sie sich das im Zweifel anhand einer schrittweisen Evaluation klar. Nur können wir Clojure auf die Endrekursion nicht per `recur` hinweisen. Und ohne diesen Hinweis läuft uns jedoch unnötiger Weise der Callstack über; wählen Sie eine genügend große Zahl, um den Überlauf zu provozieren.

```
user=> (is-even? 10000)
StackOverflowError ...
```

Was also tun?

3.8 Trampoline

Es gibt eine sehr einfache Technik, wechselseitige Endrekursion auch endrekursiv abzuhandeln. Es ist die Trampolin-Technik, und sie erfordert eine Funktion namens `trampoline`. Mit ihr wird ein Anwachsen des Callstacks verhindert.

Der Einsatz der Trampolin-Technik erfordert eine leichte Anpassung der wechselseitig (end)rekursiven Funktionen. Syntaktisch kommt die Anpassung mit einer geringfügigen Ergänzung aus. Den Aufrufen von `is-odd?` bzw. `is-even?` im Funktionsrumpf wird ein `#`-Zeichen vorangestellt. Das `#`-Zeichen triggert ein Reader-Makro und bettet die Aufrufe in einer argumentlosen Funktion ein.

```
(declare is-odd?)
(defn is-even? [n]
  (if (zero? n) true (if (pos? n) #(is-odd? (dec n)) #(is-odd? (inc n)))))
(defn is-odd? [n]
  (if (zero? n) false (if (pos? n) #(is-even? (dec n)) #(is-even? (inc n))
```

Der Ausdruck `#(is-odd? ...)` ist identisch mit `(fn [] (is-odd? ...))`. Die einbettende Funktion dient zur Verzögerung des Funktionsaufrufs. Mit diesem Trick bekommt `trampoline` die Kontrolle über den Funktionsaufruf. Entsprechendes gilt für `#(is-even? ...)`.

Der Aufruf von `is-even?` bzw. `is-odd?` geschieht nun wie folgt – und `trampoline` kümmert sich um die Realisierung des endrekursiven Ablaufs.

```
user=> (trampoline is-even? 10000)
true
user=> (trampoline is-even? 10001)
false
```

Es ist nur noch eine Sache der Zeit, wie lange auf eine Antwort gewartet werden muss. Der Callstack „funkt“ nicht mehr dazwischen.

▷ **Aufgabe 3.7** Erläutern Sie durch Begutachtung von `(source trampoline)` die Arbeitsweise der `trampoline`-Funktion.

3.9 Funktionen höherer Ordnung und Funktionskomposition

Sie haben nun schon an mehreren Stellen Funktionen benutzt, die Funktionen als Argumente erwarten und/oder Funktionen als Rückgabewerte haben. Solche Funktionen heißen „Funktionen höherer Ordnung“ oder

kurz HOF für *Higher-Order Functions*. Die Funktionen `trampoline` oder `apply` sind Beispiele dafür.

Eine HOF, die eine Funktion zurückgibt, ist ein Funktionskonstruktur (*function constructor*). Das Wort „Konstruktor“ können Sie lesen als „Erzeuger“. Man spricht auch von einem Generator (*function generator*), ebenfalls im Sinne der Bedeutung von „Erzeuger“.

Ein Beispiel ist die folgende Funktion `inc-by`, die eine Inkrement-Funktion generiert, die mit einem festen Inkrement-Wert `n` konfiguriert wird.

```
user=> (defn inc-by [n] (fn [x] (+ x n)))
#'user/inc-by
user=> ((inc-by 5) 6)
11
```

Eine Funktion, die einer HOF als Argument übergeben wird, dient in der Regel dazu, den Rechenprozess der HOF zu konfigurieren.

```
user=> (defn operate [f] (fn [x y] (f x y)))
#'user/operate
user=> ((operate *) 2 3)
6
```

Eine besondere HOF ist `comp`, die beliebig viele Funktionen entgegen nimmt und eine Funktion zurückgibt, die die Komposition (*composition*) der Funktionen realisiert. Wollten wir die Funktion selber definieren – unsere Umsetzung nennen wir `compose` – so müsste sie für zwei Funktionen wie folgt definiert sein.

```
user=> (defn compose [f g] (fn [& args] (f (apply g args))))
#'user/compose
user=> ((compose #(* % %) dec) 4)
9
user=> ((compose dec #(* % %)) 4)
15
```

▷ **Aufgabe 3.8** Definieren Sie `compose` als variadische Funktion, die kein Argument, eine Funktion, zwei oder beliebig viele Funktionen als Argument zulässt. Ohne Argument soll `compose` die Funktion `identity` (eine Clojure-Funktion) zurückliefern, bei nur einem Argument die Funktion selber.

Die Umsetzung von `comp` ist ähnlich zur Definition von `compose`, sie ist nur optimiert auf das Laufzeitverhalten von Clojure. Sie sollten daher zur Funktionskomposition `comp` verwenden.

Die Funktionskomposition hat einen „optischen“ Nachteil: Die Funktionen werden von ihrer Reihenfolge her von hinten nach vorne angewendet:

```
user=> ((comp dec #(* % %) inc) 4)
24
```

Für die gewohnte Lesart von links nach rechts bietet sich das „Thread“-Makro `->`. Es evaluiert Formen von links nach rechts und setzt den durchgereichten Wert jeweils als fehlendes Argument in der Form ein. Aus diesem Grund müssen die Funktionen lediglich „geklammert“ werden.

```
user=> (-> 4 (inc) (#(* % %)) (dec))
24
```


3.10 Lösungen

Aufgabe 3.1 Der Programmcode ist praktisch direkt aus der Definition ableitbar:

```
(defn fib [n]
  (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))
```

Aufgabe 3.2 Bei der endrekursiven Lösung müssen Sie die Berechnung neu strukturieren. Sie benötigen einen „Counter“, der das Argument `n` „herunterzählt“, einen Merker für das aktuelle und einen Merker für das vorhergehende Ergebnis. Mit jeder Rekursion summiert sich das Ergebnis auf.

```
(defn fib
  ([n] (fib n 0 1))
  ([n res prev]
    (if (== n 0) res (recur (- n 1) (+ res prev) res))))
```

Beachten Sie, dass der Vergleich auf Gleichheit bei Zahlen mit der Funktion `==` vorgenommen werden soll (da effizienter) und nicht mit `=` (generelle Wertegleichheit). Für den Vergleich mit Null ist auch `zero?` verwendbar.

Aufgabe 3.3 Prinzipiell ja. Im generellen Fall ist eine alternative endrekursive Lösung nicht so einfach umzusetzen, wie bei der Fakultätsfunktion. Der „Akkumulator“ fungiert als eine Art Speicher des aktuellen Rechenfortschritts. Vielfach kann man den Rechenfortschritt ebenfalls nur durch eine anwachsende Datenstruktur angemessen protokollieren. Man verlegt das Anwachsen des Callstacks in eine anwachsende Datenstruktur – und gewinnt nichts. Man baut sich den Callstack sozusagen nach, was das Programm in der Regel nicht klarer, sondern komplizierter macht.

Die normale Rekursion ist das Mittel der Wahl, wenn es um sich wiederholende Vorgänge geht. Endrekursion wird man überlicherweise nur dann wählen und anstreben, wenn sie unkompliziert und nicht um den Preis eines „simulierten“ Callstacks per Datenstruktur zu bekommen ist.

Aufgabe 3.4 Zum Beispiel verrät Ihnen eine Suche in Wikipedia die folgende Näherungsformel für $n!$.

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Mit e ist die Euler'sche Zahl gemeint, ihr Wert liegt angenähert bei 2.718281828459045. In Clojure ist der Zugriff auf e möglich durch `Math/E`, ebenso wie π mit `Math/PI` abrufbar ist. Über das `Math`-Interface der Java-Umgebung sind auch die Wurzelfunktion (`Math/sqrt`) und die „Hochfunktion“ (`Math/pow`) nutzbar.

```
(defn fact_approx [n]
  (*
    (Math/sqrt (* 2 Math/PI n))
    (Math/pow (/ n Math/E) n)))
```

Schon für sehr kleine Werte von `n` ist die Näherungsformel recht gut:

```
user=> (fact_approx 5)
118.0191679575901
user=> (fact_approx 6)
710.078184642185
```

An diesem einfachen Beispiel sehen Sie, wie hilfreich die Mathematik sein kann, um Rechenzeiten zu sparen. Die Wahl zwischen der Formel und der rekursiven Funktionsdefinition wirft ein typisches Ingenieursproblem auf: Genaugkeit gegen Geschwindigkeit. Es ist im Einzelfall zu entscheiden, welchem Kriterium der Vorrang zu geben ist.

In diesem speziellen Fall ist der Näherungsformel eine Grenze gesetzt, da sie mit Fließkommazahlen endlicher Genauigkeit rechnet. Für Fakultäten jenseits `170` ist Schluß. Das Ergebnis wird so groß, dass es für unendlich (*infinite*) gehalten wird.

```
user=> (fact_approx 170)
7.25385893454291E306
user=> (fact_approx 171)
Infinity
```

Aufgabe 3.5 Um Konflikte in den Namen für die Argumente zu vermeiden, wählen wir `n'` als Argument für die Funktion `fact` und `n` in `loop`.

```
(defn fact [n']
  (loop [n n' acc 1]
    (if (<= n 1) acc (recur (- n 1) (* n acc)))))
```

Dank `loop` muss `fact` nicht mehr als *variadic function* ausgelegt werden.

Aufgabe 3.6 Die Lösung lautet:

```
(defn fib [n']
  (loop [n n' res 0 prev 1]
    (if (== n 0) res (recur (- n 1) (+ res prev) res))))
```

Aufgabe 3.7 Die Funktion `trampoline` ist variadisch, sie erwartet eine Funktion und optional weitere Argumente, mit denen die Funktion aufgerufen werden soll. Im Fall weiterer Argumente packt `trampoline` die Funktionsanwendung `(apply f args)` in eine Funktion. Den „Trick“ mit dem Reader-Makro `#` haben Sie bereits kennengelernt. Damit wird die Funktionsanwendung für eine spätere Ausführung verzögert.

```
  ([f & args]
     (trampoline #(apply f args))))
```

Die Funktionsdefinition im Falle eines einzelnen Argumentes sieht wie folgt aus:

```
  ([f]
     (let [ret (f)]
       (if (fn? ret)
         (recur ret)
         ret)))
```

Die Funktion `f` wird per `(f)` ausgeführt und das Ergebnis an `ret` gebunden. Wenn das Ergebnis eine Funktion ist (`(fn? ret)`), dann wird die Funktion per `recur` aufgerufen – damit bleibt der Callstack unverändert. Ist `ret` keine Funktion, wird der Ergebniswert `ret` zurück gegeben.

In unserem Anwendungsbeispiel liefert `is-even?` die Funktion `#(is_odd? ...)` zurück und umgekehrt, die jeweils den nächsten Rechenschritt kapseln, den `trampoline` nun ohne Ansprüche an den Callstack ausführen kann.

Aufgabe 3.8 Die ersten drei Argumentfälle sind praktisch wörtlich umzusetzen. Einzig der Fall mit mehr als zwei Funktionen stellt eine kleine Herausforderung dar.


```
(defn compose
  ([] identity)
  ([f] f)
  ([f g] (fn [& args] (f (apply g args))))
  ([f g & h] (compose f (apply compose g h))))
```

Historie

22. Mai 2012 Schreibkorrektur; mit Dank an Katharina Tetkowski und Manuel Wöhr

17. Apr. 2012 Überarbeitung zur wechselseitigen Rekursion, um Namenskonflikte mit den vorhandenen Clojure-Funktionen `odd?` und `even?` zu vermeiden; Beginn eines Kapitels zur HOFs

10. Apr. 2012 Neu: wechselseitige Rekursion und Trampoline; auch sonst teils erhebliche Überarbeitungen; neue Platzierung der Aufgaben; eine neue Aufgabe

5. Apr. 2012 Leichte Überarbeitungen und Korrekturen;

1. Apr. 2012 Neues Unterkapitel zur Substitutionstechnik; fehlerhaften Labelbezug behoben; Überarbeitung des Kapitels zur Limitierung des Callstacks

27. Okt. 2011 Ein doppeltes Wort entfernt. Dank an Felix Birke.

14. Okt. 2011 Kleinere Korrekturen, Anpassung an Clojure 1.3.0, Übungen mit Lösungen hinzugefügt. Mit Dank an Marc Hesenius.

29. Apr. 2011 Kleinere Korrekturen und eine ergänzende Übung.

28. Apr. 2011 Erste Version.

4 Fallstudie: Bubblesort

4.1 Der Bubblesort-Algorithmus

Der Bubblesort ist ein naiver Algorithmus, um Daten zu sortieren. Er ist nicht sehr effizient, jedoch anschaulich zu verstehen und ein beliebtes, einführendes Beispiel zum Themengebiet der „Algorithmen & Datenstrukturen“.

Angenommen, es liegt die Zahlenfolge `[4 3 1 5 7 2 6]` vor. Der Bubblesort-Algorithmus läuft von links nach rechts und betrachtet, sofern genügend Elemente gegeben sind, immer zwei Elemente. Beginnend mit `4` und `3` entscheidet der Vergleich „Welches Element ist größer?“, ob die beiden Zahlen getauscht werden oder an ihren Plätzen verbleiben. Da `4` größer als `3` ist, findet ein Tausch statt. Die Zahlenfolge wird zu `[3 4 1 5 7 2 6]`. Der Vergleich und mögliche Tausch wird fortgesetzt mit den Elementen an der zweiten und dritten Position von links, `4` und `1`. Wieder findet ein Tausch statt: `[3 1 4 5 7 2 6]`. Diese Vergleiche werden bis zu den letzten beiden Elementen fortgesetzt. Nach einem vollständigen Durchgang ergibt sich `[3 1 4 5 2 6 7]`. Der Bubblesort ist so aufgebaut, dass sich das größte Element, die größte Zahl, nach einem Durchlauf ganz rechts befindet.

Diese Durchläufe und Vergleiche werden so oft durchgeführt, bis kein Tausch der Elemente mehr stattfindet.

Der Name „Bubblesort“ kommt von der an das Verfahren angelehnten Vorstellung, dass das größere von zwei Elementen wie die größere von zwei Blasen (*bubbles*) in einer Wassersäule aufsteigt.

Selbstredend ist, dass eine leere Folge von Elementen oder eine Folge mit nur einem Element als sortiert gilt.

4.2 Eine funktionale Betrachtungsweise

In imperativen Programmiersprachen wird der Bubblesort typischerweise mit einem Array realisiert und einem Index als Zeiger auf die aktuell betrachtete Position im Array. Im Falle eines Tauschs werden die beiden benachbarten Elemente im Array in ihren Positionen getauscht. Die deutsche Wikipediaseite erklärt den Bubblesort genau so.[1]

In funktionalen Sprachen ist dieser Implementierungsansatz nicht umsetzbar. Clojure arbeitet mit persistenten, immutablen Datenstrukturen. Eine immutable Datenstruktur ist – das Wort „immutabel“ verrät es – nicht änderbar. Sie können keine Elemente innerhalb einer Sequenz tauschen, es ist schlicht nicht möglich. Es bleibt nur die Wahl, eine neue Sequenz zu erstellen. In einer funktionalen Welt geht es stets um die Zerlegung (Destruktion, *destruction*) und Erstellung (Konstruktion, *construction*) von Daten, niemals jedoch um ihre Modifikation, ihre Veränderung. Man muss sich in diese Denkweise einfinden.

[1] http://de.wikipedia.org/wiki/Bubblesort (2011-11-07)

Eine Folge (*sequence*) von Daten kann z.B. mit den folgenden Funktionen zerlegt oder auseinander genommen werden. Es sind Funktionen, die für die Erstellung des Bubblesort-Programms von Nutzen sind. Ein `first` gibt das erste Elemente einer Folge zurück, `second` das zweite Element und `rest` eine neue(!) Folge ohne das erste Element der Ausgangsfolge; `rest` konstruiert eine neue Folge, es ist eine sogenannte Konstruktor-Funktion (*constructor function*).

```
user=> (first [4 3 1 5 7 2 6])
4
user=> (second [4 3 1 5 7 2 6])
3
user=> (rest [4 3 1 5 7 2 6])
(3 1 5 7 2 6)
```

Beachten Sie, dass `rest` eine Sequenz zurück liefert; Clojure arbeitet „unter der Haube“ nach wie vor mit einem Vektor. Mit `type` können Sie das herausfinden.

```
user=> (type (rest [4 3 1 5 7 2 6]))
clojure.lang.PersistentVector$ChunkedSeq
```

Mit der doppelten Anwendung von `rest` erstellen sie eine neue Folge ohne das erste und zweite Element.

```
user=> (rest (rest [4 3 1 5 7 2 6]))
(1 5 7 2 6)
```

Ob das erste und zweite Element getauscht werden müssen oder nicht, `cons` hilft bei der Konstruktion (`cons` heißt nicht zufällig so!) einer neuen Sequenz. Im Beispiel auf der Konsole tauschen wir die ersten beiden Elemente gemäß Bubblesort-Verfahren.

```
user=> (cons 3 (cons 4 [1 5 7 2 6]))
(3 4 1 5 7 2 6)
```

Eine weitere Hürde für Umsteiger in die funktionale Programmierung ist, dass es Wiederholungen in Form von Schleifen (for, while etc.) nicht gibt. Das „Zauberwort“ heißt Rekursion! Die Rekursion, der Selbstaufruf von Funktionen, ist der einzige Weg in funktionalen Sprachen, um Code wiederholt auszuführen. Das ist zunächst eine Umstellung, fühlt sich nach kurzer Zeit aber vollkommen normal an.

Nach dem Tausch oder auch Nicht-Tausch der ersten beiden Elemente ist das neue oder alte „erste“ Element abgehandelt und der gleiche Ablauf des Bubblesort-Durchgangs setzt sich nun fort, er wiederholt sich sozusagen mit dem neuen bzw. alten „zweiten“ Element ge-`cons`-t mit den übrig gebliebenen Elementen.

4.3 Ein Bubble-Durchgang in Clojure

Der nachstehende Code ist eine Umsetzung *eines* Durchgangs des Bubblesort-Algorithmus'. Die Funktion heißt `bubble`. Speichern Sie diesen Code in einer Datei z.B. mit Namen `bs.clj` ab. Über die REPL können Sie in einer Konsole mit `(load "bs")` den Code laden und anschließend interaktiv ausführen.

```
(defn bubble
  [itms]
```

```
(if (or (empty? itms) (empty? (rest itms)))
  itms
  (let [fst (first itms)
        snd (second itms)
        rst (rest (rest itms))]
    (if (> fst snd)
      (cons snd (bubble (cons fst rst)))
      (cons fst (bubble (cons snd rst))))))))
```

Die Funktion `bubble` testet zuerst die Bedingung für den Abbruch. Ist die Sequenz `itms` leer oder (`or`) hat sie nur ein Element, dann gibt `bubble` die leere bzw. einelementige Sequenz zurück; sie ist identisch mit `itms`. Trifft die Abbruchbedingung nicht zu, werden mittels `let` Bindungen erzeugt. Die Bindungen an `fst`, `snd` und `rst` entsprechen unserer obigen Zerlegung mittels `first`, `second` und der zweifachen Anwendung von `rest`. Das funktioniert, da wir wissen, dass die Liste von `itms` aus mindestens zwei Elementen bestehen muss. Die Bindungen helfen, den Code in der zweiten `if`-Form von Code-Doppelungen zu befreien und leichter verständlich zu machen.

Die zweite `if`-Form konstruiert, je nach Ausgang des Vergleichs von `fst` und `snd`, eine neue Sequenz per `cons`. Nach dem ersten Element (entweder `snd` oder `fst`) folgt ein rekursiver Aufruf von `bubble`. Dieser Aufruf setzt `bubble` fort mit einer Sequenz, die aus `fst` bzw. `snd` und den Elementen aus `rst` besteht.

Lassen Sie uns die Funktion `bubble` austesten.

```
user=> (bubble [])
[]
user=> (bubble [4])
[4]
user=> (bubble [4 3 1 5 7 2 6])
(3 1 4 5 2 6 7)
```

Der wiederholte Aufruf von `bubble` mit dem jeweiligen Ergebnis des vorigen Durchgangs bewirkt eine Sortierung.

```
user=> (bubble [4 3 1 5 7 2 6])
(3 1 4 5 2 6 7)
user=> (bubble [3 1 4 5 2 6 7])
(1 3 4 2 5 6 7)
user=> (bubble [1 3 4 2 5 6 7])
(1 3 2 4 5 6 7)
user=> (bubble [1 3 2 4 5 6 7])
(1 2 3 4 5 6 7)
user=> (bubble [1 2 3 4 5 6 7])
(1 2 3 4 5 6 7)
```

Nach fünf Anwendungen von `bubble` ist die Ausgangsfolge von Zahlen sortiert. Warum fünf? Als Menschen sehen Sie die Sortierung bereits nach vier Durchgängen; insgeheim überprüft Ihr Gehirn die Zahlenfolge. Clojure kann das nicht so ohne weiteres. Wir benötigen ein Abbruchkriterium für den rekursiven Aufruf von `bubble`. Das Kriterium ist einfach: Ist die Eingangsfolge, die an `bubble` übergeben wird, gleich der Ergebnisfolge, dann ist die Folge sortiert.

4.4 Bubblesort und das Fixpunktverfahren

Fassen wir das Verfahren noch einmal zusammen: Eine Funktion, hier `bubble`, wird solange mit seinem eigenen Ergebnis aufgerufen, bis das Aufrufergebnis gleich dem Aufrufargument ist. Man nennt dieses Abbruchkriterium den Fixpunkt der Funktion. Mathematisch ausgedrückt ist x^* ein Fixpunkt einer Funktion f, wenn gilt $x^* = f(x^*)$. Das Verfahren, sich schrittweise mit einem Startwert x_0 einem Fixpunkt anzunähern, kann mathematisch gefasst werden über $x_{k+1} = f(x_k)$ für $k \in \mathbb{N}_0$.

Wir können in Clojure eine Funktion `fix` erstellen, die für eine beliebige Funktion `f`, die ein Argument entgegen nimmt, das Fixpunktverfahren rekursiv aufsetzt und den Fixpunkt als Abbruchkriterium hat.

```
(defn fix [f]
  (fn [x]
    (if (= (f x) x)
      x
      (recur (f x)))))
```

Die Funktion `fix` ist eine direkte Umsetzung des beschriebenen Fixpunktverfahrens. Sie gibt eine anonyme Funktion zurück, deren Rumpf das Fixpunktverfahren realisiert und für die Funktion `f` konfiguriert. Die Funktion `fix` ist ein Beispiel für eine Funktion höherer Ordnung (*higher-order function*). Sie nimmt eine Funktion als Argument entgegen und gibt eine neue Funktion als Ergebnis zurück.

Das Fixpunktverfahren begegnet einem in der funktionalen Programmierung immer wieder. Sie tun gut daran, die Funktion `fix` in Ihr Repertoire an funktionalen Programmiertechniken aufzunehmen.

Der Einsatz von `fix` zusammen mit `bubble` ist denkbar einfach. Die Beispielfolge kann jetzt sortiert werden mit:

```
user=> ((fix bubble) [4 3 1 5 7 2 6])
(1 2 3 4 5 6 7)
```

Definieren wir die Funktion `bsort`, die den Bubblesort-Algorithmus vollständig umsetzt.

```
(defn bsort [itms] ((fix bubble) itms))
```

Unterziehen wir `bsort` einigen Tests.

```
user=> (bsort [4 3 1 5 7 2 6])
(1 2 3 4 5 6 7)
user=> (bsort [1 2 3 4 5 6 7])
[1 2 3 4 5 6 7]
user=> (bsort [2 3 2])
(2 2 3)
user=> (bsort [2 2 2])
[2 2 2]
user=> (bsort [2])
[2]
user=> (bsort [])
[]
```

4.5 Zur Performanz funktionaler Programme

Umsteiger von imperativen auf funktionale Sprachen zeigen sich oft besorgt um die Performanz ihrer funktionalen Programme. Die Funktionen mögen zwar sehr schön und glasklar den Bubblesort-Algorithmus beschreiben, doch der Argwohn nagt: Ist ein imperatives Programm mit direkten Manipulationen der Positionen beim Tausch in einer Zahlenfolge nicht viel effizienter? Möglicherweise. Der Preis dafür ist jedoch nicht selten relativ hoch: Die Optimierung auf maximal zeit- oder speichereffiziente Ausführung verstümmelt Algorithmen und Datenstrukturen oft bis zur Unkenntlichkeit. Das hat Folgen, was die Lesbarkeit, Wartbarkeit, Skalierbarkeit, Konfigurierbarkeit betrifft.

Die moderne Softwaretechnik hält diese und weitere Aspekte für mindestens ebenso wichtig wie die Performanz, wenn nicht gar für Software-Entwicklungsprozesse als entscheidend. Nicht ohne Grund verbreiten sich funktionale Sprachen und funktionale Sprachkonzepte immer mehr. Funktionale Programme sind weitaus besser lesbar, sie sind oft direkt eingängige und verständliche Beschreibungen der Vorgänge in einem Programm. Sie haben häufig gar die Qualität einer ausführbaren Spezifikation, sie sind gut generalisierbar und anpassbar und können von einem Compiler oft sehr gut optimiert werden. Funktionale Programme können bei all diesen Eigenschaft sehr performant ausgeführt werden.

Wesentlich entscheidender als das „Frisieren“ eines Algorithmus ist die Wahl geeigneter Algorithmen und Datenstrukturen. Den Bubblesort-Code sollte man nicht aus Überlegungen an vermutete Compileroptimierungen verändern. Erstens liegen Programmierer(innen) meist vollkommen daneben, was ihre Mutmaßungen anbelangt, welche Maßnahmen ein Programm schneller oder speichereffizienter machen. Zweitens: Programme sind für den Menschen geschrieben. Klarheit und Lesbarkeit sind immer vorzuziehen – Ausnahmen bestätigen die Regel! Wenn, dann sollte man den Bubblesort z.B. durch den Quicksort austauschen, statt einen leistungsschwachen Sortieralgorithmus zu „optimieren“.

4.6 Eine alternative Umsetzung mit *destructuring bind*

In Clojure kann man die Argumente zu einer Funktion oder zu einem `let` positional zerlegen und binden lassen. Man nennt das zerlegende Bindung (*destructuring bind*). Mit diesem Feature und mit `cond` kann man die `bubble`-Funktion, hier `bubble2` bezeichnet, noch etwas übersichtlicher hinschreiben. Es ist eine Konvention unter Clojure-Programmierer(inne)n, in einer `cond`-Form für den else-Fall nicht `true` zu verwenden, was absolut korrekt wäre, sondern das Keyword `:else`. Das Keyword macht den Code lesbarer.

```
(defn bubble2
  [[fst snd & rst]]
  (cond
    (not fst) []
    (not snd) [fst]
    (> fst snd) (cons snd (bubble2 (cons fst (or rst []))))
    :else       (cons fst (bubble2 (cons snd (or rst []))))
))
```

Die zusätzlichen eckigen Klammern im Argument der Funktion `bubble2` zeigen Clojure eine zerlegende Bindungsform an. Das erste Element einer übergebenen Sequenz wird an `fst` gebunden, das zweite an `snd`. Mit dem

einleitenden `&`-Zeichen werden alle übrigen Elemente an `rst` gebunden. Ein Beispiel mit `let` lässt leicht verstehen, wie die zerlegende Bindung funktioniert:

```
user=> (let [[fst snd & rst] [1 2 3 4]] (list fst snd rst))
(1 2 (3 4))
user=> (let [[fst snd & rst] [1 2]] (list fst snd rst))
(1 2 nil)
user=> (let [[fst snd & rst] [1]] (list fst snd rst))
(1 nil nil)
user=> (let [[fst snd & rst] []] (list fst snd rst))
(nil nil nil)
```

Der Zerlegungsmechanismus arbeitet im Hintergrund mit den Funktionen `nth` und `nthnext`. Von daher wird jede Datenstruktur bei der zerlegenden Bindung als Argument akzeptiert, auf die diese beiden Funktionen angewendet werden können. Dabei ist unbedingt zu beachten, dass die zerlegende Bindung selbst dann funktioniert, wenn zu wenig Elemente als Argumente übergeben werden. Das kann zu ernsthaften Problemen führen und ungewollte Fehler im Clojure-Code provozieren.

Was passiert, wenn die an `bubble2` übergebene Sequenz nur zwei Elemente hat? Dann wird `rst` an `nil` gebunden; das zeigt auch das `let`-Beispiel. Das ist auch der Grund für den „Trick“ mit dem `or`. Wenn `rst` `nil` ist, dann liefert der `or`-Ausdruck die leere Liste zurück. Ist `rst` nicht `nil`, dann ist hier `rst` eine Sequenz und `or` gibt die `rst`-Sequenz zurück. Auf diese Weise erhalten wir immer eine Sequenz als zweites Argument zu `cons`.

Und was passiert, wenn die Sequenz nur ein oder kein Element hat? Bei nur einem Element wird `snd` an `nil`, bei keinem Element außerdem auch `fst` und `nil` gebunden.

Erkennen Sie die Problematik? Was, wenn die Sequenz `nil` als Elemente hat? Das ist nicht unterscheidbar von dem Fall, wenn sie keine Elemente hat. Verdichten wir dies auf ein Beispiel an der REPL.

```
user=> ((fn [[fst]] fst) [])
nil
user=> ((fn [[fst]] fst) [nil])
nil
```

Die leere Collection `[]` als Argument kann nicht von der Collection `[nil]` aus der Sicht von `fst` unterschieden werden. Das heißt im Klartext: Sie dürfen kein `nil` als Element in einer Sequenz verwenden, ansonsten bricht der Code für die Funktion `bubble2`. Damit erhält `nil` als Wert eine Sonderstellung, die ihm eigentlich nicht zugedacht ist. Sequenzen sollten in ihren Datenwerten in keinster Weise eingeschränkt werden.

Das Problem ist lösbar, wenn neben der zerlegenden Bindung Zugriff besteht auf die gesamte übergebene Sequenz. Dafür kann `:as` in der *destructing bind*-Form verwendet werden. Die Funktion `bubble3` vermeidet die Sonderstellung von `nil`.

```
(defn bubble3
  [[fst snd & rst :as itms]]
  (cond
    (empty? itms) []
    (empty? (rest itms)) itms
```

```
    (> fst snd) (cons snd (bubble3 (cons fst (or rst []))))
    :else       (cons fst (bubble3 (cons snd (or rst []))))
))
```

Die Lektion, die Sie daraus ziehen sollten, ist: Achten Sie insbesondere beim *destructuring bind* in Clojure darauf, dass `nil` keine Sonderstellung bekommt. Eventuell sollten Sie das in Testfällen überprüfen!

Etwas unerfreulich an dem Code für `bubble3` ist der „`or`-Trick“. Er kompensiert, dass Clojure bei `& rst` nicht grundsätzlich eine Sequenz an `rst` bindet. Das fordert einer Leserin bzw. einem Leser des Codes ein unnötiges Verständnis für das `or`-Konstrukt ab.

Für die Kombination von *destructuring bind* und `cond`, das die Anzahl der Argumente prüft, gibt es in vielen anderen funktionalen Sprachen ein so genanntes *pattern matching*. Mit *pattern matching* sind syntaktisch kurze und klare Fallunterscheidungen bei den Argumenten möglich. In Clojure würde sich *pattern matching* syntaktisch sehr gut mit *variadic functions* vertragen. Wer weiß, vielleicht wird eine künftige Clojure-Version das nachrüsten oder ein Makro *pattern matching* umsetzen.

4.7 Aufgaben

▷ **Aufgabe 4.1** Eine Frage für Fortgeschrittene: Warum macht es beim Bubblesort-Algorithmus wenig Sinn, sich über den Gebrauch von *lazy sequences* Gedanken zu machen?

▷ **Aufgabe 4.2** Schreiben Sie eine Funktion `sum`, die eine Sequenz von Zahlen aufsummiert. Verwenden Sie *destructuring bind*.

▷ **Aufgabe 4.3** Schreiben Sie die Fakultätsfunktion so um, dass die Rekursion mit Hilfe der Fixpunktfunktion `fix` realisiert wird.

4.8 Lösungen

Aufgabe 4.1 Der Bubblesort muss mit jedem Durchlauf die gesamte Sequenz durchgehen. Verzögerte Sequenzen machen eher dann Sinn, wenn nicht alle Elemente einer Sequenz realisiert werden müssen.

Aufgabe 4.2 Da eine Sequenz von Zahlen nicht das Element `nil` enthält, braucht die zerlegende Bindungsform darauf keine Rücksicht nehmen.

```
(defn sum [[fst & rst]]
  (if (not fst) 0 (+ fst (sum rst))))
```

Eine Lösung, die nicht auf die zerlegende Bindung zurückgreift, ist:

```
(def sum (partial reduce +))
```

Aufgabe 4.3 Der Schlüssel zur Lösung liegt darin, die Ähnlichkeit zur endrekursiven Fassung der Fakultätsfunktion zu entdecken. Anders lässt sich die Fakultätsfunktion keinem Fixpunkt zutreiben.

```
(defn factfix [[n acc]]
  (if (<= n 1) [1 acc] [(dec n) (* n acc)]))

(defn fact [n] (second ((fix factfix) [n 1])))
```

Historie

25. Mai 2012	Schreibkorrektur; mit Dank an Christian Hecker
22. Mai 2012	Schreibkorrektur; mit Dank an Carsten Fasold
22. Mai 2012	Lösungskapitel erscheint im Inhaltsverzeichnis
11. Nov. 2011	Aufgabe 4.3 hinzugefügt; Ergänzung zu Aufgabe 4.2
7. Nov. 2011	Ergänzende Erläuterungen zum *destructuring bind*. Übungsaufgaben überarbeitet.
7. Nov. 2011	Kleinere Fehlerkorrekturen. Mit Dank an Marc Hesenius.
5. Mai 2011	Fehlerhafterweise zweimal „Quicksort“ statt „Bubblesort“ geschrieben; Rechtschreibfehler beseitigt (Dank an Peter Schiffmann)
30. Apr. 2011	Erste Version.

5 Fallstudie: Quicksort

5.1 Das Sortierverfahren allgemein beschrieben

Es gibt verschiedene Algorithmen, um Daten gemäß einem Kriterium zu sortieren. Eines der bekanntesten und sehr effizienten Verfahren ist der Quicksort-Algorithmus.

Der Algorithmus arbeitet wie folgt: Gegeben sei eine Folge von Daten. Der Anschaulichkeit halber nehmen wir eine Folge von Zahlen, z.B. den Clojure-Vektor `[4 3 1 5 7 2 6]`. Aus dieser Folge von Daten wählen wir ein beliebiges Element aus; wir nennen es Pivot-Element. Da jedes Element so gut wie jedes andere gewählt werden kann, nehmen wir stets das erste Element aus der Folge als Pivot-Element; im Beispiel ist das die Zahl `4`.

Nun bilden wir zwei „Pakete“. Das eine „Paket“ ist eine Folge von Daten mit ausschließlich den Elementen, die kleiner als das Pivot-Element sind. In unserem Beispiel ist das die Folge mit den Zahlen `[3 1 2]`. Beachten Sie, dass die Zahlen in der Reihenfolge gelistet sind, in der sie auch in der Ausgangsfolge auftreten. Das andere „Paket“ besteht aus der Folge der „übrig“ gebliebenen Daten, also den Zahlen im Beispiel, die größer oder gleich dem Pivot-Element sind, `[5 7 6]`.

Auf den beiden Paketen wird wieder ein Quicksort angewendet. Der Algorithmus ist rekursiv. Sprich, aus dem Paket mit den kleineren Elementen `[3 1 2]` wird die sortierte Folge `[1 2 3]` und aus dem Paket mit den größeren bzw. gleichen Elementen `[5 7 6]` wird `[5 6 7]`. Wenn Sie dem Sortierergebnis mit den kleineren Elementen erst das Pivot-Element hinzufügen und dann die Elemente aus dem Sortierergebnis mit den größeren bzw. gleichen Elementen ergänzen, ergibt sich das sortierte Endergebnis `[1 2 3 4 5 6 7]`. Für den rekursiven Abstieg müssen Sie lediglich noch wissen, dass der Quicksort auf eine leere Datenfolge angewendet eine leere Datenfolge ergibt.

Die Herausforderung ist, den Quicksort-Algorithmus in Clojure umzusetzen. Wir werden dabei schrittweise vorgehen, so, wie eine Programmiererin bzw. ein Programmierer sich Stück für Stück das Programm erschließt. Dieses Vorgehen ist wichtig nachzuvollziehen. In Clojure entwickelt man Programme immer wieder im Rückgriff auf die REPL in einer Konsole. Man interagiert mit seinem Programm. Das ist ganz anders, als Sie das vielleicht von einer Sprache wie Java kennen. In Java arbeiten Sie dateizentriert, eine direkte Interaktion mit Ihrem Programm ist nicht möglich, bestenfalls bleibt Ihnen ein Java-Debugger.

Sie werden in Kürze sehen, wie das Clojure-Programm eine perfekte Beschreibung des Algorithmus' ist. Die obige Erklärung dagegen wirkt fast sperrig und umständlich. Es ist eine bemerkenswerte Eigenschaft funktionaler Programme, dass sie sich – wenn sie gut geschrieben sind – wie eine Spezifikation lesen lassen.

5.2 Die Bausteine des Algorithmus'

Wie kommen wir an das Pivot-Element einer Datenfolge? Bleiben wir bei dem Beispiel mit dem Vektor `[4 3 1 5 7 2 6]`. Mit `first` erhält man das erste Element einer Sequenz, mit `rest` den Rest. Probieren Sie es selbst an der Konsole aus:

```
user=> (first [4 3 1 5 7 2 6])
4
user=> (rest [4 3 1 5 7 2 6])
(3 1 5 7 2 6)
```

Ihnen fällt sicher auf, dass `rest` keinen Vektor zurück gibt, sondern eine Sequenz. Das liegt daran, dass `rest` und einige weitere Funktionen, die wir verwenden werden, von den konkreten zusammengesetzten Datentypen der Liste, dem Vektor, der Map und dem Set abstrahieren und darüber ein einheitliches logisches Interface für Collections (Datenansammlungen) legen. Man spricht dann auch von einer Sequenz in Clojure.

Es gibt zahlreiche Funktionen für die Arbeit mit Sequenzen. In der Regel ist für viele der grundlegenden Operationen schon etwas vorhanden. Sie müssen die Dokumentation zu Clojure bemühen oder sich eines Buchs bedienen. Es ist eine Funktion gesucht, mit der wir Elemente aus einer Folge herausfiltern können.

Tatsächlich existiert die Funktion `filter`. Die Funktion erwartet zwei Argumente, das verrät der Aufruf `(doc filter)`: ein Prädikat und eine Datenfolge, eine „Kollektion“ (*collection*). Ein Prädikat ist eine Funktion, die als Rückgabe nur einen der Bool'schen Werte `true` oder `false` hat. Hier hat sie eine besondere Aufgabe: Das Prädikat soll einen Wert aus der Collection bekommen und entscheiden, ob das Element ein Kriterium erfüllt oder nicht erfüllt. Die Filterfunktion `filter` kann so für jedes Element der Collection ermitteln, welche Elemente ausgefiltert und in einer neuen Collection zusammengestellt werden.

Schreiben wir uns eine einfache Funktion, die ermittelt, ob eine Zahl kleiner als das Pivot-Element `4` ist, oder nicht. Die Variable `itm` steht für „*item*“.

```
user=> (fn [itm] (< itm 4))
#<user$eval1613$fn__1614 user$eval1613$fn__1614@b4faeb>
```

Probieren wir diese Funktion direkt aus.

```
user=> ((fn [itm] (< itm 4)) 1)
true
user=> ((fn [itm] (< itm 4)) 5)
false
```

Nun kann die anonyme Funktion zum `filter`-Einsatz kommen. Wir übernehmen die Zahlenfolge unseres Beispiels ohne Pivot-Element.

```
user=> (filter (fn [itm] (< itm 4)) [3 1 5 7 2 6])
(3 1 2)
```

Clojure bietet eine zu `filter` „inverse“ Funktion an, die all die Elemente als Sequenz zurückgibt, die das durch das Prädikat ausgedrückte Kriterium genau nicht erfüllen.

```
user=> (remove (fn [itm] (< itm 4)) [3 1 5 7 2 6])
(5 7 6)
```


Bleibt noch eine Frage übrig zu klären: Wie kann aus den Elementen mehrerer Einzelsequenzen eine neue zusammenfassende Sequenz gebildet werden? Dazu gibt es die Funktion `concat`; das Wort „konkatenieren“ gibt es auch im Deutschen, was soviel heißt wie „aneinanderfügen“. Erinnern Sie sich an die freisprachliche Beschreibung des Quicksort-Algorithmus' und konkatenieren wir die Collection `[1 2 3]`, das in eine Collection verpackte Pivot-Element `[4]` und die Collection `[5 6 7]`.

```
user=> (concat [1 2 3] [4] [5 6 7])
(1 2 3 4 5 6 7)
```

5.3 Eine erste Umsetzung

Wir haben damit alle Bausteine zusammen, um daraus den Quicksort als Programm zu schreiben. Legen Sie eine Datei an, z.B. `qs.clj`, und beginnen Sie, Ihr Programm mit einem Editor zu erstellen.

Das Grundgerüst der Funktion `qsort` besteht aus dem obligatorischen Dokumentationsstring – das gehört zum guten Programmierstil und sollten Sie sich angewöhnen. Übrigens programmiert man üblicherweise auf Englisch! Üben Sie auch das ein. Danach folgt die Deklaration des Funktionsarguments. Die Funktion `qsort` erwartet lediglich ein Argument. Beim Funktionsaufruf wird der Aufrufparameter an die Variable `col` gebunden.

```
(defn qsort
  "Quicksort numbers of a given collection"
  [col]
  (if (empty? col)
    ()
    (let [pvt (first col)
          rst (rest col)]
      ; more to come
)))
```

Der Rumpf der `qsort`-Funktion definiert das Abbruchkriterium für den Quicksort. Ist die Collection leer (`empty?`), gibt sie eine leere Sequenz `()` zurück; Sie könnten prinzipiell auch `col` zurückgeben, was man in der Regel jedoch nicht tun wird. Es ist klarer und expliziter und erfordert kein besonderes Nachdenken, wenn man ein `()` statt eines `col` liest für den „then“-Teil des `if`-Ausdrucks.

Den „else“-Teil des `if`-Ausdrucks bereiten wir mit einem `let` vor. Zwei Bindungen gelten in dem Kontext des `let`. Einmal wird `pvt` an das erste Element von `col` gebunden, zum anderen wird `rst` an alle übrigen Elemente von `col` exklusive Pivot-Element gebunden. Denken Sie daran, dass diese Ausdrucksweise etwas ungenau ist. Tatsächlich erzeugt `rest` eine neue Sequenz ohne das Pivot-Element. Der abschließende Kommentar deutet an, dass noch mehr kommen wird. Beachten Sie beim Schreiben von Clojure-Programmen, dass Einrückungen das Lesen beachtlich erleichtern. Niemand mag Klammern abzählen.

Lassen Sie das Programm laufen und prüfen Sie seine Reaktion direkt über die Konsole. Wenn der `CLASSPATH` für Clojure richtig gesetzt ist,[1] können Sie das Programm in der Konsole mit `load` laden und aufrufen.

```
user=> (load "qs")
nil
```

[1] Unter http://denkspuren.blogspot.com finden Sie dazu einen Hinweis.


```
user=> (qsort [4 3 1 5 7 2 6])
nil
user=> (qsort [])
()
```

Alles verhält sich wie erwartet. Ein leerer Vektor liefert eine leere Sequenz zurück, ein befüllter Vektor liefert ein `nil` als Folge eines nicht ausformulierten `let`-Kontextes.

Entfernen Sie nun den Kommentar und ersetzen Sie ihn mit dem folgenden Programmausschnitt.

```
(concat
  (qsort (filter (fn [itm] (< itm pvt)) rst))
  (list pvt)
  (qsort (remove (fn [itm] (< itm pvt)) rst)))
```

Diese Zeilen sollten sich Ihnen vollständig erschließen. Der Unterschied zu unseren anfänglichen Experimenten in der Konsole liegt darin, dass wir nun statt mit konkreten Werten mit Variablen arbeiten. Hinzu kommt noch die Rekursion, der Aufruf von `qsort` mit der über `filter` bzw. `remove` erzeugten Teilsequenzen. Bei der Konkatenation habe ich mich entschieden, dass Pivot-Element in eine Sequenz zu packen statt in einen Vektor. Sie können auch `[pvt]` statt `(list pvt)` schreiben. Probieren Sie es aus!

Lassen Sie uns ein paar Probeläufe mit dem Programm machen.

```
user=> (qsort [4 3 1 5 7 2 6])
(1 2 3 4 5 6 7)
user=> (qsort [1 2 3])
(1 2 3)
user=> (qsort [3 2 1])
(1 2 3)
```

5.4 Testfälle

Alles läuft wie erwartet. Um das Programm einigen Tests zu unterwerfen, fügen wir dem Programm `qs.clj` zu Beginn die folgende Zeile hinzu, um die Infrastruktur für Unit-Tests nutzen zu können.

```
(use 'clojure.test)
```

Wenn Sie Testfälle formulieren, konfrontieren Sie Ihren Code möglichst mit „pathologischen“ Fällen. Suchen Sie Grenzfälle und bemühen Sie sich, Ihren Code mit grundsätzlich gültigen Eingaben zu brechen. Fügen Sie nach der `qsort`-Funktion zum Beispiel folgende Tests hinzu.

```
(is (= (qsort [4 3 1 5 7 2 6]) [1 2 3 4 5 6 7]))
(is (= (qsort [1 2 3 4 5 6 7]) [1 2 3 4 5 6 7]))
(is (= (qsort [2 3 2]) [2 2 3]))
(is (= (qsort [2 2 2]) [2 2 2]))
(is (= (qsort [7]) [7]))
(is (= (qsort []) []))
```

Mit jedem Laden Ihres Code aus der Datei z.B. per `load` wird `qsort` automatisch diesen Tests unterzogen. Das erhöht die Zuverlässigkeit und das Vertrauen in Ihren Code. Das gilt vor allem dann, wenn Sie Änderungen an Ihrem Code vornehmen.

5.5 Überarbeitung

Der Code realisiert einen Quicksort. Man könnte sich damit zufrieden geben. Allerdings beschränkt sich der Quicksort auf das Sortieren von Zahlenfolgen. Das ist ein unnötige Einschränkung. Der Algorithmus ist unabhängig von den verwendeten Datentypen. Versuchen wir eine Generalisierung.

Man kann den Quicksort-Algorithmus mit einer Sammlung beliebiger Daten durchführen, sofern man eine Funktion hat, die als Kriterium dient, eine Rangfolge unter den Daten festzulegen. Welches Datum ist „größer“ oder „kleiner“ als ein anderes Datum? Eine solche Funktion kann zwei Daten entgegen nehmen und liefert als Ergebnis ein `true` oder `false` zurück; wir benötigen also eine entsprechende Prädikatsfunktion.

Erweitern wir die Liste der Argumente für `qsort` neben der Collection um die Prädikatsfunktion und passen den Funktionsrumpf entsprechend an. In Konsequenz muss der rekursive Aufruf der `qsort`-Funktion um das weitere Argument `cmp` ergänzt werden. Zusätzlich nutzen wir statt `fn` die Schreibweise `#(...)` (ein Reader-Makro) für einfache, anonyme Funktionen; das würzt den Code mit Kürze.

```
(defn qsort
  "Quicksort data collection according to a cmp function"
  [col cmp]
  (if (empty? col)
    ()
    (let [pvt (first col)
          rst (rest col)]
          (concat
            (qsort (filter #(cmp % pvt) rst) cmp)
            (list pvt)
            (qsort (remove #(cmp % pvt) rst) cmp)))))
```

Der Code arbeitet tadellos, wenn ihm z.B. beim Sortieren von Zahlenfolgen eine Vergleichsfunktion wie `<` oder `>=` übergeben wird. Das hat einen interessanten Effekt. Mit der Vergleichsfunktion ist steuerbar, ob die Sortierung auf- oder absteigend erfolgt.

```
user=> (qsort [4 3 1 5 7 2 6] <)
(1 2 3 4 5 6 7)
user=> (qsort [4 3 1 5 7 2 6] >=)
(7 6 5 4 3 2 1)
```

Eine Funktion, die eine andere Funktion als Argument hat, wird auch als Funktion höherer Ordnung (*Higher-Order Function, HOF*) bezeichnet. Die überarbeitete, generalisierte `qsort`-Funktion ist eine HOF.

Unglücklicherweise brechen mit dieser `qsort`-Funktion die bisherigen Testfälle, da das zusätzliche `cmp`-Argument fehlt. Clojure bietet die Möglichkeit, unterschiedliche Anzahlen von Argumenten beim Aufruf von Funktionen zu unterscheiden. Ergänzen wir den Code um die Entgegennahme eines einziges Arguments. Wir gehen dann davon aus, dass es sich in einem solchen Fall um eine Zahlenfolge handelt und standardmäßig eine Sortierung gemäß des „Kleiner“-Kriteriums mit der Funktion `<` erfolgt. Beachten Sie, dass nun jede Realisierung in einem gesonderten runden Klammerpaar „eingepackt“ ist.

```
(defn qsort
```

```
  "Quicksort data collection according to a cmp function"
  ([col]
    (qsort col <))
  ([col cmp]
    (if (empty? col)
      ()
      (let [pvt (first col)
            rst (rest col)]
            (concat
              (qsort (filter #(cmp % pvt) rst) cmp)
              (list pvt)
              (qsort (remove #(cmp % pvt) rst) cmp))))))
```

Nun funktioniert die bisherige Aufrufweise mit der Collection als einzigem Argument wieder.

```
user=> (qsort [4 3 1 5 7 2 6])
(1 2 3 4 5 6 7)
```

Alle Testfälle passieren den Code ohne Fehlermeldung. Allerdings ist es angeraten, ein paar weitere Testfälle hinzuzunehmen, die den Quicksort mit zwei Argumenten aufrufen. Wieder gehen wir davon aus, dass die Eingaben zu `qsort` grundsätzlich erlaubt sind. Die Ergänzungen folgen den Testfällen in der Datei `qs.clj`.

```
(is (= (qsort [4 3 1 5 7 2 6] <) [1 2 3 4 5 6 7]))
(is (= (qsort [4 3 1 5 7 2 6] <=) [1 2 3 4 5 6 7]))
(is (= (qsort [4 3 1 5 7 2 6] >) [7 6 5 4 3 2 1]))
(is (= (qsort [4 3 1 5 7 2 6] =) [4 3 1 5 7 2 6]))
(is (= (qsort [4 3 1 5 4 7 2 6 1] =) [4 4 3 1 1 5 7 2 6]))
```

Können Sie sich das Verhalten für den Gleichheitsfall in den beiden letzten Testfällen erklären? Generalisierungen von Algorithmen haben teilweise interessante Verhaltensweisen, die sich gezielt nutzen lassen. Jetzt kennen Sie beispielsweise eine Möglichkeit, in einer Collection gleiche Daten zu clustern, ohne die grundsätzliche Reihenfolge der Elemente zu verändern.

Der Code ist nun generalisiert und eine präzise und leicht zugängliche Beschreibung des Quicksort-Algorithmus'. Ein Anlaß für eine Überarbeitung ist, mit so genannten *lazy sequences* zu arbeiten. Wir verzichten an dieser Stelle darauf.

▷ **Aufgabe 5.1** Schreiben sie eine Funktion namens `str-cmp`, mit der Sie den generalisierten Quicksort für die Sortierung von Zeichenketten konfigurieren. Die Sortierung soll alphabetisch aufsteigend sein und Groß- bzw. Kleinschreibung ignorieren. Folgende Testfälle soll Ihre Lösung erfolgreich bestehen:

```
(is (= (qsort ["aa" "b" "ba" "a"] str-cmp)
       ["a" "aa" "b" "ba"]))
(is (= (qsort
  ["A" "couple" "of" "Strings" "to" "be" "sorted"] str-cmp)
  ["A" "be" "couple" "of" "sorted" "Strings" "to"]))
```

Hinweis: Es gibt in Clojure die Bibliothek `clojure.string` für String-Funktionen.

5.6 Lösungen

Aufgabe 5.1 Die Lösung lautet:

```
(use '[clojure.string :only (lower-case)])
(defn str-cmp [str1 str2]
  (< (compare (lower-case str1) (lower-case str2)) 0))
```

Wenn Sie nur `(use 'clojure.string)` verwenden, weist Sie Clojure auf Konflikte in Namensräumen hin. Sie dürfen das im Rahmen dieser Aufgabe ignorieren. Die angegebene Lösung importiert aus `clojure.string` nur die Funktion `lower-case`.

Historie

22. Mai 2012 Lösungskapitel im Inhaltsverzeichnis

27. Okt. 2011 Zwei kleine Korrekturren und eine neue Übungsaufgabe. Mit Dank an Marc Hesenius.

22. Apr. 2011 Erste Version.

6 Fallstudie: Flipflop

6.1 Aufbau und Funktionsweise eines Flipflops

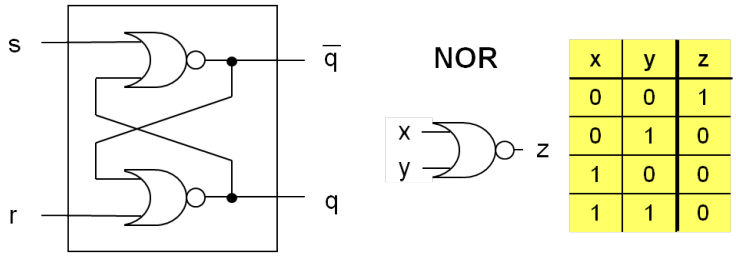
Das Flipflop ist ein elementarer Baustein in der Schaltungstechnik. Es kann entweder eine Information von einem Bit speichern oder im Schwingzustand als Oszillator dienen.

Zur Auffrischung und Funktionsweise: Das RS-Flipflop besteht aus zwei NOR-Gattern. Ein Eingang des einen NOR-Gatters heiße s (für „setzen“, *set*), ein Eingang des anderen NOR-Gatters r (für „zurücksetzen“, *reset*). Der Ausgang des NOR-Gatters mit dem s-Eingang heiße $\bar{q}$, der Ausgang des anderen NOR-Gatters q. Die Ausgänge sind mit dem jeweils freien Eingang des anderen NOR-Gatters verbunden. Die Schaltung ist in Abb. 6.1 dargestellt.

Das Verhalten des Flipflops erschließt sich, wenn man die zeitliche Dimension berücksichtigt. Es genügt ein einfaches zeitdiskretes Modell. Jedes Schaltglied verbrauche eine konstante Zeit Δt für den Schaltvorgang. Die Werte an den Eingängen eines Gatters zum Zeitpunkt t führen zu einem Ergebnis am Ausgang zum Zeitpunkt $t + \Delta t$. Leitungen hingegen übertragen Informationen unmittelbar und ohne zeitliche Verluste. Der Einfachheit halber zählen wir die Zeitschritte mit einem ganzzahligen Zeitindex durch.

Das heißt für das Flipflop: Zu einem beliebigen Zeitpunkt t liegen an den Eingängen der NOR-Gatter zum einen die „von außen“ zugeführten Werte s_t und r_t an, zum anderen liegen die „von innen“ rückgeführten Ausgangswerte q_t und $\bar{q}_t$ an. Zum nächsten Zeitpunkt $t + 1$ bestimmen sich die neuen Ausgangswerte wie folgt: $\bar{q}_{t+1} = NOR(s_t, q_t)$ und $q_{t+1} = NOR(\bar{q}_t, r_t)$. Die Eingangswerte für den nächsten Schritt in der Ausführung vom Zeitpunkt $t+1$ hin zum Zeitpunkt $t+2$ sind zum einen die „Außenwerte“ s_{t+1} und r_{t+1} und zum anderen die soeben ermittelten „Innenwerte“ q_{t+1} und $\bar{q}_{t+1}$. Dieses Schema der Zuführung von Werten für s und r und der Rückkopplung der Ausgangswerte setzt sich endlos fort. Tab. 6.1 stellt die Ergebnisse für einen Zeitschritt aus den 16 möglichen Eingangskombinationen von $\bar{q}_t$, q_t, s_t und r_t übersichtlich zusammen.

▷ **Aufgabe 6.1** Erstellen Sie eine Funktion `ff`, die die Funktion eines RS-Flipflops nachbildet und komponiert ist aus zwei NOR-Gattern! Das NOR-Gatter werde durch die Funktion `nor` umgesetzt.



x	y	z
0	0	1
0	1	0
1	0	0
1	1	0

Abbildung 6.1: RS-Flipflop, aus zwei NOR-Gattern aufgebaut

$\overline{q}_t$	q_t	s_t	r_t	$\overline{q}_{t+1}$	q_{t+1}	
1	x	1	0	0	0	set
0	x	1	0	0	1	
x	1	0	1	0	0	reset
x	0	0	1	1	0	
x	x	1	1	0	0	invalid
0	0	0	0	1	1	idle⁺
1	1	0	0	0	0	
0	1	0	0	0	1	idle‡
1	0	0	0	1	0	

\+ instabil, schwingend
‡ stabil, zustandserhaltend

Tabelle 6.1: Wertetabelle zum Flipflop; `x` steht für 0 bzw. 1

Bevor Sie weiterlesen sei Ihnen empfohlen, mindestens 15 Minuten für den Versuch einer Lösung aufzuwenden. Nur so werden Sie das Problem entdecken, was es zu lösen gilt.

6.2 1. Lösungsansatz: Rückgabe einer Fortsetzung

Das NOR-Gatter kann als einfacher Einzeiler realisiert werden; die Funktion ist so kurz und übersichtlich, dass man sie nicht über mehrere Zeilen strecken sollte.

```
(defn nor [x y] (if (and (zero? x) (zero? y)) 1 0))
```

Bestimmt sind Sie mit Ihrer Lösung zum Flipflop so weit gekommen:

```
(defn ff [nq q s r]
  (let [q+ (nor r nq) nq+ (nor s q)]
   ; ???
))
```

Das Flipflop nimmt die vier Eingangswerte für einen Zeitpunkt t auf (`nq` steht für $\bar{q}$) und bindet die Ergebnisse für die Ausgänge an `q+` und `nq+` für den Zeitpunkt $t + 1$.

Nun gilt es, die Ausgangswerte der NOR-Gatter zurück zu geben, die Rückkopplung zu realisieren und neue Werte „von außen“ für `s` und `r` entgegen zu nehmen. Das scheint unmöglich zu realisieren. Wir können keine Rückgabe machen *und* gleichzeitig neue Werte als Eingabe annehmen. Das Ende der Ausführung einer Funktion, sprich die Rückgabe eines Ergebnisses, verträgt sich nicht mit dem gleichzeitigen Neustart der Funktion.

Die Lösung des Problems besteht darin, die Funktion `ff` nicht vollständig neu aufzurufen, sondern die Funktion als Datum („Closure“ genannt) zusammen mit `nq+` und `q+` zurück zu geben. Bleibt die Frage, wie man eine Closure mit einer Bindung der Argumente `q` und `nq` an die Werte `q+` und `nq+` hinbekommt. Hier hilft der Einsatz von `partial`. Die Werte werden zusammen mit der Closure als Vektor zurückgegeben.

```
(defn ff [nq q s r]
  (let [nq+ (nor s q) q+ (nor r nq)]
    [nq+ q+ (partial ff nq+ q+)]))
```

Rufen wir die Funktion mit den folgenden Werten auf, dann ergeben sich als korrektes Ergebnis im Vektor zwei `0`-Werte (siehe auch erste Zeile in Tab. 6.1). Als drittes Element folgt die Clojure.

```
user=> (ff 1 0 1 0)
[0 0 #<core$partial$fn__3680 clojure.core$partial$fn__...>]
```

In der Closure ist der aktuelle Zustand des Flipflops gekapselt. Der interne Zustand, repräsentiert durch die Argumente `nq` und `q` von `ff`, wird mittels `partial` an die Werte `nq+` und `q+` gebunden; `partial` liefert eine Funktion, genauer, eine Clojure zurück mit den noch ausstehenden Argumenten `s` und `r`.

Man kann auf den Einsatz von `partial` gänzlich verzichten. Die nachstehende Umsetzung `ff2` des Flipflops ist der Funktion `ff` sogar vorzuziehen. Die Funktion `ff2` trennt klar zwischen der Initialisierung des Flipflops mit den internen Werten `nq` und `q` und der Verarbeitung der „Außenwerte“ `s` und `r`.

```
(defn ff2 [nq q] ; initialize flipflop
  (fn [s r]
    (let [nq+ (nor s q) q+ (nor r nq)]
      [nq+ q+ (ff2 nq+ q+)])))
```

Die Interaktion mit `ff2` gestaltet sich geringfügig anders.

```
user=> ((ff2 1 0) 1 0)
[0 0 #<user$ff2$fn__60 user$ff2$fn__60@7f2e7c6a>]
```

Um die Interaktion mit dem Flipflop fortzusetzen, muss die Funktion zunächst z.B. per `last` aus dem Vektor geholt werden. Anschließend kann die Funktion auf ein weiteres Eingabepaar angewendet werden.

```
user=> ((last ((ff2 1 0) 1 0)) 1 0)
[0 1 #<user$ff2$fn__60 user$ff2$fn__60@5d04e28b>]
```

Ein Abgleich mit Tab. 6.1 bestätigt das Ergebnis. Zuerst kommt die erste Zeile zur Anwendung, dann die zweite. Der interne Zustandsübertrag entspricht dem Abgleich des Ergebnisse von $\bar{q}$ und q ganz rechts aus der ersten Zeile mit den Werten ganz links in der zweiten Zeile.

▷ **Aufgabe 6.2** Was ist davon zu halten, ein `println` gemäß `ff2p` zur Ausgabe der aktuellen Flipflop-Ausgabe zu nutzen?

```
(defn ff2p [nq q]
  (fn [s r]
    (let [nq+ (nor s q) q+ (nor r nq)]
      (println nq+ q+)
      (ff2p nq+ q+))))
```

Ziehen wir ein Resümee: Die Folge von Werten für `s` und `r` muss für die Ausgabe des aktuellen Ergebnisses, den Ausgangswerten der NOR-Gatter, unterbrochen werden. Die Fortsetzung (*continuation*) steht mit der ebenso zurückgegebenen Clojure zur Verfügung. Die Clojure enthält den notwendigen Bindungskontext, hier `nq` und `q`; die Bindungen repräsentieren den Zustand des Flipflops.

6.3 2. Lösungsansatz: Historien mit Sequenzen abbilden

Diese Zerlegung der zeitlichen Abläufe in ein Wechselspiel von Ein- und Ausgaben hat eine Alternative. Man kann die gesamte Interaktion mit dem Flipflop durch Historien von Binärwerten für den `s`- bzw. `r`-Eingang verdichten. So repräsentiere beispielsweise die Folge `[1 1 0]` von links nach rechts gelesen die Werte s_0, s_1 und s_2 für den `s`-Eingang und `[0 0 0]` die Wertfolge r_0, r_1 und r_2 für den `r`-Eingang.

Mit der Idee der Historien verändert sich der Code zum Flipflop wie folgt.

```
(defn ff3 [nq q s r]
  (if (and (seq s) (seq r))
    (let [nq+ (nor (first s) q) q+ (nor (first r) nq)]
      (concat [nq+ q+] (ff3 nq+ q+ (rest s) (rest r))))
    []))
```

Die Interaktion gestaltet sich wie erwartet mit den angegebenen beispielhaften Historien für `s` und `r`. Die ersten beiden Werte initialisieren das Flipflop, dann folgen die Historien. Als Ergebnis ergibt sich eine Historie, die die Werte für $\bar{q}_0$, q_0, $\bar{q}_1$, q_1 und $\bar{q}_2$, q_2 (von links nach rechts gelesen) angibt.

```
user=> (ff3 1 0 [1 1 0] [0 0 0])
(0 0 0 1 0 1)
```

Mit Clojure ist eine dritte Variante möglich, die sich fast wie ein Verschnitt aus den beiden vorangegangenen Ansätzen `ff1` bzw. `ff2` und `ff3` gibt. Die Historien werden nachwievor über Sequenzen abgebildet. Die Berechnung von Ausgabewerten wird solange verzögert – man könnte auch sagen „vermieden“ –, bis ein Wert eingefordert wird. Dieses Verhalten kapselt Clojure weg und bezeichnet derartige Sequenzes als *lazy sequences*, zu deutsch „verzögerte Sequenzen“.

```
(defn ff4 [nq q s r]
  (if (and (seq s) (seq r))
    (let [nq+ (nor (first s) q) q+ (nor (first r) nq)]
      (lazy-cat [nq+ q+] (ff4 nq+ q+ (rest s) (rest r))))
    []))
```

Der einzige Unterschied zwischen `ff4` und `ff3` liegt in der Verwendung von `lazy-cat` (*lazy concatenation*) statt `concat`. Auf den ersten Blick scheinen sich die Programme in ihrem Verhalten nicht zu unterscheiden. Über die Konsole zeigt sich ein gleiches Verhalten.

```
user=> (ff4 1 0 [1 1 0] [0 0 0])
(0 0 0 1 0 1)
```

Die interne Datenhaltung ist jedoch gänzlich verschieden. In einer durch `lazy-cat` verzögerten Sequenz werden die Werte erst dann bereitgestellt, wenn sie angefordert werden. Für `ff4` bedeutet das, dass `s` und `r` erst dann mit `rest` „verkürzt“ werden, wenn der rekursive Aufruf tatsächlich notwendig ist. Beispielsweise liefert

```
user=> (take 2 (ff4 1 0 [1 1 0] [0 0 0]))
(0 0)
```

nur die Werte $\bar{q}_0$ und q_0 für den ersten Zeitschritt. Alle weiteren Zeitschritte werden erst gar nicht berechnet.

Noch deutlicher wird das, wenn wir für `s` und `r` ebenfalls verzögerte Sequenzen nutzen, die unendlich viele Werte bereitstellen; das geht beispielsweise mit `cycle`. Geben Sie Folgendes an der Konsole ein, und Sie werden zwei Beobachtungen machen:

```
user=> (ff4 1 0 (cycle [1 1 0]) (cycle [0 0 0]))
(0 0 0 1 0 1 0 1 0 1 0 1 0 1 ... ; infinite output
```

1. Die Berechnung findet keinen Abbruch und füllt die Konsole mit einer unendlichen Binärfolge. Das ist konsequent und folgerichtig, da der rekursive Aufruf von `ff4` niemals das Abbruchkriterium erreicht. Die unendlichen Eingangssequenzen verhindern ein Ausführungsende. Es hilft nur ein gewaltsamer Abbruch der Ausführung von Clojure über die Konsole.
2. Es findet trotz Rekursion von `ff4` kein Stapelüberlauf (*stack overflow*) statt, wie das bei „normaler“ Rekursion ohne Verzögerung der Fall wäre. So meldet der gleiche Aufruf mit `ff3` unmittelbar einen Überlauffehler des Stapels zurück.

   ```
   user=> (ff3 1 0 (cycle [1 1 0]) (cycle [0 0 0]))
   java.lang.StackOverflowError (NO_SOURCE_FILE:0)
   ```

Beschränkt man das Resultat in der Rückgabe auf eine endliche Auswahl an Elementen, wird sofort klar: Die Verzögerung sorgt dafür, dass die Rückgabesequenz nur soweit wie nötig berechnet wird. Wäre dem nicht so, dann käme `take` nie zum Zuge, da `ff4` – wie eben gezeigt – prinzipiell unendlich lange und speicherfüllend die Ergebnissequenz zu berechnen versuchte.

```
user=> (take 6 (ff4 1 0 (cycle [1 1 0]) (cycle [0 0 0])))
(0 0 0 1 0 1)
```

Dass `ff4` tatsächlich eine verzögerte Sequenz (*lazy sequence*) liefert, lässt sich mit `type` überprüfen.

```
user=> (type (ff4 1 0 (cycle [1 1 0]) (cycle [0 0 0])))
clojure.lang.LazySeq
```

▷ **Aufgabe 6.3** Schreiben Sie eine Funktion `account`, die den Zustand eines Bankkontos verwaltet. Positive Beträge gelten als Abhebungen, negative als Einzahlungen. Vollziehen Sie mit zwei Varianten von `account` die aus dem Flipflop-Beispiel bekannten Lösungsansätze nach: ohne und mit (verzögerten) Sequenzen.

6.4 Reflektion

Fassen wir zusammen: In einer funktionalen Welt („gleicher Input, gleicher Output“) muss der Aspekt der Zeit, der einher geht mit Zustandsänderungen, ausdrücklich modelliert und berücksichtigt werden. Die Zerlegung zeitlicher Vorgänge in diskrete Wertfolgen ist unumgänglich. Wir haben das Zeitphänomen und die Behandlung von Zustand am Beispiel des Flipflops diskutiert.

Zwei Techniken haben wir betrachtet: Zum einen kann der Zustand in einer Closure über einen aufgesetzten Bindungskontext „gespeichert“ werden. Zumindest bei der direkten Interaktion über die Konsole bekommt

eine derartig genutzte Closure die Qualität eines Objekts mit einer einzigen Methode.[1]

Die andere Technik besteht darin, zeitliche Verläufe von Ein- und Ausgaben als Historien von Werten zu betrachten und über Sequenzen abzubilden. Problematisch ist bei der Verarbeitung großer Eingangshistorien der drohende Stack-Overflow bei der Berechnung der Ausgangshistorie. Die rekursive Konstruktion einer Rückgabesequenz verbietet aufgrund der „hängenden“, stückweisen Konstruktion der Rückgabesequenz eine Endrekursion. Selbst wenn das Rekursionslimit nicht erreicht wird, erzwingen umfangreiche Eingangshistorien die vollständige Berechnung der Ausgangshistorie. Wenngleich es „normal“ für Funktionen ist, von einer Eingabe getrieben die gesamte Ausgabe zu berechnen, so ist dieses Verhalten für große Datenmengen in der Regel eine erhebliche Verschwendung von Rechenressourcen. Eigentlich braucht nur das berechnet zu werden, was an Ausgaben von einer weiterverarbeitenden Funktion tatsächlich benötigt wird.

Nutzt die verarbeitende Funktion eine verzögerte Sequenz als Historien-Rückgabe, so wandelt sich eine „hängende“ Rekursion um in eine inkrementelle Rekursion, die nicht mehr Gefahr läuft, einem Stack-Overflow zum Opfer zu fallen. Die Rekursion zur Konstruktion einer verzögerten Sequenz ist modulo der Konstruktion endrekursiv. Wie viele Werte der verzögerten Ausgabesequenz berechnet werden, hängt ausschließlich vom „Datenhunger“ der sie nachverarbeitenden Funktion ab. Konsequenterweise müssen auch nicht alle Werte der Eingangshistorien bekannt sein. Die Eingangshistorien können ebenfalls mit verzögerten Sequenzen abgebildet werden.

Die Programmierung mit verzögerten Listen wandelt die *push*-basierte Verarbeitung einer Funktion um in eine *pull*-basierte Verarbeitung. Im ersten Fall stößt (*push*) ein Aufruf mit Eingabewerten die vollständige Berechnung eines Ausgabewerts an. Im zweiten Fall werden mit dem schrittweisen dekonstruierenden Abgriff (*pull*) von Werten der verzögerten Ausgabesequenz nur die wirklich benötigten Sequenzwerte berechnet. Das hat zur Folge, dass auch von einer Eingabesequenz nicht mehr an Werten abgegriffen wird als nötig.

Verzögerte Sequenzen sind für die effiziente Nutzung von Rechenressourcen („Rechne nicht mehr als nötig!“) derart interessant, dass viele Funktionen in Clojure verzögerte Sequenzen zurückgeben, so zum Beispiel die Funktionen `map` und `filter`. Dank Verzögerung sind auch rekursive und damit prinzipiell unendlich lange Wertfolgen definierbar wie z.B. mit `cycle`.

Die Funktionen zum Umgang mit „normalen“ und verzögerten Sequenzen unterscheiden sich dem Namen nach nicht; Clojure suggeriert ein einheitliches Interface. Die notwendige unterschiedliche Abarbeitung erledigt Clojure im Hintergrund. Bei der Arbeit an der Konsole fällt einem deshalb in der Regel gar nicht auf, wann und ob Clojure mit verzögerten statt gewöhnlichen Sequenzen arbeitet. Dennoch sollten Sie bei der Programmierung darüber nachdenken, ob eine Funktion statt einer gewöhnlichen Sequenz als Rückegabewert auch eine verzögerte Sequenz zurückliefern kann und sollte.

[1] Aus diesem Gedanken heraus stammt der Spruch „Closures are a poor man's objects“. Den Kontrapunkt setzt „Objects are a poor man's closures“. Siehe auch http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html.

Historie

22. Mai 2012 Schreibkorrektur; mit Dank an Nicolas de Klerk

10. Juli 2011 Korrektur eines Schreibfehlers und eines Trennfehlers von LaTeX.

27. Mai 2011 Erste Version.

7 Fallstudie: Peano-Zahlen

Peano-Zahlen basieren auf einem sehr einfachen System zur Konstruktion von Zahlen: (1) Es gibt eine Zahl, die wir „Null“ nennen. (2) Alle weiteren Zahlen sind über die Nachfolge definiert. Der Nachfolger von Null ist das, was wir „Eins“ nennen. Der Nachfolger von Eins ist das, was wir „Zwei“ nennen. Die Zwei ist der Nachfolger des Nachfolgers von Null. Gemäß diesem Nachfolge-Prinzip lassen sich alle natürlichen Zahlen $\mathbb{N}_0$ erzeugen. Die Idee geht zurück auf den italienischen Mathematiker Peano und ist festgeschrieben in den so genannten Peano-Axiomen.

Dieses einfache System kann auf ganzzahlige Peano-Zahlen erweitert werden, indem von Null ausgehend nicht nur die Nachfolger, sondern auch die Vorgänger betrachtet werden. Sei p eine ganzzahlige Peano-Zahl, so ist $succ(p)$ ihr Nachfolger (*successor*) und $pred(p)$ ihr Vorgänger (*predecessor*). Über alle Nachfolger von Null ergeben sich die positiven Ganzzahlen, über alle Vorgänger die negativen Ganzzahlen. Zusammen mit der Null spannt diese Definition den Raum der ganzen Zahlen (*integers*) $\mathbb{Z}$ auf. Es gilt $succ(pred(p)) = p$ und $pred(succ(p)) = p$.

In dieser Fallstudie werden wir die Grundrechenarten für ganzzahligen Peano-Zahlen definieren.

7.1 Ein Namensraum für Peano-Zahlen

Um die von Clojure bekannten arithmetischen Funktionen wie `+`, `-`, `*`, `==` etc. für Peano-Zahlen verwenden zu können, legen wir einen eigenen Namensraum an. Das Anlegen eines Namensraums mit `ns` importiert automatisch alle Funktionen aus dem Clojure-Kern. In aller Regel ist das sinnvoll. In unserem speziellen Fall müssen wir zur Vermeidung von Namenskonflikten eine Reihe arithmetischer Funktionen aus dem Clojure-Kern ausschließen. Um während der Entwicklung jederzeit Testfälle formulieren zu können, ist die Testumgebung einzubinden.

```
(ns pno
  #^{:doc "Arithmetics for integer peano numbers"
     :author "Dominikus Herzberg, Heilbronn University"}
  (:refer-clojure
     :exclude (zero? == + - * / mod pos? neg? inc dec))
  (:use clojure.test))
```

Üblich ist noch das Hinzufügen von Meta-Informationen. Als Standard haben sich die Keywords `:doc` und `:author` eingebürgert. Es steht Ihnen frei, beliebige weitere Meta-Informationen anzulegen.

Eröffnen Sie mit dieser Einleitung eine Clojure-Datei z.B. unter dem Namen `pno.clj`. Sie können in der REPL vom User-Namensraum die Datei laden und anschließend in den Namensraum wechseln.

```
user=> (load "pno") ; also for reload while in namespace pno
nil
user=> (in-ns 'pno)
#<Namespace pno>
```

```
pno=>
```

Ein Hinweis: Achten Sie darauf, keine Definition unter dem gleichen Namen wie dem Namen des Namensraums anzulegen.

7.2 Darstellung von ganzzahligen Peano-Zahlen

Man kann ganzzahlige Peano-Zahlen auf verschiedene Weise repräsentieren. Wir verwenden für die interne Darstellung von ganzzahligen Peano-Zahlen ein einfaches Schema. Das Schema für die Datenkodierung wird durch die Prädikate `zero?` und `is-peano?` abgebildet.

Die Peano-Zahl „Null“ sei durch die leere Liste `()` repräsentiert. Das wollen wir festhalten mit dem Prädikat `zero?`. Da wir die Übernahme von `zero?` aus dem Clojure-Kern in den Namensraum `pno` ausgeschlossen haben, können wir die entsprechende Funktion für Peano-Zahlen sorglos unter diesem Namen einführen.

```
(defn zero? [p] (= () p))
```

▷ **Aufgabe 7.1** Die Definition `(defn zero? [p] (empty? p))` erfüllt nicht ihren Zweck. Warum?

Nachfolger bzw. Vorgänger von der Peano-Null werden durch verschachtelte Listen der Form `(succ` p`)` bzw. `(pred` p`)` kodiert; p steht für eine Peano-Zahl, die ihrerseits der Form `(succ` p`)` bzw. `(pred` p`)` folgt. Alternativ kann p die Peano-Null `()` sein. Anders formuliert: Eine ganzzahlige Peano-Zahl ist entweder über die Peano-Null kodiert oder als zweielementige Liste. Das erste Element der Liste ist das Symbol `succ` oder `pred`, das zweite wiederum eine kodierte Peano-Ganzzahl.

Genau dieses Kodierschema für Peano-Ganzzahlen fasst das Prädikat `is-peano?`.

```
(defn- is-peano? [[fst snd :as p]]
  (or (= fst 'succ) (= fst 'pred) (zero? p)))
```

Das Prädikat `is-peano?` folgt dem beschriebenen Kodierschema nur “zur Hälfte“. Die Definition für die Datenrepräsentation von ganzzahligen Peano-Zahlen ist rekursiv. Das Prädikat `is-peano?` ist jedoch nicht rekursiv definiert. Das zweite Element der Paar-Darstellung wird nicht darauf überprüft, ob es seinerseits eine Peano-Zahl ist.

Der Verzicht auf die Rekursion hat einen Grund. Alle grundlegenden Funktionen, die Peano-Zahlen erzeugen oder zerlegen, tun dies schrittweise konstruktiv oder rekursiv. In diesen Funktionen findet sich direkt bzw. implizit die Rekursion, die wir in `is-peano?` nicht zusätzlich anlegen. Das Prädikat taugt also nicht zur vollständigen, sondern nur zur Teilüberprüfung, ob eine gültige Datenrepräsentation für Peano-Zahlen vorliegt. Das Prädikat sollte also nicht öffentlich verfügbar sein und wird mit `defn-` als private Funktion deklariert.

▷ **Aufgabe 7.2** Definieren Sie ein Prädikat `peano?`, das vollständig überprüft, ob eine gültige Peano-Ganzzahl als Argument vorliegt.

7.3 Die Konstruktion von Peano-Zahlen

Die Funktionen zur Konstruktion von Nachfolgern und Vorgängern von Peano-Zahlen heißen `succ` und `pred`. Beide Funktionen erzeugen ausschließlich Peano-Ganzzahlen gemäß der vereinbarten Datenkodierung. Die Funktionen setzen unmittelbar die Gleichungen $succ(pred(p)) = p$ und $pred(succ(p)) = p$ um.

```
(defn succ [[fst snd :as p]]
  {:pre [(is-peano? p)]}
  (if (= fst 'pred) snd (list 'succ p)))

(defn pred [[fst snd :as p]]
  {:pre [(is-peano? p)]}
  (if (= fst 'succ) snd (list 'pred p)))
```

Die Funktionen erzeugen durch die Anwendung der Gleichungen eine normalisierte Form der internen Repräsentation ganzzahliger Peano-Zahlen. Normalisierung bedeutet: Die Zahlen werden mit einer minimalen Anzahl von verschachtelten `succ`- bzw. `pred`-Symbolen kodiert. Das ist nur möglich, wenn neben der Peano-Null die Zahlenrepräsentationen ausschließlich entweder aus `succ`-Symbolen oder `pred`-Symbolen bestehen.

```
pno=> (succ ())
(succ ())
pno=> (succ (succ ()))
(succ (succ ()))
pno=> (succ (pred (succ ())))
(succ ())
```

Die Vorbedingung (*precondition*) in den Funktionen `succ` und `pred` stellt sicher, dass nur zulässige Peano-Kodierungen erstellt werden können. Machen Sie sich das unbedingt klar! Die schrittweise Konstruktion von Peano-Zahlen basiert immer auf einem gültigen Vorgänger oder Nachfolger. Deshalb muss `is-peano?` in der Vorbedingung nicht rekursiv angelegt werden. Es ist unmöglich, eine Peano-Zahl zu erstellen, die sich nicht an die Kodierungsvereinbarung hält. Ähnlich, wie Ihnen der Clojure-Compiler bei der Kodierung `12:4` (statt `12.4`) in die Eisen steigt und diese Schreibweise als „invalid number“ verweigert, werden Ihnen `succ` und `pred` dazwischen gehen, wenn Sie unsinnige Peano-Zahlen erstellen wollen.

Einige der folgenden Funktionen setzen die normalisierte Zahlendarstellung voraus. Zum Beispiel kann die bereits eingeführte Funktion `zero?` nur normalisierte Peano-Kodierungen sinnvoll verarbeiten. Die Kodierung `'(succ (pred ()))` wird – da nicht normalisiert – von `zero?` nicht als Peano-Null erkannt.

Ihnen ist sicher nicht entgangen, dass die Kodierung von Peano-Zahlen gültige Ausdrücke zur Konstruktion von Peano-Zahlen sind. Interpretiert man eine Zahlenkodierung als Programm, erhält man die normalisierte Zahlenkodierung. Anders ausgedrückt: Die normalisierte Zahlenkodierung ist ein Fixpunkt unter Anwendung der `eval`-Funktion. Nehmen wir z.B. die Peano-Zahl aus der letzten Interaktion mit der REPL. Beachten Sie die Quotierung der kodierten Peano-Zahl.

```
pno=> (eval '(succ (pred (succ ()))))
(succ ())
```


Mit lediglich einer Anwendung von `eval` kann jede gültige interne Datenrepräsentation einer Peano-Ganzzahl normalisiert werden. Trotzdem ist eine Normalisierung einer Peano-Kodierung z.B. vor Anwendung von `zero?` nicht notwendig. Warum?

Das Datenschema für Peano-Zahlen ist eine von der Entwicklerin bzw. vom Entwickler (in dem Fall von mir) gewählte *interne* Kodierung. Der Gebrauch der internen Repräsentation ist für die Anwenderin bzw. den Anwender tabu. Das ist ein wichtiges Prinzip der Softwaretechnik: Details der Kodierung und Implementierung sind zu verbergen (*information hiding*). Zur Erzeugung von Peano-Ganzzahlen sind die leere Liste und die Konstruktoren `succ` und `pred` zu verwenden. Und diese Funktionen nehmen direkt eine Normalisierung vor.

Die Peano-Ganzzahlen sind vollkommen symmetrisch konstruiert. Es ist nicht unterscheidbar, ob eine Peano-Ganzzahl, die nicht Peano-Null ist, als eine positive oder negative Zahl gilt. Schließlich ist für einen Computer die Wahl der Symbole `succ` und `pred` willkürlich und sinnfrei.

Eine Vereinbarung ist zu treffen, was als positiv und was als negativ zu gelten hat.

```
(defn pos? [p] (= (first p) 'succ))
(defn neg? [p] (= (first p) 'pred))
```

Die Funktionen `pos?` und `neg?` sind zwei Prädikate, die die Details der Kodierung von Peano-Ganzzahlen verbergen. Darüber hinaus greifen beide Prädikate die Bezeichnungen der gleichnamigen Funktionen aus dem Clojure-Core auf. Das Rechnen mit Peano-Ganzzahlen soll sich nicht vom Interface mit „normalen“ Ganzzahlen unterscheiden.

7.4 Ein Interface für den Umgang mit Peano-Zahlen

Zugegeben, es ist etwas mühsam, Peano-Ganzzahlen über die Konstruktorfunktionen zu erstellen. Zumal Sie in Dezimalzahlen denken und Ihnen größere Peano-Ganzzahlen ein mühsames Abzählen der Vorgänger bzw. Nachfolger abverlangen. Das macht das Rechnen mit Peano-Zahlen unattraktiv. Muss Ihnen als Anwenderin bzw. Anwender das interne Schema zur Kodierung von Peano-Zahlen überhaupt bekannt und verständlich sein?

Die Funktionen `peano` und `decimal` sind ein weiterer Schritt, die Interna der Kodierung und Konstruktion von Peano-Ganzzahlen zu verbergen. Mit `peano` wird eine dezimale Ganzzahl in eine Peano-Ganzzahl gewandelt.

```
(defn peano
  ([n] {:pre [(integer? n)]} (peano n ()))
  ([n p]
    (cond
      (clojure.core/zero? n) p
      (clojure.core/pos?  n) (recur (clojure.core/dec n) (succ p))
      :else                  (recur (clojure.core/inc n) (pred p)))))
```

Beispiele für den Gebrauch von `peano` sind:

```
pno=> (peano 0)
()
pno=> (peano 1)
```

```
(succ ())
pno=> (peano -2)
(pred (pred ()))
```

Die Funktion `decimal` macht das Gegenteil. Sie wandelt eine Ganzzahl in Peano-Kodierung um in eine Dezimalzahl. Die Funktion arbeitet auch mit nicht normalisierten Peano-Zahlen. Die Vorbedingung samt Rekursion stellt auch hier sicher, dass `decimal` nur gültige Peano-Kodierungen verarbeitet.

```
(defn decimal
  ([p] (decimal p 0))
  ([[fst snd :as p] n]
    {:pre [(is-peano? p)]}
    (cond
      (= fst 'pred) (recur snd (clojure.core/dec n))
      (= fst 'succ) (recur snd (clojure.core/inc n))
      :else          n)))
```

Beispiele für den Einsatz von `decimal` sind:

```
pno=> (decimal ())
0
pno=> (decimal '(pred (pred (succ ()))))
-1
pno=> (decimal (peano -7))
-7
```

Diese Wandlung von der Peano-Kodierung in die Kodierung für Dezimalzahlen und umgekehrt ist nicht vielmehr als eine Typkonvertierung. Schemata für Kodierungen sind Datentypen gleichzusetzen.

Übrigens nimmt Clojure eine entsprechende Konvertierung bei der Interaktion mit der Konsole vor. Wenn Sie in der REPL `12.3` schreiben, dann wandelt Clojure diese Repräsentation (eine Folge der Zeichen `1`, `2`, `.` und `3`) um in eine maschinennahe Zahlenkodierung im Binärformat. Von diesen Interna bekommen Sie überhaupt nichts zu sehen. Das Ergebnis einer Evaluation wird Ihnen anwendungsfreundlich wieder in der vertrauten Dezimalschreibweise auf der Konsole präsentiert. Während Clojure solche Umwandlungen beim Einlesen einer Eingabe automatisch vornimmt, müssen wir den Prozess der Wandlung von Kodierformaten explizit vornehmen.

7.5 Addition und Subtraktion

Die Addition ist für Peano-Ganzzahlen mit einem Zweizeiler definiert. Die Funktion operiert vergleichbar dem Rechnen mit Streichhölzern: Alle Hölzer der einen Seite (`p2`) werden nacheinander den Hölzern auf der anderen Seite (`p1`) hinzugefügt, bis alle Hölzer verbraucht sind. Mit dem Unterschied, dass es hier sozusagen zwei Arten von Hölzern gibt, `succ` und `pred`. Da die Bezeichnung identisch ist mit den entsprechenden Funktionen, führt ein `succ`-Hölzchen (`fst` von `p2`) zur Anwendung der Funktion `succ` auf `p1`. Die „Resthölzer“ von `p2` sind an `snd` gebunden.

```
(defn + [p1 [fst snd :as p2]]
  (if (zero? p2) p1 (recur (eval (list fst p1)) snd)))
```

Man kann sich das Vorgehen der Addition von `(peano 1)` mit `(peano -2)` schrittweise wie folgt verdeutlichen. Beachten Sie dabei sorgfältig die Unterscheidung von quotierten Ausdrücken für interne Zahlenrepräsentationen und nicht quotierten Ausdrücken, die die Konstruktorfunktionen aktivieren.

```
(0)  (+ (peano 1) (peano -2))
(1)  (+ '(succ ()) '(pred (pred ())))
(2a) (+ (pred '(succ ())) '(pred ()))
(2b) (+ () '(pred ()))
(3a) (+ (pred ()) ())
(3b) (+ '(pred ()) ())
(4)  '(pred ())
```

Die Ausgangsituation `(0)` wird in `(1)` mit den Kodierungsergebnissen dargestellt. `(2a)` zeigt, wie das `pred`-Symbol von rechts nach links „wandert“ und auf der linken Seite die Anwendung der an `pred` gebundenen Funktion auslöst. Das Ergebnis zeigt `(2b)`. Mit `(3a)` und `(3b)` geht die Endrekursion in den nächsten Durchgang. Das Abbruchkriterium ist erfüllt und `(4)` gibt das Endergebnis der Evaluation an.

Praktikabel wird das Rechnen mit Peano-Ganzzahlen mit `decimal`.

```
pno=> (decimal (+ (peano 7) (peano 3)))
10
pno=> (decimal (+ (peano -7) (peano 3)))
-4
```

Es ist schon erstaunlich, wie wenig es bedarf, um einem Computer mit einem einfachen Kodierschema und einer simplen Additionsfunktion das Rechnen beizubringen.

Die Subtraktion funktioniert ebenso einfach. Es wird lediglich das Gegenteil von dem gemacht, was die Kodierung der zweiten Zahl sagt: Liegt ein `succ` vor, wird auf die erste Peano-Zahl ein `pred` angewendet, und umgekehrt. Für diese Invertierung dient die private Hilfsfunktion `flip`.

```
(defn- flip [f] (if (= f 'succ) 'pred 'succ))

(defn - [p1 [fst snd :as p2]]
  (if (zero? p2) p1 (recur (eval (list (flip fst) p1)) snd)))
```

Nun kann der Vergleich zweier Peano-Ganzzahlen auf der Subtraktion aufbauen.

```
(defn == [p1 p2] (zero? (- p1 p2)))
```

Ein paar Beispiele demonstrieren die Substraktion und den Vergleich.

```
pno=> (decimal (- (peano 7) (peano 3)))
4
pno=> (== (peano -3) (peano 3))
false
pno=> (== (peano -3) (peano -3))
true
```

7.6 Multiplikation und Fakultätsfunktion

Mit der Addition und Subtraktion als Grundlage ist die Definition der Multiplikation kein großes Problem mehr. Die Multiplikation kann auf

eine wiederholte Anwendung der Addition bzw. Subtraktion von Peano-Ganzzahlen zurückgeführt werden. Das Ergebnis dieser wiederholten Anwendung wird als drittes Argument `res` mitgeführt. Die zweite Peano-Zahl `p2` dient als Zähler für die Anzahl der Wiederholungen, bis die Peano-Null erreicht ist.

```
(defn *
  ([p1 p2] (* p1 p2 (peano 0)))
  ([p1 p2 res]
    (if (zero? p2)
      res
      (if (pos? p2)
        (recur p1 (- p2 (peano 1)) (+ p1 res))
        (recur p1 (+ p2 (peano 1)) (- res p1))))))
```

Peano-Ganzzahlen sind keine Kodierung, mit der sich besonders effizient große Zahlen addieren oder subtrahieren lassen. Die Anwendung der Multiplikation als vervielfachte Anwendung der Addition bzw. Subtraktion lässt die Rechenlaufzeiten spürbar zunehmen. Spätestens bei Funktionen wie etwa der Fakultätsfunktion treten deutlich die Leistungsgrenzen der Peano-Kodierung zutage.

```
(defn factorial [p]
  (if (zero? p)
    (peano 1)
    (* p (factorial (- p (peano 1))))))
```

Auf meinem Rechner ist die Berechnung der Fakultät von `5` gerade noch vertretbar. Darüber hinaus muss man sich auf deutliche Wartezeiten einstellen, selbst bei leistungsstarken Rechnern.

```
pno=> (decimal (factorial (peano 1)))
1
pno=> (decimal (factorial (peano 4)))
24
pno=> (decimal (factorial (peano 5)))
120
```

▷ **Aufgabe 7.3** Die Division ist verwandt zur Multiplikation. Es ist zu zählen, wie oft der Divisor in den Dividenden „hineinpasst“. Definieren Sie die Division `/` und die Modulo-Funktion `mod` für Peano-Ganzzahlen!

7.7 Funktionales Testen

Das Testen ist bei der Entwicklung funktionalen Codes von immenser Bedeutung. Das funktionale Prinzip („Gleicher Input, gleicher Output!“) und die Rekursion machen oft nur wenige Tests erforderlich, um sicher zu gehen, dass sich der Code korrekt und wie erwartet verhält. Die Erstellung von Testfällen ist relativ systematisch und stringent. In imperativen Sprachen ist das allein aufgrund der Seiteneffekte oftmals nicht der Fall.

Zum Beispiel lassen sich die Funktionen `succ` und `pred` mit den folgenden sechs Testfällen zweifelsfrei auf korrekte Arbeitsweise hin überprüfen.

```
(is (= (succ ()) '(succ ())))
(is (= (pred ()) '(pred ())))
(is (= (succ (succ ())) '(succ (succ ()))))
(is (= (pred (pred ())) '(pred (pred ()))))
```

```
(is (= (succ (pred ())) ()))
(is (= (pred (succ ())) ()))
```

Neben den Grundfällen der Konstruktion `(succ ())` und `(pred ())` decken die weiteren Testfälle die vier Kombinationsmöglichkeiten einer unmittelbaren Anwendungsfolge der `succ`- und `pred`-Funktionen ab. Aufgrund des Konstruktionsprinzips der Konstruktoren (ich bin geneigt, es einen „rekursiven Aufstieg“ zu nennen), lassen sich alle weiterreichenden Anwendungsfolgen von `succ` und `pred` auf diese sechs Fälle zurückführen. Sind diese sechs Testfälle erfolgreich, muss notwendigerweise jede beliebige andere Konstruktion einer Peano-Zahl korrekt und richtig sein.

Ähnlich lassen sich die Testfälle z.B. für die Multiplikationsfunktion `*` begründen. Hier reichen zehn Testfälle, um Gewissheit über die korrekte Funktionsweise der Multiplikation mit Peano-Ganzzahlen zu erlangen.

```
(is (== (* (peano 0) (peano 0)) (peano 0)))
(is (== (* (peano 0) (peano 1)) (peano 0)))
(is (== (* (peano 1) (peano 0)) (peano 0)))
(is (== (* (peano 1) (peano 1)) (peano 1)))
(is (== (* (peano 1) (peano 3)) (peano 3)))
(is (== (* (peano 3) (peano 1)) (peano 3)))
(is (== (* (peano 7) (peano 3)) (peano 21)))
(is (== (* (peano 7) (peano -3)) (peano -21)))
(is (== (* (peano -7) (peano 3)) (peano -21)))
(is (== (* (peano -7) (peano -3)) (peano 21)))
```

Bei der Multiplikation nehmen zwei Zahlen eine besondere Rolle ein: die Eins als neutrales und die Null als absorbierendes Element. Der korrekte Umgang mit beiden Zahlen ist zu testen, wobei der Null als Abbruchkriterium unbedingt Beachtung zu schenken ist. Darüber hinaus ist das Kommutativgesetz $a \cdot b = b \cdot a$ zu beachten. Die Multiplikationsfunktion ist nicht symmetrisch definiert. Folglich muss für jedes Paar an Argumenten auch die vertauschte Folge von Argumenten getestet werden. Zu guter Letzt darf ein Test der korrekten Behandlung der Vorzeichen nicht fehlen. Was kann nach diesen Testfällen – mit Blick auf die Implementierung – noch schief gehen?

Lassen Sie das Schreiben von Testfällen zum Reflex werden. Es wird Ihnen Stunden an Fehlersuche ersparen und vor allem das Anpassen und Verändern von Code, modern als Refaktorisierung (*refactoring*) bezeichnet, erleichtern. Sie werden feststellen, dass das Testen in funktionalen Sprachen eher einem Korrektheitsbeweis nahekommt, als denn ein willkürliches Durchtesten von irgendwelchen Fällen ist. Das Kernel-Prinzip sorgt zudem dafür, dass bei Änderungen am Code nicht mehr als die Kernel-Tests nachgezogen werden müssen. Das ist effektives Code- und Testengineering.

7.8 Diskussion

Hätten Sie die Aufgabe gehabt, eine Bibliothek für das Rechnen mit Peano-Zahlen zu entwickeln, ich könnte mir vorstellen, dass Ihr Ergebnis den vorgestellten Funktionen nicht unähnlich wäre. Besonders das `eval` in der Additions- und der Subtraktionsfunktion scheint ein raffinierter Hack zu sein, der Raffinesse im Umgang mit Clojure offenbart. Mit dem Blick eines Software Engineers gefällt mir der bisherige Code jedoch nicht.

7.8.1 Das Kernel-Prinzip

Schauen Sie sich einmal den Code für die Multiplikation an. Dieser Code greift ausschließlich auf die Funktionen `zero?`, `pos?`, `+` und `-` zu und nutzt den Daten-Konvertierer `peano`. Dieser Code nimmt in keinster Weise Bezug auf die Peano-Kodierung. Abgesehen von der Verwendung von `peano` für die Zahlen Null und Eins ist der Code vollkommen generisch und definiert die Multiplikation, ohne von Details über Peano-Zahlen abhängig zu sein. Ähnliche Anmerkungen kann man zur `factorial`-Funktion machen.

Für die Addition und die Subtraktion gilt das nicht mehr. Beide Funktionen nehmen auf die Kodierung der Peano-Zahlen Bezug durch Zerlegung des zweiten Arguments `p2` in `fst` und `snd`. Davon profitiert der `eval`-Hack, der `fst` auf `p1` anwendet.

Bei aller vermeintlichen Eleganz: Ist es so gescheit, auf die Interna der Peano-Kodierung zuzugreifen?

Mit den Funktionen `zero?`, `pos?`, `succ` und `pred` und der Peano-Null `()` ist das Grundgerüst gegeben, auf dem *alle* weiteren arithmetischen Funktionen aufbauen können. Das ist die auf das Wesentliche reduzierte Idee, die hinter dem Aufbau einer Arithmetik mit Peano-Zahlen steht.

Verbergen wir die Konstruktoren `succ` und `pred` hinter dem Interface `inc` und `dec` und nutzen die Typkonverter `peano` und `decimal`, dann ist es darüber hinaus möglich, alle weiteren arithmetischen Funktionen zu formulieren, *ohne* sich mit der internen Kodierung von Peano-Zahlen befassen zu müssen. Die sechs Funktionen `zero?`, `pos?`, `inc`, `dec`, `peano`, `decimal` konstituieren einen Kern an Funktionalitäten, der es erlaubt, alle weiteren Rechenoperationen vollkommen unabhängig von den Details der Implementierung von Peano-Ganzzahlen vorzunehmen.

Aus Sicht der Softwaretechnik ist ein kleiner Kern (*kernel*), in dem sich alle Implementierungsdetails verbergen, in aller Regel als sehr vorteilhaft und erstrebenswert anzusehen. Er entflechtet Abhängigkeiten in der Software, kapselt Implementierungsdetails, was wartungsfreundlich ist, und macht den Aufbau der Software übersichtlich. Wird das Kernel-Prinzip konsequent umgesetzt, kann der Kernel sogar durch eine vollkommen andere Kodierung ersetzt werden, ohne dass weite Teile der Software angepasst werden müssten.

7.8.2 Überarbeitung der arithmetischen Funktionen

Die Funktionen `succ` und `pred` dienen der Konstruktion von Peano-Ganzzahlen entsprechend dem in Kap. 7.2 diskutierten Datenschema. Es ist sozusagen ein Zufall oder – wenn Sie es anders formuliert wissen möchten – eine Besonderheit der Peano-Konstruktion, dass beide Funktionen unmittelbar auf die Inkrementierung bzw. Dekrementierung von Zahlen abbilden.

```
(defn inc [p] (succ p))
(defn dec [p] (pred p))
```

Die Abgrenzung von `inc` zu `succ` (bzw. von `dec` zu `pred`) scheint lapidar zu sein, ist aber äußerst wichtig. Die Konstruktoren `succ` und `pred` gehören zur Schnittstelle, Peano-Zahlen als Datentypen zu erfassen. Die Funktionen `inc` und `dec` sind die gleichnamigen Funktionen aus dem Clojure-Kern und präsentieren sich als Rechenoperationen; hier, im Gegensatz zum Clojure-Kern, als Rechenoperationen für Peano-Zahlen.

Auch wenn `inc` und `dec` wie eine Neuetikettierung von `succ` und `pred` wirken, einem Menschen erzählen die unterschiedlichen Namen etwas von der Semantik, die hinter diesen Funktionen steht: Datenkonstruktoren vs. Rechenoperationen.

Der Aufbau einer Arithmetik mit Peano-Ganzzahlen beruht also neben den Typkonvertern `peano` und `decimal` konzeptuell auf den Funktionen `zero?`, `pos?`, `inc` und `dec` und nicht auf `zero?`, `pos?`, `succ` und `pred`.

▷ **Aufgabe 7.4** Die Funktion `inc` (und entsprechend `dec`) kann kürzer definiert werden mit `(def inc succ)`. Warum ist diese Definition aus softwaretechnischer Sicht abzulehnen?

▷ **Aufgabe 7.5** Definieren Sie das Prädikat `neg?` aus Kap. 7.3 ausschließlich unter Verwendung von `zero?` und `pos?`.

Unter der Maßgabe, alle Abhängigkeiten von der Peano-Kodierung aufzulösen und nur auf den Funktionen des Kernels aufzubauen, zeigt sich die Additionsfunktion in einem neuen Gewand. Der `eval`-Hack verbietet sich durch diese Einschränkungen.

```
(defn + [p1 p2]
  (cond (zero? p2) p1
        (pos?  p2) (recur (inc p1) (dec p2))
        :else      (recur (dec p1) (inc p2))))
```

Die Additionsfunktion ist deutlich klarer zu lesen und zu verstehen. Sie ist präziser und eindrücklicher nicht mehr zu fassen. Die Abhängigkeiten von der Peano-Kodierung entfallen vollständig. Die Symmetrie mit der Subtraktionsfunktion wird offensichtlich.

```
(defn - [p1 p2]
  (cond (zero? p2) p1
        (pos? p2)  (recur (dec p1) (dec p2))
        :else      (recur (inc p1) (inc p2))))
```

Auch die Multiplikationsfunktion sei diesem Schema angepasst. Dem Augenschein nach handelt es sich nur um eine optische „Anpassung“. Softwaretechnisch ist aber auch wichtig, dass der Code entsprechend den Funktionen `+` und `-` einem sich wiederholenden Muster folgt. Programmcode wird für Menschen geschrieben! Das `cond`-Muster mit den stereotypen Fallunterscheidungen von `zero?`, `pos?` und `:else` macht den Code verständlicher und wartbarer, die innere Ablauflogik und Funktionsweise wird „offensichtlich“.

```
(defn *
  ([p1 p2] (* p1 p2 (peano 0)))
  ([p1 p2 res]
    (cond (zero? p2) res
          (pos? p2)  (recur p1 (dec p2) (+ p1 res))
          :else      (recur p1 (inc p2) (- res p1)))))
```

Ob man in der Fakultätsfunktion ein `(dec p)` verwendet statt einer ausdrücklichen Subtraktion von Peano-Eins, ist Geschmackssache.

▷ **Aufgabe 7.6** Befreien Sie die Funktion `decimal` ebenso von einem unnötigen Wissen um die Kodierung von Peano-Ganzzahlen. Greifen Sie auf das `cond`-Muster zurück.

▷ **Aufgabe 7.7** Aus der Sicht des Kernel-Prinzips könnte man dafür argumentieren, das Prädikat `is-peano?` wie folgt zu implementieren:

```
(defn- is-peano? [p]
  (or (zero? p) (pos? p) (neg? p)))
```

Was halten Sie davon?

▷ **Aufgabe 7.8** Der überarbeitete Kernel soll an eine neue Kodierung angepasst werden. Die Peano-Null sei als leerer Vektor `[]` kodiert, Nachfolger bzw. Vorgänger über einen Vektor mit `succ`- und `pred`-Symbolen. So stellt sich Zahl Eins dar als `['succ]`, die Zahl −2 als `['pred 'pred]`.

7.8.3 Kernel-Prinzip vs. `eval`-Hack

Diese „Umbauten“ bringen einen interessanten Effekt mit sich: Das Rechnen mit Peano-Zahlen ist deutlich schneller geworden. Das Kernel-Prinzip erzwingt den Abschied vom `eval`-Hack, der Clojure viel Zeit kostet. Der klar strukturierte, aufeinander aufbauende Code tut auch dem Clojure-Compiler gut. Auf meinem Rechner benötigt die Fakultätsfunktion mit der „alten“ Additionsfunktion für die Fakultät von `(peano 5)` um die 1700 ms, die Fakultät von `(peano 6)` gar 120 Sekunden, sprich zwei Minuten. Mit `time` können Sie so etwas selber messen. Dieselben Rechnungen mit der „neuen“ Additionsfunktion benötigen nur 6 ms und 132 ms. Das sind Verbesserungen um den Faktor 280 und 900!

Wie Sie gesehen haben, gibt es in der funktionalen Programmierung einen Weg der Modellierung, der sich von der imperativer Sprachen – insbesondere objekt-orientierter Sprachen – unterscheidet. Man strebt an, sich möglichst zügig von der internen Datenrepräsentation zu lösen. Das erreicht man durch das Anlegen eines Kerns, in dem sich alle Details der Datenkodierung verbergen. Weitere Funktionen greifen – wenn nötig – nur auf Funktionen des Kerns zurück, niemals auf Details der Datenkodierung.

Der funktionale Weg bringt schon im Kleinen Möglichkeiten der Modularisierung und Kapselung mit, wie er für imperative Sprachen unüblich ist. Nicht ohne Grund skalieren funktionale Programme in der Regel weitaus besser als ihre imperativen Gegenstücke.

7.9 Lösungen

Aufgabe 7.1 Der Aufruf `(zero? nil)` liefert `true` zurück, was nicht der vereinbarten Kodierung für eine Peano-Null entspricht.

Aufgabe 7.2 Man kann bei der rekursiven Definition von `peano?` auf `is-peano?` zurück greifen, um eine wiederholte Programmierung zur Prüfung der Kodierungsvorschrift zu vermeiden. In der Regel ist ein Verzicht auf Redundanz guter Programmierstil.

```
(defn peano? [p]
  (or (zero? p) (and (is-peano? p) (recur (second p)))))
```

▷ **Aufgabe 7.9** Nebenbei gefragt: Warum kann in einem `and` Endrekursion zum Einsatz kommen?

Man kann gegen die Lösung einwenden, dass ihre Arbeitsweise nicht auf einen Blick erfassbar ist. Eine alternative Lösung ist:

```
(defn peano? [p]
  (cond (zero? p) true
        (is-peano? p) (recur (second p))
        :else false))
```

Beide Lösungen lassen Listenkodierungen mit mehr als zwei Elementen zu. So wird z.B. `(peano? '(succ () "hey"))` als gültige Peano-Zahl erkannt. Zusätzliche Listenelemente „stören“ als Zusatzinformation beim Rechnen mit Peano-Zahlen nicht. In einer statisch typisierten Sprache wären solche „unsauberen“ Kodierungen untragbar, in einer dynamisch typisierten Sprache gehört es zum Charme, solche Freiheiten zu lassen – ganz risikolos sind solche Freiheiten nicht. Möchte man dieses Risiko eindämmen, ist `peano?` mit einer Vorbedingung zu versehen, die eine leere oder zweielementige Liste erwartet.

```
(defn peano? [p]
  {:pre [(or (zero? p) (clojure.core/== (count p) 2))]}
  (or (zero? p) (and (is-peano? p) (recur (second p)))))
```

Aufgabe 7.3 Die Lösung ist nicht ganz einfach. Versuchen Sie es selbst!

Aufgabe 7.4 Die Definition erklärt `inc` zum Alias von `succ` – und verrät nichts darüber, wieviele Argumente `inc` erwartet. Dazu muss die Definition von `succ` zurate gezogen werden. Die Absicht ist jedoch, Anwendungsentwickler von diesen Implementierungsdetails fern zu halten. Ein weiterer Punkt: Der Aufruf `(doc inc)` sagte ebenfalls nichts über die Argumente aus und lässt Anwendungsentwickler ratlos zurück.

Aufgabe 7.5 Das `neg?`-Prädikat ist der Ausschluß des Nullfalls und des Positivfalls.

```
(defn neg? [p] (not (or (zero? p) (pos? p))))
```

Diese Logik greift auch in z.B. der überarbeiteten Additionsfunktion: Mit `:else` wird der Negativfall durch Ausschluß abgedeckt.

Aufgabe 7.6 Die Lösung kommt ohne internes Wissen um die Kodierung von Peano-Zahlen aus.

```
(defn decimal
  ([p] (decimal p 0))
  ([p n] {:pre [(is-peano? p)]}
    (cond
      (neg? p) (recur (inc p) (clojure.core/dec n))
```

```
(pos? p) (recur (dec p) (clojure.core/inc n))
:else n)))
```

Aufgabe 7.7 Das ist keine gute Idee und zielt am Kernel-Prinzip vorbei. Die beiden Prädikate `zero?` und `is-peano?` definieren ein für sich stehendes Kodierschema – mit Funktionen wie `pos?` oder `neg?` hat das nichts zu tun. Es ist vielmehr so, dass diese Funktionen erst nach der Einführung des Kodierschemas und aufgrund der Kenntnis des Kodierschemas definiert werden können. Erst das Schema, dann die Prädikate, die das Schema interpretieren. Eine Vermischung von Interpretationen des Schemas und seiner Definition stiftet nur Verwirrung und verstellt den Blick, was wovon abhängt.

Aufgabe 7.8 Zu dieser Aufgabe gibt es keine Musterlösung. Entscheidend an dieser Aufgabe ist, dass Sie merken, wie leicht sich eine Änderung des Kernels umsetzen lässt ohne die übrigen arithmetischen Funktionen ändern zu müssen.

Aufgabe 7.9 `and` ist keine Funktion, sondern ein Makro. Der zweite Ausdruck in `and` wird nur ausgewertet, wenn der erste weder `false` noch `nil` liefert.

Historie

22. Mai 2012	Lösungskapitel erscheint im Inhaltsverzeichnis
28. Nov. 2011	Kleinere Schreibkorrekturen.
28. Nov. 2011	Lösungen zu den Aufgaben hinzugefügt.
28. Nov. 2011	Mit Dank an Marc Hesenius für die Korrekturen und Vorschläge. Darunter: `is_peano?` in `is-peano?` geändert.
16. Mai 2011	Neue Aufgabe (Vektor zur Kodierung von Peano-Zahlen) hinzugefügt.
12. Mai 2011	Neue Aufgabe (Makro für `peano`) hinzugefügt.
12. Mai 2011	Update: „Kommutativgesetz“ falsch geschrieben, Dank an Peter Schiffmann; „wie erwartet“ falsch geschrieben, Dank an Patrick Kubach
11. Mai 2011	Update: In der Funktion `decimal` fehlen die Bezüge auf `clojure.core`. Dank an Markus Remensberger!
11. Mai 2011	Erste Version.

8 Fallstudie: Parserkombinatoren

8.1 Parserbau einfach gemacht

Der Compilerbau ist ein Thema, das die Informatik seit ihren Anfangstagen beschäftigt: Wie lässt sich der Programmcode in einer Textdatei übersetzen in den Maschinen- oder Bytecode eines Prozessors oder einer virtuellen Maschine?

In dieser Fallstudie beschäftigen wir uns mit dem Parsen (*parsing*). Ein Compiler besteht – grob gesagt – aus einem Parser und einem Codegenerator. Der Parser wandelt einen eingehenden Zeichenstrom um in eine strukturierte Datenrepräsentation. Am Ende des Verarbeitungsprozesses steht in der Regel der sogenannte Abstrakte Syntaxbaum (AST, *Abstract Syntax Tree*); allgemeiner spricht man auch von einem Parsebaum (*parse tree*). Der Codegenerator beginnt seine Arbeit mit dem Parsebaum und erzeugt daraus Maschinen- oder Bytecode. Auch Interpreter arbeiten oft direkt mit dem Parsebaum.

Jedes Computerprogramm muss nach den Regeln einer Grammatik geschrieben werden. Die Grammatikregeln für eine Programmiersprache legen fest, wann ein Programm als syntaktisch korrekt gilt. Das meint: Ist die Folge von Zeichen, die den Programmcode ausmachen, eine gültige Abfolge von Zeichen oder nicht? Zum Beispiel erkennen Sie die Zeichenfolge „`2 + 3`“ als gültigen arithmetischen Rechenausdruck; die Regeln dazu hat man Ihnen in der Schule beigebracht, sie sind Ihnen intuitiv zugänglich. Anhand dieses Wissens können Sie ebenso entscheiden, dass „`I am!`“ kein arithmetischer Ausdruck ist.

Grammatiken für formale, sprich Computersprachen, werden üblicherweise in der Backus-Naur-Form (BNF) oder der Erweiterten Backus-Naur-Form (EBNF) oder in verwandten Formen notiert. Mit Hilfe von Werkzeugen lässt sich aus einer Grammatik-Beschreibung vollautomatisch ein Parser für die Sprache generieren. Ein sehr mächtiges solches Werkzeug ist ANTLR (http://www.antlr.org). Die meisten Informatiker haben auch schon einmal etwas von den Werkzeugen „Lex“ und „Yacc“ gehört. Für eine dem Parsen vorangeschaltete lexikalische Analyse dient „Lex“. Und mit „Yacc“ (*Yet Another Compiler Compiler*) kann ein Parser und Codegenerator erzeugt werden.

Man kann sich einen Parser auch ohne Hilfe von Werzeugen selber programmieren. Besonders einfach geht das mit sogenannten Parserkombinatoren. Parserkombinatoren werden in der Literatur auch als Erkenner (*recognizer*) klassifiziert [GJ08]. Mit ihnen erstellt man direkt ein Programm, das einen Text gemäß der Kombination von Parsern erkennt. Statt aus einer Grammatik einen Parser zu erzeugen, „steckt“ man Parser mittels Parserkombinatoren zu neuen Parsern zusammen.

8.2 Die Parser `item`, `success` und `fail`

Jeder der nachfolgenden Parser nimmt als Input `in` eine Sequenz von Daten entgegen. Normalerweise verarbeiten Parser Zeichenketten. Wir

verallgemeinern das auf „Datenketten“, was uns von der Beschränkung auf Zeichenketten löst.

Unsere Parser liefern immer zwei Werte in einem Vektor zurück. Der erste Wert liefert das Ergebnis des Parsens zurück, der zweite den Rest, der von der Eingangssequenz übrig bleibt. Schlägt das Parsen fehl, ist der erste Wert immer `nil` und der zweite Wert die unveränderte Eingangssequenz `in`.

Die Funktion `item` realisiert einen solchen Parser. Man kann sie mit einem Datenwert `itm` konfigurieren und erhält eine anonyme Funktion zurück, die einen Parser für `itm` umsetzt.

```
(defn item [itm]
  (fn [in]
    (if (empty? in)
      [nil in]
      (if (= (first in) itm)
        [[itm] (rest in)]
        [nil in]))))
```

Liegen keine Daten mehr in der Sequenz vor, `(empty? in)`, dann wird `[nil in]` zurückgegeben. Ansonsten wird getestet, ob das erste Element `(first in)` in der Sequenz gleich ist mit `itm`. Ist das der Fall, wird das erkannte Item in einem Vektor verpackt zusammen mit dem Rest der Sequenz zurückgegeben. Ist `itm` nicht das erste Element von `in`, so ist `[nil in]` das Ergebnis.

Ganz deutlich wird das Verhalten in der Interaktion über die REPL. Zum Beispiel erzeugen wir einen Parser für eine `2` mit

```
user=> (item 2)
#<user$item$fn__32 user$item$fn__32@66bcb23e>
```

Dieser Parser erkennt, ob in einer beliebigen Datenfolge das erste Element eine `2` ist. Wir spielen an der Konsole die drei Fälle durch: (1) erfolgreich, (2) nicht erfolgreich, da Sequenz leer, (3) nicht erfolgreich, da erstes Element keine `2` ist.

```
user=> ((item 2) [2 3 4])
[[2] (3 4)]
user=> ((item 2) [1 2 3 4])
[nil [1 2 3 4]]
user=> ((item 2) [])
[nil []]
```

Da wir später davon noch Gebrauch machen möchten, seien hier zwei weitere, sehr einfache Parser definiert.

```
(defn success [in] [[] in])
(defn fail [in] [nil in])
```

Der `success`-Parser verarbeitet zwar die `in`-Sequenz nicht, liefert jedoch kein `nil`, sondern einen leeren Vektor als Parse-Ergebnis zurück. Das markiert einen erfolgreichen Parsevorgang, auch wenn keine Daten vom Eingabegestrom berücksichtigt werden. Der `fail`-Parser meldet immer einen nicht erfolgreichen Parsevorgang. Man kann mit dem `fail`-Parser (siehe z.B. im obigen Code für `item`) die Rückgaben `[nil in]` etwas aussagekräftiger mit `(fail in)` ersetzen.

8.3 Der Alternativkombinator `or`

Ein Parserkombinator nimmt eine Anzahl von Parsern als Argument entgegen und liefert seinerseits einen Parser zurück, der die Parserkombination realisiert. Ein interessanter Parserkombinator ist der Alternativ-Kombinator, der einen Alternativ-Parser realisiert: Der erste Parser wird angewendet. Bei Misserfolg wird der zweite Parser angewendet. Bei wiederholtem Misserfolg der dritte Parser usw. Entweder ist ein Parser erfolgreich oder ein Misserfolg beendet die Suche nach einem „passenden“ Parser.

Wir nennen diesen Alternativ-Parser auch einen „Oder“-Parser und verwenden für den Parserkombinator `or`. Um nicht in Konflikt zu geraten mit dem `or` aus dem Clojure-Kern, müssen wir einen eigenen Namensraum aufmachen und die Übernahme von `or` und einiger weiterer Symbole aus dem Clojure-Kern verbieten; wir werden noch `and`, `not` und `drop` definieren. Ich gehe davon aus, dass Sie den Code zu den Parserkombinatoren in einer Datei namens `pc.clj` abspeichern und den Beginn der Clojure-Datei mit der folgenden Deklaration des Namensraums `pc` versehen. Wir haben ähnliches schon in der Fallstudie zu den Peano-Zahlen gemacht.

```
(ns pc
  #^{:doc "Parser Combinators"
     :author "Dominikus Herzberg, Heilbronn University"}
  (:refer-clojure :exclude (and or not drop))
  (:use clojure.test))
```

Zur Auffrischung: Mit `(load "pc")` laden Sie den Programmcode aus der Datei, mit `(in-ns 'pc)` wechseln Sie in den Namensraum `pc`.

Der folgende Code für den `or`-Parserkombinator liest sich wie eine formalisierte Version der freisprachlichen Beschreibung. Dass Beschreibung und Code so nah beieinander liegen, ist ein typisches Merkmal funktionaler Programme.

```
(defn or [& parsers]
  (fn [in]
    (loop [ps parsers]
      (if (empty? ps)
        [nil in]
        (let [[res1 in1] ((first ps) in)]
          (if res1 [res1 in1] (recur (rest ps))))))))
```

Die Funktion `or` nimmt eine beliebige Anzahl an Parsern als Argument entgegen und legt sie in der Liste `parsers` ab. Die Funktion `or` gibt mit der anonymen Funktion einen Parser zurück, der mit `loop` die Parser nacheinander durchgeht. Der erste Parser in der Liste wird probehalber auf `in` angewendet und das Ergebnispaar an `res1` und `in1` gebunden. Hat `res1` den Wert `nil`, so geht `loop` mit `recur` und der verminderten Parserliste `(rest ps)` in die nächste Runde. Beachten Sie, dass `nil` und `false` die einzigen beiden Werte sind, die bei einem `if` zur Auswertung des *else*-Teils, also des zweiten Ausdrucks führt. Alle anderen Werte werden wie ein logisches `true` behandelt. Sollte `res1` in der `if`-Form einen Erfolg vermelden (also nicht `nil` sein), ist die Rückgabe `[res1 in1]` das Parseergebnis des erfolgreichen Parsers.

Schauen wir uns den `or`-Kombinator in Aktion an. Angenommen, wir benötigen einen Parser, der entweder das Item `2` *oder* das Item `3` am Anfang eines Eingangsstroms erkennt.

```
pc=> ((or (item 2) (item 3)) [1 2 3])
[nil [1 2 3]]
pc=> ((or (item 2) (item 3)) [2 3])
[[2] (3)]
pc=> ((or (item 2) (item 3)) [3 2 1])
[[3] (2 1)]
pc=> ((or (item 2) (item 3)) [])
[nil []]
```

Der Parserkombinator arbeitet wie gewünscht – und man könnte damit zufrieden sein. Der Programmcode zur Funktion `or` kann erheblich verkürzt werden – wenn man sich einlässt auf die Arbeit mit verzögerten Sequenzen (*lazy sequences*) und Funktionen höherer Ordnung (*Higher-Order Functions*, HOF). In der Zusammenarbeit sind beide Stilmittel sehr ausdrucksstark und typisch für die funktionale Denke. Gerade wenn Ihnen die verzögerten Sequenzen nicht ganz geheuer sind, sollten Sie sich unbedingt in dieser Fallstudie mit Ihnen auseinander setzen. Die Perspektive der Datenströme ist äußerst produktiv und beliebt in der funktionalen Welt.

Machen wir es konkret. Der bisherige Code zum `or`-Parserkombinator geht die Parser per `loop` durch und wendet einen Parser nach dem anderen per `first` auf `in` an. Exakt dies lässt sich mit Hilfe der HOF `map` kompakt ausdrücken: `map` wendet eine Funktion auf alle Elemente einer Sequenz an und stellt die Ergebnisse in einer – und das ist das entscheidende – verzögerten(!) Sequenz zusammen. Das Programmfragment unter Verwendung von `map` sieht wie folgt aus.

```
(map #(% in) parsers)
```

Erinnern Sie sich: `#(` aktiviert ein Reader-Makro. Es ist eine Kurznotation für eine anonyme Funktion mit `%` als Argument.

Schauen wir uns die Funktionsweise von `map` am Beispiel an:

```
pc=> (map #(% [3 2 1]) [(item 2) (item 3)])
([nil [3 2 1]] [[3] (2 1)])
```

An der Konsole sieht es so aus, als würde `map` eine Sequenz mit allen Anwendungsergebnissen liefern. Tatsächlich handelt es sich um eine verzögerte Sequenz – eine Auskunft, die Ihnen auch `(doc map)` gibt. Setzen Sie ein `type` um den vorigen Ausdruck.

```
pc=> (type (map #(% [3 2 1]) [(item 2) (item 3)]))
clojure.lang.LazySeq
```

Das heißt: `map` produziert die Elemente der verzögerten Ergebnissequenz einen nach dem anderen, jedoch nur auf Aufforderung. Darum leistet `map` nicht mehr Arbeit als nötig, wendet also die Funktion nicht zwangsläufig auf alle Elemente der Eingangssequenz an.

Bislang liefert uns `map` die Ergebnisse der reihenweisen Anwendung der Parser zurück. Interessiert sind wir nur an den Ergebnissen erfolgreicher Parser. Also filtern wir mit `filter` all die Vektoren heraus, deren erstes Element nicht `nil` ist. Da `nil` wie ein logisches `false` behandelt wird und alle anderen Werte (außer selbstverständlich `false` selber) wie ein logisches `true` fungieren, muss die beizusteuernde Prädikatsfunktion lediglich das erste Elemente aus dem Parseergebnis extrahieren.

```
pc=> (filter #(first %) (map #(% [3 2 1]) [(item 2) (item 3)]))
```



```
([[3] (2 1)])
```

Beachten Sie auch hier, dass `filter` eine verzögerte Sequenz zum Ergebnis hat. Mit jeder Anforderung wird `filter` nur soviele Sequenzelemente von `map` abrufen, bis das Prädikat zu `filter` einen Erfolg meldet. Mit den verzögerten Sequenzen wird die Rechenarbeit kaskadenartig eingefordert; niemals jedoch werden mehr Elemente einer verzögerten Sequenz betrachtet als absolut notwendig.

Wenn wir so weit mit `filter` und `map` gekommen sind, benötigt es nur noch einer einfachen Abfrage: Konnte überhaupt ein erfolgreich angewendeter Parser gefunden werden? Falls nicht, muss das Resultat `[nil in]` oder alternativ `(fail in)` sein. Gibt es jedoch ein erfolgreiches Parseergebnis, so wird dies mittels `first` aus der verzögerten Sequenz entnommen.

Hier der vollständige, neue Code zum `or`-Parserkombinator.

```
(defn or [& parsers]
  (fn [in]
    (let [res (filter #(first %) (map #(% in) parsers))]
      (if (empty? res) [nil in] (first res)))))
```

Sie können sich mit einem kleinen „Trick“ davon überzeugen, dass diese `or`-Version wirklich *lazy* arbeitet und nicht jeden Parser stumpf auf `in` anwendet. Sobald ein erfolgreicher Parser gefunden ist, ist `or` fertig, egal welche Parser dem Erfolgsparser folgen. Fügen wir also zu „Testzwecken“ einen Parser mit einem unangenehmen Seiteneffekt hinzu; der Parser erzeugt mittels `(assert false)` eine Fehlermeldung.

```
pc=> ((or (item 2) (item 3) (fn [in] (assert false))) [2 1])
[[2] (1)]
pc=> ((or (item 2) (item 3) (fn [in] (assert false))) [3 1])
[[3] (1)]
pc=> ((or (item 2) (item 3) (fn [in] (assert false))) [4 1])
java.lang.AssertionError: Assert failed: false (NO_SOURCE_FILE:0)
```

Würde der Parser des `or`-Kombinators jeden Parser auf den Input `in` anwenden, dann müsste schon in den ersten beiden Fällen ein Zusicherungsfehler (*assertion error*) auftreten. Dank verzögerten Sequenzen passiert das nicht.

Der überarbeitete Code für den `or`-Parserkombinator ist deutlich kürzer und kompakter als die „alte“ Fassung mit `loop`. Der Preis ist ein etwas höherer Aufwand in der intellektuellen Durchdringung des Codes. Das ist genau die Abwägung, die funktionale Programmierer(innen) immer wieder vornehmen müssen: Will ich kürzeren Code und damit mehr Code in der Gesamtschau intellektuell überblicken können? Oder liegt mir an der Verständlichkeit einer Einzelfunktion, benötige an der Stelle also mehr Codezeilen und nehme dafür einen Verlust an Übersicht in Kauf?

Es gibt dazu keine grundsätzlich richtige Antwort. Funktionale Programmierer(innen) tendieren zur Devise „In der Kürze liegt die Würze“. Das Denken in verzögerten Sequenzen und der Einsatz von Funktionen höherer Ordnung ist mehr Übungssache als denn eine unüberwindbare intellektuelle Barriere. Funktionen höherer Ordnung wie z.B. `map` und `filter` sind Abstraktionen von Ablaufmustern, was hilft, Gemeinsamkeiten der Abarbeitung in Funktionsrümpfen deutlich besser zu erkennen. Es liegt damit ein höheres Augenmerk auf die Wiederverwendbarkeit von Codefragmenten. Funktionale Programmierer(innen) entdecken auf diese Weise oft wie

von selbst Wiederholungsmuster oder gar *Design Patterns* (Entwurfsmuster) in ihrem Code. Das wiederum mag Anlaß zu neuen Abstraktionen geben, die den Code weiter verdichten und auf das Wesentliche reduzieren.

8.4 Die Sequenzkombinatoren `and` und `and-lazy`

Eine weitere Klasse von Parserkombinatoren betrifft die sequentielle Kombination von Parsern. Mit der Hilfsfunktion `pipe` beschreiben wir das generelle Vorgehen: Ein Parser nach dem anderen wird auf die sich mit jeder Anwendung gleich bleibenden oder „verkürzenden“ Eingangsequenz angewendet. Jeder Folgeparser erhält die Sequenz zur Eingabe, die sein Vorgänger als Ausgabe hat – ungeachtet dessen, ob der Parsevorgang des Vorgängers erfolgreich ist oder nicht. Natürlich kommt hier wieder eine verzögerte Sequenz zum Einsatz, die ein Parseergebnis nach dem anderen liefert – aber eben nicht mehr berechnet, als angefordert wird.

```
(defn- pipe [parsers in]
  (lazy-seq
    (if (empty? parsers)
      ()
      (let [[res1 in1 :as res] ((first parsers) in)]
        (cons res (pipe (rest parsers) in1))))))
```

Der `and`-Parserkombinator erzeugt aus einer Folge von Parsern einen neuen Parser, der erst den ersten Parser auf den Eingangsstrom anwendet *und* dann den zweiten Parser auf den (in der Regel) reduzierten Ausgabestroms des ersten Parsers anwendet *und* dann den dritten Parser auf den Ausgabestrom des zweiten Parsers anwendet usw. Der `and`-Parser ist erfolgreich, wenn jeder Einzelparser erfolgreich ist. Mit Hilfe der `pipe`-Funktion und der Funktion `every?` ist der `and`-Parserkombinator einfach zu beschreiben. Die Funktion `every?` überprüft, ob alle Elemente einer Sequenz die Bedingung eines übergebenen Prädikats erfüllen.

```
(defn and [& parsers]
  (fn [in]
    (let [res (pipe parsers in)]
      (if (every? #(first %) res)
        [(reduce concat (map first res)) (second (last res))]
        [nil in]))))
```

Das Ergebnis des `and`-Parsers (die anonyme Funktion, die der `and`-Parserkombinator zurück liefert), ist – bei Erfolg – ein Vektor, dessen erstes Element alle Sequenzen bzw. Vektoren der Einzelergebnisse `(map first res)` konkateniert. Das zweite Element ist der übrig gebliebene Eingangsstrom aus der letzten Parseranwendung `(second (last res))`. Im Misserfolgsfall kommt, wie gehabt, `[nil in]` zurück.

Wenn man es etwas anschaulich formulieren möchte: Jeder Einzelparser „knabbert“ etwas von der Eingangssequenz weg, sofern alle Einzelparser erfolgreich sind.

```
pc=> ((and (item 2) (item 3)) [1 2 3])
[nil [1 2 3]]
pc=> ((and (item 2) (item 3)) [2 3 1])
[(2 3) (1)]
pc=> ((and (item 2) (item 3)) [2 1 3])
[nil [2 1 3]]
```


Ein mit `and` verwandter Parserkombinator, den wir `and-lazy` nennen wollen, versucht, so viele Parser erfolgreich nacheinander anzuwenden, wie möglich. Er gibt beim ersten Fehlschlag das bis dahin erfolgreich geparste Ergebnis zurück. Im Gegensatz zum `and`-Parser müssen nicht alle an den `and-lazy`-Kombinator übergebenen Parser erfolgreich sein, sondern nur so viele wie möglich. Mit `take-while` steht eine HOF zur Verfügung, die beim Einsammeln der Ergebnisse bis zum ersten Fehlschlag bezüglich eines Prädikats hilft.

```
(defn and-lazy [& parsers]
  (fn [in]
    (let [res (take-while #(first %) (pipe parsers in))]
      (if (empty? res)
        [nil in]
        [(reduce concat (map first res)) (second (last res))]))))
```

Da bereits der erste übergebene Parser nicht erfolgreich sein kann, muss überprüft werden, ob `res` leer ist, um `[nil in]` zu liefern. Ansonsten werden wie beim `and`-Parserkombinator die gefundenen Ergebnisse konkateniert und der Rest-Eingabestrom zurück geliefert.

```
pc=> ((and-lazy (item 2) (item 3)) [1 2 3])
[nil [1 2 3]]
pc=> ((and-lazy (item 2) (item 3)) [2 3 1])
[(2 3) (1)]
pc=> ((and-lazy (item 2) (item 3)) [2 1 3])
[[2] (1 3)]
```

8.5 Für den Komfort: `optional`, `more` und `many`

Allein mit dem `item`-Parser und den Parserkombinatoren `or`, `and` und `and-lazy` lassen sich bereits aufwendige Grammatiken abbilden. Um den Komfort im Umgang mit den Parserkombinatoren zu erhöhen, definieren wir noch zwei weitere, aus `or` und `and-lazy` abgeleitete Parser.

```
(defn optional [parser] (or parser success)) ; 1..0
(defn more [parser] (apply and-lazy (cycle [parser]))) ; *..1
```

Mit `optional` wird ein Parser versucht, einmal erfolgreich auf die Eingangssequenz anzuwenden. Sollte das misslingen, ist auch die „keinmalige“ Anwendung des Parser als Erfolg zu werten. Diesem Vorgehen entspricht die `or`-Kombination des übergebenen Parsers mit dem `success`-Parser.

Mit `more` wird ein Parser versucht so oft wie möglich erfolgreich auf den Eingabestrom anzuwenden, mindestens jedoch einmal. Mit `cycle` wird der an `more` übergebene Parser beliebig oft repliziert und `and-lazy` erledigt die maximal mögliche Anwendung der übergebenen Parserfolge. Hier muss die Funktion `apply` zum Einsatz kommen, da `and-lazy` mit beliebig vielen Argumenten und nicht mit einer Sequenz aufgerufen wird, was `apply` entsprechend umsetzt.

▷ **Aufgabe 8.1** Definieren Sie einen `many`-Parser, der einen Parser so oft wie möglich erfolgreich anzuwenden versucht, aber auch den Misserfolg „keinmaliger“ Anwendung akzeptiert.

8.6 Notation und Umsetzung von Grammatikregeln

Zur übersichtlichen Darstellung von Grammatiken, die wir mit Parsern und Parserkombinatoren abbilden, führen wir eine kompakte Notation ein. Eine Grammatik soll beschreibbar sein unabhängig davon, ob wir zu ihrer Umsetzung Clojure oder eine andere Programmiersprache verwenden. Wir vereinbaren die folgenden Konventionen.

Ein Parser (besser: Erkenner) für eine Grammatik setzt sich dank Parserkombinatoren aus Teilparsern zusammen. Meist ist es hilfreich, sich auf die Teilparser per Namen zu beziehen. Nicht zuletzt wird eine Grammatik dadurch übersichtlicher organisiert. Wir setzen die Namen von Parsern in spitze Klammern, wie z.B. den Parser ⟨*integer*⟩.

Definiert wird ein solcher namentlicher Parser entweder über einen `item`-Parser oder einen Parser, der aus anderen Parsern kombiniert ist. Die `and`-Kombination wird schlicht durch eine Folge von Parsern ausgedrückt, die per Leerzeichen getrennt sind. Zum Beispiel ist der ⟨*integer*⟩-Parser definiert durch einen Parser namens ⟨*sign*⟩? `and` einen Parser namens ⟨*dig1-9*⟩ `and` einen Parser ⟨*dig0-9*⟩*.

⟨*integer*⟩ := ⟨*sign*⟩? ⟨*dig1-9*⟩ ⟨*dig0-9*⟩∗

Der ⟨*sign*⟩-Parser ist `optional`, was sich ausdrückt durch das angehängte Fragezeichen. Der Stern steht für `many` und bezieht sich ebenso auf den unmittelbar vor ihm stehenden Parser ⟨*dig0-9*⟩. In Clojure übersetzt lautet der Ausdruck:

```
(def integer (and (optional sign) dig1-9 (many dig0-9)))
```

Ein `more` wird durch ein Pluszeichen „+“ notiert und steht wie das Fragezeichen bzw. der Stern in nachgestellter, so genannter Postfix-Position, und bezieht sich direkt auf den voranstehenden Parser. Soll sich ein *many*, *more* oder *optional* auf einen umfassenderen Ausdruck beziehen, so sind Klammern zu verwenden. Zum Beispiel meint (⟨*data*⟩ ⟨*space*⟩)?, dass die Folge der Parser ⟨*date*⟩ und ⟨*space*⟩ optional ist.

Die `or`-Kombination wird durch den Strich „|“ notiert. Zum Beispiel ist ⟨*sign*⟩ definiert als ein '`+`' *oder* ein '`-`'.

⟨*sign*⟩ := '`+`' | '`-`'

Die Hochkommata markieren Einzelzeichen bzw. Einzelzeichenfolgen und werden auch Terminalsymbole (*terminal symbols*) bzw. Token (*token*) genannt. Dem entspricht ein `item`-Parser, der ein `\+` bzw. `\-` erkennt; in Clojure kodiert der umgekehrte Schrägstrich ein Einzelzeichen. Der obige Ausdruck für ⟨*sign*⟩ übersetzt sich mit Clojure zu:

```
(def sign (or (item \+) (item \-)))
```

Der Parser ⟨*dig1-9*⟩ ist – Sie werden es sicher vermuten – definiert über

⟨*dig1-9*⟩ := '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

Eine ausschließliche Folge von Terminalsymbolen kann mit Hilfe von `map` und `item` angenehm verkürzt an `or` übergeben werden. In Clojure ergibt sich:

```
(def dig1-9 (apply or (map item [\1 \2 \3 \4 \5 \6 \7 \8 \9])))
```

▷ **Aufgabe 8.2** Übersetzen Sie ⟨*dig0-9*⟩ := '`0`' | ⟨*dig1-9*⟩ in einen Clojure-Ausdruck.

Wenn Sie einen Zeichenstrom vom ⟨*integer*⟩-Parser bearbeiten lassen wollen, müssen Sie darauf achten, die Teilparser in einer geeigneten Reihenfolge über die REPL oder in einer Datei einzugeben. Der ⟨*integer*⟩-Parser kann nicht definiert werden, bevor die Parser ⟨*sign*⟩, ⟨*dig1-9*⟩ und ⟨*dig0-9*⟩ definiert sind; und ⟨*dig0-9*⟩ erfordert die vorhergehende Definition von ⟨*dig1-9*⟩.

Ein paar Beispiele der Interaktion über die Konsole zeigen, wie ⟨*integer*⟩ gemäß seiner Definition arbeitet. Zeichenketten wie z.B. `"123"` müssen per `seq` in eine Sequenz aus Einzelzeichen umgewandelt werden, da unsere Parser nicht angepasst sind auf die Verarbeitung von Zeichenketten, sondern allgemein mit Wertfolgen beliebiger Datentypen umgehen können.

```
pc=> (integer (seq "123"))
[(\1 \2 \3) ()]
pc=> (integer (seq "0123"))
[nil (\0 \1 \2 \3)]
pc=> (integer (seq "+0123"))
[nil (\+ \0 \1 \2 \3)]
pc=> (integer (seq "-90"))
[(\- \9 \0) ()]
pc=> (integer (seq "0"))
[nil (\0)]
```

Wie Sie sehen, akzeptiert die Grammatik für ⟨*integer*⟩ keine führenden Nullen, was beabsichtigt ist. Unglücklicherweise wird aber auch eine alleinstehende Null nicht akzeptiert. Ein Mangel, der mit der folgenden Aufgabe behoben werden soll.

▷ **Aufgabe 8.3** Verändern Sie die Grammatik für ⟨*integer*⟩ so, dass auch eine alleinstehende Null gültig ist. Verwenden Sie so wenige Terminalsymbole (sprich `item`-Parser) wie möglich.

Die Grammatik für ⟨*integer*⟩ kümmert sich nicht darum, was nach einer vom Parser erkannten Zahl kommen darf. Das führt dazu, dass der Parser mit der vorstehenden Aufgabe 8.3 verbesserte ⟨*integer*⟩-Parser Zahlen mit einer führenden Null abspaltet in die erkannte Null und Restziffern, die nicht als Teil der Zahl gelten.

```
pc=> (integer (seq "0"))
[(\0) ()]
pc=> (integer (seq "023"))
[(\0) (\2 \3)]
```

Da wir hier den ⟨*integer*⟩-Parser nicht in eine weitere Grammatik einbetten, ist an diesem Verhalten grundsätzlich nichts zu bemängeln. Das Beispiel demonstriert jedoch deutlich, dass man bei der Definition von Grammatiken ein wachsames Auge haben muss: Setzt die Grammatik meine Vorstellungen, meine Intentionen um? Oder habe ich Sonderfälle oder ungültige Kodierungen übersehen?

▷ **Aufgabe 8.4** Erstellen Sie eine Grammatik ⟨*flt*⟩ für einfache Fließkommazahlen (*floating point numbers* oder kurz *floats*) und übersetzen Sie sie in Clojure. Eine Fließkommazahl unterscheidet sich von einer Ganzzahl durch einen Punkt „`.`“, dem Ziffern voranstehen und nachfolgen.

▷ **Aufgabe 8.5** Erstellen Sie eine Grammatik ⟨*hhmm*⟩, die ausschließlich gültige Zeitangaben im 24-Stunden-Format `hh:mm` erkennt. Verwenden Sie so wenige Terminalsymbole wie möglich. Übersetzen Sie die Grammatik in einen Clojure-Parser.

Hinweis: Die eingeführte Notation orientiert sich an verbreitete Schreibweisen für kontextfreie Grammatiken. Obwohl wir eine ähnliche Notation gewählt haben, interpretieren wir unsere Grammatiken mit Parsern und Parserkombinatoren. Andere Parsertechniken interpretieren dieselbe Notation teilweise etwas anders.

8.7 Aufbau eines Parsebaums mit `tag` und `drop`

Bis jetzt können wir Parser konstruieren, die Grammatiken abbilden und gültige Eingabesequenzen erkennen. Was uns fehlt, ist der Aufbau eines Parsebaums. Ein Vektor mit dem erfolgreich erkannten Teil der Eingabesequenz und dem verbleibenden Rest ist nur bedingt hilfreich.

Es ist erstaunlich einfach, die „Baumfunktionalität“ nachzurüsten. Wir führen einen `tag`-Parser ein, der ein erfolgreiches Parse-Ergebnis `res1` eines Parsers `parser` in einen Vektor fasst und mit einem Auszeichner `tag` annotiert.

```
(defn tag [tag parser]
  (fn [in]
    (let [[res1 in1 :as res] (parser in)]
      (if res1 [[tag res1] in1] res))))
```

Ein weiterer nützlicher Parser ist der mit der Funktion `drop` erzeugte Parser. Er verwirft einen erfolgreichen Parsevorgang mit dem konfigurierten Parser `parser` und ersetzt ihn durch einen `success`-Parser.

```
(defn drop [parser]
  (fn [in]
    (let [[res1 in1 :as res] (parser in)]
      (if res1 (success in1) res))))
```

Nehmen wir als Beispiel den Parser für Fließkommazahlen, siehe Aufgabe 8.4 (Seite 96). Ohne Annotationen und ohne ein `drop` ist der Parser in Clojure definiert über:

```
(def flt (and integer (item \.) (more dig0-9)))
```

Für die Darstellung im Parsebaum ist der Punkt als Trennzeichen für die Ziffern unerheblich – wir wollen ihn verwerfen. Erhalten möchten wir uns die Information, dass wir eine Fließkommazahl und ihre Ziffernanteile erkannt haben. Dafür verwenden wir als Auszeicher die Schlüsselwörter `:flt`, `:left` und `:right`. Der erweiterte `flt`-Parser stellt sich übersichtlich formatiert dar als:

```
(def flt
  (tag :flt
    (and
     (tag :left integer)
     (drop (item \.))
     (tag :right (more dig0-9)))))
```

Anhand einer beispielhaften Interaktion wird sofort deutlich, wie der Parsebaum aussieht. Der Ergebnisvektor hat das Schlüsselwort `:flt`, dem eine Sequenz folgt mit den beiden und über die jeweiligen Schlüsselwörter ausgewiesenen Teilergebnisse.

```
pc=> (flt (seq "123.45"))
[[:flt (:left (\1 \2 \3) :right (\4 \5))] ()]
```

Wenn man eine Grammatik gut annotiert und „unwesentliche“ Teile verwirft, kann der Parsebaum in eine Map gewandelt werden, was den Zugriff auf den Baum sehr einfach macht.

▷ **Aufgabe 8.6** Der Clojure-Parser für die Uhrzeit aus Aufgabe 8.5 soll erweitert werden. Die erkannten Stunden sollen mit dem Tag `:hh`, die Minuten mit dem Tag `:mm` und die Gesamtzeit mit dem Tag `:time` versehen werden. Der Doppelpunkt als Trennzeichen zwischen Stunden- und Minutenangaben soll im Parsebaum nicht erscheinen.

In der Notation für Grammatiken sind Annotationen und „Verwerfungen“ nicht vorgesehen. Vorstellbar sind Erweiterungen wie z.B. Tags als tiefergestellte Indizes und die Markierung von Auslassungen mit einem Minuszeichen. Das soll uns an dieser Stelle jedoch nicht weiter beschäftigen.

8.8 Rekursive Grammatiken

Die bisherigen Grammatiken, die wir uns in dieser Fallstudie angeschaut haben, zeichnet eine Besonderkeit aus: die Teilgrammatiken bzw. Teilparser bauen schrittweise aufeinander auf. Wenn Sie sich mit Grammatiken etwa für Programmiersprachen beschäftigen, werden Sie schnell feststellen, dass ein solcher Aufbau nicht durchzuhalten ist. Es gibt Bezüge auf Teile der Grammatik, die noch gar nicht definiert sind. Ein Vorziehen der Definition nützt nichts, da der vorgezogene Teil sich ebenso auf noch nicht definierte Teile bezieht. Die Problematik ist rekursiven Grammatiken eigen.

Schauen wir uns dazu die folgende Grammatik an. Das Beispiel ist konstruiert und von wenig praktischem Nutzen, bringt den wechselseitigen Bezug (die Rekursion) jedoch auf den Punkt.

⟨*a-parser*⟩ := ‘`a`’ b-parser*

⟨*b-parser*⟩ := ‘`b`’ a-parser*

Man kann es drehen und wenden wie man will, der wechselseitige Bezug der beiden Parser lässt sich nicht auflösen. Er ist in der Definition verankert.

▷ **Aufgabe 8.7** Wird die Zeichenfolge `(seq "abba")` von dem Parser ⟨*a-parser*⟩ als gültige Folge erkannt? Begründen Sie Ihre Antwort.

Wenn wir dem bisherigen Ansatz folgen und die Grammatik schematisch in Clojure übersetzen, stoßen wir auf Probleme. Der Versuch scheitert, den ⟨*a-parser*⟩ in Clojure wie gehabt zu übersetzen. Um eine direkte Rückmeldung zu haben, programmieren wir über die REPL.

```
pc=> (def a-parser (and (item \a) (many b-parser)))
java.lang.Exception: Unable to resolve symbol: b-parser in this context (N
CE_FILE:3)
```


Das erste Problem ist, dass ein vorangehendes `(declare b-parser)` fehlt. Bei rekursiven Definitionen müssen Sie in Clojure das noch nicht an einen Wert gebundene Symbol, hier `b-parser`, per `declare` mit einer Variablen assoziieren. Der Bindungswert ist zwar noch offen, aber das Symbol ist nicht mehr undefiniert. Die Assoziation eines Symbols mit einer Variablen und deren Bindung an einen Wert sind zwei fein zu unterscheidende Dinge.

Leider löst uns `declare` das Problem nicht vollständig.

```
pc=> (declare b-parser)
#'pc/b-parser
pc=> (def a-parser (and (item \a) (many b-parser)))
java.lang.IllegalStateException: Var pc/b-parser is unbound. (NO_SOURCE_FI
```

Nun beklagt sich Clojure über den fehlenden Bindungswert an die Variable. Das Problem entsteht durch die Auswertung des `and`-Ausdrucks als Teil der Auswertung des `def`-Ausdrucks. Im Zuge dieser Auswertung wird auch der Bindungswert für `b-parser` angefordert – der noch nicht aufgesetzt ist.

Die Auswertung eines Ausdrucks kann auf zwei Weisen unterdrückt werden: mit `quote` (bzw. dem Reader-Makro `'`) und Funktionsausdrücken. Im Fall von `quote` wird die Auswertung mit `eval` „nachgeholt“. Ein einfaches Beispiel:

```
pc=> (eval '(+ 7 3))
10
```

Packt man einen Ausdruck in den Rumpf einer anonymen Funktion, die keine Argumente erwartet, so wird die Auswertung des Rumpfes ebenfalls unterdrückt. Seine „nachgeholte“ Auswertung geschieht über einen Funktionsaufruf ohne Argumente. Am Beispiel:

```
pc=> (fn [] (+ 7 3))
#<pc$eval70$fn__71 pc$eval70$fn__71@23000bcf>
pc=> ((fn [] (+ 7 3)))
10
```

Falls Sie das `delay`-Makro in Clojure kennen: Mit `delay` lässt sich die Auswertung des übergebenen Ausdrucks unterdrücken und mit der Funktion `force` nachholen. Tatsächlich basiert das `delay`-Makro genau auf dem beschriebenen Mechanismus der Verzögerung mit einer Funktion.

In unserem konkreten Fall ist die Rückstellung der Auswertung per Funktion die bessere Wahl der Mittel. Sie bringt am wenigsten Komplikationen mit sich und fügt sich bestens in unseren funktionsbasierten Parseransatz ein.

Wir wissen, dass sowohl die Auswertung von Parserkombinatoren als auch die Auswertung eines `item`-Ausdrucks immer einen Parser ergibt. Jeder dieser Parser ist eine Funktion, die ein `in`-Argument erwartet. Setzen wir den Kombinatorausdruck in den Rumpf einer Funktion, so können wir die Auswertung zum Parser unterbinden. Bietet diese einbettende Funktion selbst ein `in`-Argument an, so sieht die Funktion selber wie ein Parser aus, die bei ihrem Aufruf das Argument einfach nach „innen“ durchreicht. (Beachten Sie den Gebrauch von `defn` statt von `def`!)

```
pc=> (declare b-parser)
#'pc/b-parser
```

```
pc=> (defn a-parser [in] ((and (item \a) (many b-parser)) in))
#'pc/a-parser
pc=> (defn b-parser [in] ((and (item \b) (many a-parser)) in))
#'pc/b-parser
```

Nun kann uns die Realisierung der rekursiven Grammatik eine Antwort auf Aufgabe 8.7 geben; eine Erklärung des Verhaltens finden Sie in der Lösung zur Aufgabe.

```
pc=> (a-parser (seq "abba"))
[(\a \b \b \a) ()]
```

Mit diesem Rüstzeug ausgestattet, können Sie nun aufwendige Grammatiken als Parser in Clojure realisieren und sich einen Parsebaum erzeugen.

▷ **Aufgabe 8.8** Clojure ist ein Nachfahre der Sprache Lisp. Eine einfache Grammatik für Lisp bestehe einzig aus Zahlen, Symbolen und Listen.

⟨*space*⟩ := (‘ ’|newline|tab)+

⟨*symbol*⟩ := (‘a’-‘Z’)+

⟨*number*⟩ := (‘+’|‘-’)? (‘0’-‘9’)+

⟨*llist*⟩ := ‘(’ ⟨*space*⟩? (⟨*expr*⟩? (⟨*space*⟩ ⟨*expr*⟩)*)? ⟨*space*⟩? ‘)’

⟨*expr*⟩ := ⟨*symbol*⟩ | ⟨*number*⟩ | ⟨*llist*⟩

Der Einfachheit halber sind Zahlen und Symbole mit einer minimalen Syntax versehen. Die Ausdrücke (‘a’-‘Z’) und (‘0’-‘9’) sind eine verkürzte Schreibweise für die Alternativen der Einzelzeichen von ‘a’ bis ‘Z’ des Alphabets und der Ziffern von ‘0’ bis ‘9’. Die Grammatik für ⟨*llist*⟩ geht großzügig mit dem Gebrauch von ⟨*space*⟩ innerhalb eines Listenausdrucks um. Setzen Sie diese Grammatik mit Clojure um mit dem Ziel, einen Parser für ⟨*expr*⟩ zur Verfügung zu stellen.

▷ **Aufgabe 8.9** Die Umsetzung der Lisp-Grammatik aus Aufgabe 8.8 soll einen Parsebaum erzeugen. Leerzeichen in Ausdrücken sollen im Parsebaum vollständig ignoriert werden. Ebenso sind die öffnenden und schließenden Klammerzeichen einer Liste im Parsebaum unerheblich. Symbole, Zahlen und Listen sind mit den Tags `:sym`, `:num` und `:lst` auszuzeichen. Passen Sie den Clojure-Parser entsprechend an.

8.9 Lösungen

Aufgabe 8.1 Erinnern Sie sich an das Kernel-Prinzip: Wenn möglich, definieren Sie „neue“ Funktionen durch Verwendung bereits erstellter Funktionen. Ein `many`-Parser ist ein `optional` eingesetzter `more`-Parser:

```
(defn many [parser] (optional (more parser))) ; *..0
```

Aufgabe 8.2

```
(def dig0-9 (or (item \0) dig1-9))
```

Aufgabe 8.3 Die Grammatik ist in der eingeführten Notation angegeben. Die Übersetzung in Clojure bleibe Ihnen zur Übung.

⟨*sign*⟩ := ‘`+`’ | ‘`-`’

⟨*zero*⟩ := ‘`0`’

⟨*dig1-9*⟩ := ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’

⟨*dig0-9*⟩ := ⟨*zero*⟩ | ⟨*dig1-9*⟩

⟨*integer*⟩ := ⟨*sign*⟩? ((⟨*dig1-9*⟩ ⟨*dig0-9*⟩∗) | ⟨*zero*⟩)

Wenn es gilt, so wenige Terminalsymbole wie möglich zu verwenden, so ist bei Mehrfachverwendung eines Terminalsymbols ein namentlicher Parser einzuführen. Hier verwenden wir ⟨*zero*⟩ für ‘`0`’.

Aufgabe 8.4 Man kann sich der bisherigen Parser-Definitionen bedienen. Die Übersetzung in Clojure bleibe Ihnen zur Übung, findet sich aber aufgelöst in Kap. 8.7 (Seite 97).

⟨*flt*⟩ := ⟨*integer*⟩ ‘`.`’ ⟨*dig0-9*⟩+

Zu Aufgabe 8.5: Um Namenskonflikte zu vermeiden mit vorstehenden Aufgaben, tragen die Parser-Definitionen für Ziffern (*digits*) andere Namen. Die Übersetzung in Clojure bleibe Ihnen zur Übung.

⟨*digit0-1*⟩ := ‘`0`’ | ‘`1`’

⟨*digit2*⟩ := ‘`2`’

⟨*digit0-3*⟩ := ⟨*digit0-1*⟩ | ⟨*digit2*⟩ | ‘`3`’

⟨*digit0-5*⟩ := ⟨*digit0-3*⟩ | ‘`4`’ | ‘`5`’

⟨*digit0-9*⟩ := ⟨*digit0-5*⟩ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’

⟨*hh*⟩ := (⟨*digit0-1*⟩ ⟨*digit0-9*⟩) | (⟨*digit2*⟩ ⟨*digit0-3*⟩)

⟨*mm*⟩ := ⟨*digit0-5*⟩ ⟨*digit0-9*⟩

⟨*hhmm*⟩ := ⟨*hh*⟩ ‘`:`’ ⟨*mm*⟩

Die Erstellung der Clojure-Parser für ⟨*hh*⟩, ⟨*mm*⟩ und ⟨*hhmm*⟩ bleibe Ihnen zur Übung; eine leicht erweiterte Lösung findet sich in der Lösung zu Aufgabe 8.6.

```
(def digit0-1 (or (item \0) (item \1)))
(def digit2   (item \2))
(def digit0-3 (or digit0-1 digit2 (item \3)))
(def digit0-5 (or digit0-3 (item \4) (item \4)))
(def digit0-9 (apply or digit0-5 (map item [\6 \7 \8 \9])))
```

Aufgabe 8.6 Die Clojure-Parser für die Ziffern sind identisch mit der Lösung zu Aufgabe 8.5.

```
(def hh (tag :hh
  (or
    (and digit0-1 digit0-9)
```

```
    (and digit2   digit0-3))))
(def mm   (tag :mm (and digit0-5 digit0-9)))
(def hhmm (tag :time (and hh (drop (item \:)) mm)))
```

Aufgabe 8.7 Ja, die Folge `(seq "abba")` ist gültig. Der $\langle a \rangle$-Parser erkennt das erste '`a`', dem beliebig Anwendungen des $\langle b \rangle$-Parsers folgen. Die erste Anwendung des $\langle b \rangle$-Parsers „verzichtet“ auf die Anwendung des $\langle a \rangle$-Parsers, die zweite Anwendung des $\langle b \rangle$-Parsers lässt eine Anwendung des $\langle a \rangle$-Parsers folgen, der – da nun keine Zeichen mehr folgen – die Nicht-Anwendung des $\langle b \rangle$-Parsers tolleriert und einen Schluß findet.

Historie

22. Mai 2012	Anpassung der Formatierung für Lösungen
23. Dez. 2011	Korrektur von Rechtschreibfehlern. Mit Dank an Marc Hesenius.
10. Juli 2011	Korrektur: „erfolgreich“ ohne „g“ geschrieben.
8. Juli 2011	Zwei Korrekturen: Ein „Wie“ sollte „Wir“ heißen, ein „`:float`“ im Code „`:flt`“. Vielen Dank an Alexander Vogel.
3. Juli 2011	Erste Version.

9 Makros

Eine Besonderheit von Clojure ist, dass jedes Programm grundsätzlich auch eine gültige Repräsentation eines Datums ist. Jede Programmform, wie z.B. eine Funktionsdefinition, kann quotiert und als Liste manipuliert werden. Das Beispiel zerlegt die `fn`-Form in ihre Bestandteile.

```
user=> (first '(fn [x] (* x x)))
fn
user=> (second '(fn [x] (* x x)))
[x]
user=> (last '(fn [x] (* x x)))
(* x x)
```

Umgekehrt kann aus Daten ein Programm zusammengesetzt werden – vorausgesetzt, die erstellte Form ist eine gültige Programmform. Aus den Fragmenten `fn`, `[x]` und `(* x x)` lässt sich wieder eine Liste erzeugen, die per `eval` als Programm interpretiert wird. Zur Anschauung wenden wir die Funktion auf `3` an.

```
user=> (list 'fn '[x] '(* x x))
(fn [x] (* x x))
user=> ((eval (list 'fn '[x] '(* x x))) 3)
9
```

Die Grundlage dieser Wandlungsfähigkeit, Programme als Daten und Daten als Programme behandeln zu können, liegt in der Syntax von Clojure begründet. Programmausdrücke sind Listen, ihre Teilausdrücke werden ebenfalls wie Daten notiert. Das macht Clojure zu einer sogenannten *homoikonischen* Sprache.

Diese syntaktische „Formengleichheit“ von Programmen und Daten erlaubt auf einfache Weise die Realisation eines Umschreibsystems per Makros. Makros sind neben den Funktionen ein weiterer Mechanismus zur Bildung von Abstraktionen.

Das Makrosystem von Clojure ist vollständig programmierbar, was es zu einem mächtigen Werkzeug macht. Es ist ein programmierbares Umschreibsystem. Man kann Code-Generatoren schreiben, domänenspezifische Sprachen (*Domain Specific Languages*, DSLs) implementieren oder ansonsten komplizierte und unübersichtliche Programmformen in eine syntaktisch übersichtliche und einfache Form bringen.

9.1 Makros definieren

Mit `defmacro` wird ein Makro definiert. Äußerlich sieht `defmacro` wie eine Funktionsdefinition per `defn` aus. Es folgen der Makroname, die Argumente und der Rumpf, der das Verhalten des Makros definiert. Der Aufruf eines Makros unterscheidet sich vom Aufruf einer Funktion in zwei entscheidenden Punkten:

- Ein Makro wertet die Argumente nicht aus!

- Das Ergebnis eines Makro-Aufrufs wird anschließend automatisch evaluiert.

Wenn wir z.B. die Quadratfunktion als Makro realisieren wollen, dann schreibt das Makro die Eingabeform um in einen Ausdruck, der die Eingabeform wiederholt und davor ein Symbol stellt, das – wenn der Ausdruck ausgewertet wird – die Multiplikation umsetzt.

```
> (defmacro sq [x] (list '* x x))
#'user/sq
> (sq 3)
9
```

Die Anwendung von `(sq 3)` liefert das erwartete Ergebnis. Die inneren Abläufe sind jedoch gänzlich andere, als wenn die Quadratfunktion definiert worden wäre über `(defn sq [x] (* x x))`.

Mit `macroexpand` kann das Ergebnis der Makroanwendung ohne abschließende Evaluation betrachtet werden. (Mit `macroexpand-1` wird nur ein einziger Auflösungsschritt angezeigt.) Das Makro `sq` erzeugt unter Verwendung der Argumente eine neue Form *ohne* die Argumente auszuwerten. Das macht das zweite Beispiel mit `(+ 2 1)` als Argument klar.

```
> (macroexpand '(sq 3))
(* 3 3)
> (macroexpand '(sq (+ 2 1)))
(* (+ 2 1) (+ 2 1))
```

Das `sq`-Makro liefert auch mit einem nicht quotierten „`*`“-Zeichen korrekte Ergebnisse. Dennoch sollte das Multiplikationszeichen quotiert werden. Makros manipulieren Formen und sollen Formen zurückgeben, die dann – in einem zweiten Schritt – evaluiert werden. Diese beiden Schritte sollten nicht miteinander vermischt werden. Es ist schlechter Stil, das Multiplikationszeichen bereits während des Makro-Aufrufs in die Multiplikationsfunktion aufzulösen.

Damit ist eigentlich schon alles zu Makros gesagt! Schwierig ist einzig, sich in das Programmieren mit Makros einzudenken.

9.2 Makroprogrammierung mit Templates

Das Umschreiben von Formen schreit ein wenig nach Komfort. Bequem wäre es, das `sq`-Makro ohne die `list`-Funktion direkt in der Form `(* x x)` hinzuschreiben. Das wäre lesbarer und bei umfangreicheren Makroformen praktisch. Um die sonst greifende Evaluierung zu verhindern, müsste der Ausdruck quotiert werden, `'(* x x)`. Allerdings verhindert die Quotierung das Einsetzen der an `x` gebundenen Form.

Es gibt eine besondere Quotierungsart, die dieses gezielte Einsetzen von Formen in eine „Quotierungsschablone“ ermöglicht. Sie heißt Syntax-Quote und wird mit dem Backquote-Zeichen „`‘`“ eingeleitet. Auf einer deutschen Tastatur liegt das Backquote-Zeichen neben der „ß“-Taste und ist zusammen mit der Umschalt-Taste erreichbar.

Syntax-Quote verhält sich fast wie eine „normale“ Quotierung, mit zwei Abweichungen. Erstens: Ein Syntax-Quote wandelt die Symbole in einem quotierten Ausdruck um in voll qualifizierte Symbole.

```
> '(* x x)
(* x x)
```



```
> `(* x x)
(clojure.core/* user/x user/x)
```

Zweitens: Der Wert eines Symbols, sprich seine Evaluation, kann durch ein vorangestelltes Unquote-Zeichen „`~`“ bzw. Unquote-Splicing „`~@`“ angefordert werden. Ein Unquote ersetzt das Symbol mit der an ihn gebundenen Form, während ein Unquote-Splicing die Einzelwerte einer gebundenen Sequenz übernimmt.

```
> (let [x '(+ 2 1)] `(* ~x ~x))
(clojure.core/* (+ 2 1) (+ 2 1))
> (let [x '(3 2 1)] `(* ~@x ~@x))
(clojure.core/* 3 2 1 3 2 1)
```

Das eingangs eingeführte `sq`-Makro lässt sich also per Syntax-Quote auch definieren als

```
(defmacro sq [x] `(* ~x ~x))
```

Die Quotierung mittels Backquote und die vorrübergehende Aufhebung der Quotierung per Unquote bzw. Unquote-Splicing realisieren einen sehr einfachen aber wirkungsvollen Mechanismus für Code-Schablonen (*templates*). Mit einem Backquote wird eine Form aufgesetzt, in der Unquotes bzw. Unquote-Splicings wie Platzhalter für Einfügungen agieren. Auf diese Weise vereinfacht sich die Makro-Programmierung in vielen Fällen auf die Programmierung von Code-Schablonen.

Clojure definiert einige Makros, teils um syntaktisch geeignetere Formen anzubieten. Diese Abstraktionen werden auch als „syntaktischer Zucker“ (*syntactic sugar*) bezeichnet. Ein Beispiel ist `when`, das Ihnen statt eines `if`s die Angabe eines else-Teils erspart. Unser Makro heiße `my-when`, um das originale `when`-Makro nicht zu überschreiben.

```
(defmacro my-when [test & body] `(if ~test (do ~@body)))
```

Wie bei Funktionen kann ein Makro optionale Argumente entgegen nehmen. Der gesamte `body` wird mit einem `do` umschlossen, um eine „geordnete“ Evaluation aller `body`-Argumente im `if` zu garantieren.

Mit `(source when)` gibt Clojure den Makro-Code zu `when` preis. Der Clojure-Kernel definiert Makros bisweilen ohne Template-System, der Code ist aber praktisch identisch mit `my-when`.

▷ **Aufgabe 9.1** Warum wird im Makro `my-when` ein Unquote-Splicing `~@body` verwendet? Warum ist ein Unquote `~body` nicht ausreichend?

▷ **Aufgabe 9.2** Schreiben Sie ein Makro `my-when-not`, das sich wie das Makro `when-not` verhält.

▷ **Aufgabe 9.3** Angenommen, in Clojure gäbe es nur Funktionen und Makros und keine weiteren Spezialformen. Konsequenterweise existierte die Spezialform `if` nicht, sondern eine primitive Funktion z.B. namens `choice`. Da Clojure keine derartige primitive Funktion bereit stellt, werde `choice` mittels `if` simuliert.

```
(defn choice [pred t-case f-case] (if pred t-case f-case))
```

(1) Inwiefern unterscheidet sich `choice` von `if`? (2) Schreiben Sie ein Makro `if-macro`, dass sich wie `if` verhält, ohne es zu verwenden. Das `if-macro` ist mit `choice` umzusetzen.

9.3 Symbole generieren mit `gensym`

Die aktuelle Version des `sq`-Makros generiert ein Code-Template, das das Argument verdoppelt und damit zweimal auswerten lässt. Möchte man das Argument nur einmal auswerten, kann `let` zum Einsatz kommen.

```
> (defmacro sq [x] `(let [x ~x] (* x x)))
#'user/sq
> (macroexpand '(sq (+ 2 3)))
(let* [user/x (+ 2 3)] (clojure.core/* user/x user/x))
> (sq (+ 2 3))
CompilerException java.lang.RuntimeException: ...
```

Die Makroexpansion funktioniert zwar, aber bei der Übersetzung des Templates in ausführbaren Code verweigert sich der Compiler: `let` arbeitet nicht mit qualifizierten Symbolen, es „stört“ sich an der automatischen Konversion von Symbolen in qualifizierte Symbole.

Hier hilft ein besonderes Makro-Feature von Clojure aus: Fügt man einem Symbol ein `#`-Symbol an, dann generiert Clojure ein „frisches“, eindeutiges Symbol für den Symbolnamen. Im Hintergrund wird `gensym` zur Erzeugung des Symbols genutzt.

```
> (defmacro sq [x] `(let [x# x] (* x# x#)))
#'user/sq
> (macroexpand '(sq (+ 2 3)))
(let* [x__439__auto__ user/x]
  (clojure.core/* x__439__auto__ x__439__auto__))
user=> (sq (+ 2 3))
25
```

Die bei der Generation eines `gensym`-Symbols verwendeten Zahlen machen die generierten Symbole gleichsam zufällig aber eindeutig im verwendeten Kontext.

▷ **Aufgabe 9.4** Das folgende Makro sieht der Makrodefinition mit dem `let` im Template sehr ähnlich. Dennoch gibt es einen Unterschied! Worin besteht er?

```
(defmacro sq [x] (let [x' x] (list '* x' x')))
```

Analog zu Funktionen können Makros für verschiedene Anzahlen an Argumenten definiert werden (*variadic macros*). Und selbstverständlich dürfen Makros Makros enthalten, selbst rekursive Makros sind erlaubt. Zum Beispiel ist `and` in Clojure über ein rekursives Makro definiert. Abzüglich Kommentare und Metainformation liefert Ihnen `(source and)` folgenden Code:

```
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

Ohne Argument gibt `(and)` ein `true` zurück. Bei einem Argument liefert `and` den Wert des Arguments selbst zurück. Bei zwei und mehr Argumenten wird es interessant. Um eine Doppelauswertung zu vermeiden, wird das erste Argument an das generierte Symbol `and#` gebunden. Wenn im

`if` der `and#`-Wert `false` oder `nil` ist (darauf testet `if`), dann wird dieser Wert zurück gegeben (else-Teil des `if`s). Ansonsten wird das `and`-Makro mit allen übrigen Argumenten aufgerufen; dafür sind die `next`-Argumente mit einem Unquote-Splicing aus der Klammer zu befreien.

▷ **Aufgabe 9.5** Realisieren Sie ein Makro `my-or`, das sich wie `or` verhält. Die Arbeitsweise von `or` können Sie nachlesen unter `(doc or)`.

9.4 Alternativen zu einem Makrosystem

Es gibt wenige Programmiersprachen, die eine Manipulation des Programmcodes erlauben. Da viele Programmiersprachen keine homoikonischen Sprachen sind, also auf die syntaktische „Formengleichheit“ von Programmen und Daten verzichten, kann kein Makrosystem zum Einsatz kommen. Nur wenige Programmiersprachen bieten als Makro-Alternative die Repräsentation eines Programms in der aus dem Compilerbau bekannten Form des „Abstrakten Syntaxbaums„ (*Abstract Syntax Tree*, AST) an. Über ein Interface zur Manipulation der Baumstruktur kann ein Programm verändert werden. Um diese Manipulationen zu erleichtern, wird oftmals ein auf Vorlagen (*templates*) basierendes Umschreibsystem eingesetzt, was von der Idee her ähnlich zu Clojure-Templates ist – eine Technik, die in der Web-Programmierung Verbreitung gefunden hat für die Erzeugung von HTML-Seiten. Allerdings ist die Einführung neuer syntaktischer Formen in konventionellen Programmiersprachen komplizierter, da die bestehende Grammatik der Programmiersprache nicht Mehrdeutigkeiten hervorbringen darf. Auch sind Konflikte mit der syntaktischen Evolution einer nicht-homoikonischen Sprache ein Problem.

In homoikonischen Sprachen ist die Bereitstellung eines Makrosystems zur Umschreibung von Programmteilen praktisch „umsonst“ zu haben. Es ist kein Aufwand, ein Makrosystem einer homoikonischen Sprache als Sprachfeature mitzugeben. In nicht-homoikonischen Sprachen ist die Entwicklung einer benutzerfreundlichen AST-Manipulation und die Bereitstellung eines Vorlagen-basierten Umschreibsystems eine beachtliche Herausforderung. Für nicht-homoikonische Sprache werden eher eigene Systeme zur Code-Generierung oder für die Umsetzung von DSLs angeboten. Der Engineering- und Einarbeitungsaufwand ist deutlich höher als die Nutzung des Makrosystems in Clojure.

Unter Clojure-Programmierer(innen) gibt es eine Regel: Wenn möglich, verzichte auf Makros und definiere stattdessen Funktionen. In der Tat kann man sich an „unsauber“ konzeptionierten Makros die Zähne ausbeißen. Manchmal ist es schwer, den intendierten Umschreibeffekt korrekt umzusetzen. Bei allen Hürden der Makro-Programmierung sind Makros ein sehr leistungsfähiges und flexibles Werkzeug. Gezielt eingesetzt sind mit ihnen elegante und sehr effiziente Lösungen möglich.

9.5 Lösungen

Aufgabe 9.1 Wenn mehr als ein Argument an `my-when` übergeben wird, werden die „überzähligen“ Argumente als Liste zusammengefasst an `body` gebunden. Für die `do`-Form müssen die Argumentwerte aus der Liste per Unquote-Splicing geholt und in die Form eingefügt werden.

Aufgabe 9.2 Die Lösung verrät Ihnen `(source when-not)`. Ebenso gültig ist die Template-Variante:

```
(defmacro my-when-not [test & body]
  `(if ~test nil (do ~@body)))
```

Aufgabe 9.3 Bei der Spezialform `if` bestimmt das Ergebnis der Auswertung des ersten Ausdrucks, welcher der beiden Folgeausdrücke ausgewertet wird. Ist das Ergebnis ein `false` oder `nil`, wird der dritte Ausdruck ausgewertet, ansonsten der zweite. Die Funktion `choice` hingegen wertet immer alle drei Ausdrücke aus und liefert abhängig vom ersten Auswertungergebnis entweder den Wert des zweiten oder des dritten Ausdrucks zurück. Mit `choice` werden überflüssigerweise Rechenressourcen zur Auswertung eines Ausdrucks beansprucht, dessen Ergebnis verworfen wird. Besonders auffällig ist dieses Verhalten bei Seiteneffekten. Mit `if` wird nur der gewünschte Seiteneffekt erzeugt, mit `choice` werden immer beide erzeugt.

```
> (if true (println "Launch Rocket") (println "Wait"))
Launch Rocket
nil
> (if false (println "Launch Rocket") (println "Wait"))
Wait
nil
> (choice true (println "Launch Rocket") (println "Wait"))
Launch Rocket
Wait
nil
> (choice false (println "Launch Rocket") (println "Wait"))
Launch Rocket
Wait
nil
```

Um mit `if-macro` das Verhalten von `if` nachzubilden, müssen der zweite und dritte an `choice` übergebe Ausdruck von einer Auswertung abgehalten werden. Eine Einbettung in eine `fn`-Form verzögert die Auswertung bis die Funktion ausdrücklich aufgerufen wird.

```
(defmacro if-macro [pred t-case f-case]
  `((choice ~pred (fn [] ~t-case) (fn [] ~f-case))))
```

Die Auswertung kann ebenso mit einer Quotierung unterdrückt werden. Die Lösung hat nur einen Haken: Sie benötigt die Spezialform `quote`! Das ist laut Aufgabe nicht erlaubt.

```
(defmacro if-macro [pred t-case f-case]
  `(choice ~pred '~t-case '~f-case))
```

Die Nachbildung von `quote` per Makro ist nicht trivial. Ähnlich wie bei einer Evaluation mittels `eval` muss ein `quote`-Makro den übergebenen

Ausdruck vollständig analysieren und als Liste, Vektor etc. neu konstruiert zurückgeben.[1]

Aufgabe 9.4 Den Unterschied verrät ein `macroexpand`. Es wird die übergebene Form an `x'` gebunden und damit wieder ein Code-Template erzeugt, in dem die Form zweimal vorkommt.

```
> (macroexpand '(sq (+ 2 3)))
(* (+ 2 3) (+ 2 3))
```

Aufgabe 9.5 Die Lösung hat Clojure für Sie parat unter `(source or)`.

[1] Eine Lösung von Oleg Kiselyov für den Lisp-Dialekt Scheme findet sich unter http://okmij.org/ftp/Scheme/macros.html#quote.

Historie

22. Mai 2012	Lösungskapitel erscheint im Inhaltsverzeichnis
2. Feb. 2012	Schreibkorrektur bei Aufgabe zu `if-macro`.
30. Nov. 2011	Ein Dank an Marc Hesenius für die Korrekturen und Anmerkungen und eine weitere Übungsaufgabe (Aufgabe 9.1).
25. Nov. 2011	Erste Fassung.
8. Aug. 2011	Schreibbeginn

10 Ohne `if` zur Polymorphie

Die Anti-IF-Kampagne von Francesco Cirillo hat dem `if` im Code den Kampf angesagt:[1] Je weniger `if`, desto besser – dafür gebe es einen Zugewinn an Flexibilität, der den Streit um jedes einzelne `if` wert sei.

Der Feldzug gegen `if`-Ausdrücke scheint provokant zu sein. Doch nehmen wir die Kampagne beim Wort und verbieten jeglichen Gebrauch eines `if`s im Code. Geht das überhaupt?

Ja, es geht. Selbst erfahrene Programmierer(innen) können das aufs Erste nicht glauben. Sie werden in diesem Kapitel lernen, dass Maps alles sind, was man als `if`-Ersatz braucht. Und wenn man den Ansatz mit den Maps konsequent weiter denkt, landet man bei dem Konzept polymorpher Funktionen. Von dort aus ist es nur noch ein kleiner Schritt zur Objekt-Orientierung.

Sind Sie so weit gekommen, ist leicht verständlich, wie Clojure Polymorphie umsetzt. Clojure ist mit seinen Sprachkonzepten wesentlich flexibler als es die meisten objekt-orientierten Programmiersprachen sind.

10.1 Maps beinhalten das Konzept zur Entscheidung und zum Vergleich

Erinnern Sie sich an die Funktionweise von Maps. Eine Map besteht aus Paaren von Werten: der erste Wert ist der Schlüssel (*key value*) der mit dem jeweils zweiten Paarwert als Zielwert (*target value*) assoziiert ist. Ein Beispiel sei die Assoziation eines Wochentages mit einer Zahl. Die Wochentage seien mittels abgekürzter Schlüsselwörter kodiert.

```
user=> (def day {:mon 1 :tue 2 :wed 3 :thu 4 :fri 5})
#'user/day
```

Maps können wie Funktionen in einem Programmausdruck behandelt werden – eine Besonderheit in Clojure, die das Abfragen von Zielwerten zu Schlüsselwerten sehr einfach macht.

```
user=> (day :thu)
4
user=> (day :sat)
nil
```

Für unbekannte Schlüsselwerte kann optional ein Default-Wert zur Rückgabe angegeben werden.

```
user=> (day :thu 0)
4
user=> (day :sat 0)
0
```

Offensichtlich muss zur Abfrage von assoziierten Werten ein Vergleich mit den Schlüsselwerten stattfinden. Das Konzept der Gleichheit von Werten ist bereits in der Idee der Map eingebaut. Wir können das nutzen, um mit Hilfe einer Map die Funktion der Gleichheit zu realisieren.

[1] http://www.antiifcampaign.com (2012-05-04)

```
user=> (defn equal? [x y] ({y true} x false))
#'user/equal?
```

Die Funktion `equal?` leistet genau das gleiche wie die Funktion `=`. Ist der Wert `x` in der Map `{y true}` als Schlüssel vorhanden, ist der assoziierte Wert `true` das Ergebnis. Ansonsten ist es der Default-Wert `false`.

```
user=> (equal? [1 2 3] [1 2 3])
true
user=> (equal? [2 3] [1 2 3])
false
```

▷ **Aufgabe 10.1** Implementieren Sie mit Hilfe einer Map eine Funktion namens `NOT`, die das Verhalten der Funktion `not` nachbildet. Die Funktion `not` gibt `true` zurück, wenn das Argument logisch falsch, d.h. `false` oder `nil` ist. Ansonsten ist die Rückgabe `false`.

Das Beispiel der Umsetzung von `equal?` und die Aufgabe zu `NOT` geben Ihnen den entscheidenden Hinweis, wie man ein `if` mit einer Map nachbilden kann. Die `if`-Spezialform hat mindestens zwei Argumente und ein optionales drittes Argument: einen `test`-Ausdruck, einen `then`- und einen optionalen `else`-Ausdruck; fehlt der `else`-Ausdruck, so wird er als `nil` angenommen. Der `else`-Ausdruck wird nun dann ausgewertet, wenn das Ergebnis der `test`-Auswertung `false` oder `nil` ist. In allen anderen Fällen wird der `true`-Ausdruck ausgewertet. Siehe auch `(doc if)`.

Mit Hilfe einer Map könnte man `(if (== 2 3) (+ 2 3) (- 2 3))` wie folgt umsetzen:

```
user=> (eval ({false '(- 2 3) nil '(- 2 3)} (== 2 3) '(+ 2 3)))
-1
```

Die Quotierungen des `then`- und des `else`-Ausdrucks sind notwendig, um eine vorzeitige Evaluierung auszuschließen. Nach der „Entscheidung“ durch die Map wird die Evaluation per `eval` nachgeholt.

Es ist also möglich, `if`-Spezialform durch einen Ausdruck zu ersetzen, der die Maps als Entscheidungsoperator nutzt. Diese Form eines `if`-Ersatzes wollen wir mit Hilfe eines Makros erfassen, das `IF` heißen soll.

```
(defmacro IF
  ([test then]      `(IF ~test ~then nil))
  ([test then else] `(eval ({false '~else, nil '~else} ~test '~then))))
```

Zur Auffrischung: Das Quotierungszeichen ` zeichnet einen Ausdruck als Template aus, in das per `~` (*unquote*) die Ausdrücke aus den Makro-Argumenten eingesetzt werden.

Das Faszinierende ist nicht nur, dass sich `IF` wie ein `if` verhält – probieren Sie es an der Konsole aus:[2]

```
user=> (IF (== 2 3) (+ 2 3) (- 2 3))
-1
```

Vielmehr noch: Das `IF`-Makro liest sich wie eine perfekte Spezifikation! Das Makro sagt exakt und unmissverständlich, wie ein `IF` (bzw. `if`) zu

[2] Ein Hinweis für Makro-Experten: Das `eval` verhindert die Berücksichtigung eines lexikalischen Kontextes. Ein `(let [x 3] (IF true x))` liefert eine Fehlermeldung, während der gleiche Ausdruck mit `if` eine `3` ergibt. Das ist ein Problem, dessen Lösung uns hier nicht beschäftigen soll.

verstehen ist – wir haben es gerade eben beschrieben und sehen nun im `IF`-Makro die nahezu wörtliche Umsetzung in Code: Ein fehlendes optionales drittes Argument wird durch ein `nil` ersetzt; die Auswertung des `test`-Ausdrucks zu `false` bzw. `nil` führt zur Auswertung des `else`-Ausdrucks, ansonsten zur Auswertung des `then`-Ausdrucks.

Alle weiteren Formen, die den Fluss der Evaluation lenken, wie z.B. `if-not`, `cond`, `when` und `when-not` sind in Clojure als Makros realisiert, die ausnahmslos auf der `if`-Spezialform basieren. Und da unser `IF`-Makro identisch zur `if`-Spezialform ist, haben wir den Beweis angetreten: Man kann auf jegliche `if`s im Code verzichten.

Damit wäre zwar das Anliegen der Anti-If-Kampagne erfüllt, aber eben nur scheinbar. Die Ersetzung von `if` durch ein Makro, das zwar intern mit einer Map realisiert ist, sich aber ansonsten wie ein `if` anfühlt und arbeitet, ist nur ein Scheingewinn. Das ist auch nicht das Anliegen der Anti-If-Kamagne. Interessant wird es, wenn man die Idee einer Map als Grundlage für die entscheidungsbasierte Ausführung ausbaut. So kann ein echter Vorteil gegenüber dem konventionellen `if` erreicht werden. Und genau darauf zielt die Anti-If-Kampagne ab.

10.2 Polymorphie

Abhängig von einem Schlüsselwert liefert eine Map den assoziierten Zielwert zurück. Wenn wir für die Zielwerte ausschließlich Funktionen verwenden, ist über den Schlüsselwert steuerbar, welche Funktion zum Einsatz kommen soll.

Angenommen, wir definieren eine Map, die für einige geometrische Formen die jeweiligen Funktionen zur Berechnung ihrer Fläche kennt. Im Beispiel seien die Grundformen Kreis (*circle*), Rechteck (*rectangle*) und Dreieck (*triangle*) durch entsprechende Schlüsselwörter ausgewiesen.

```
(def shape2func
  {:circle (fn [r] (* r r Math/PI))
   :rectangle (fn [a b] (* a b))
   :triangle (fn [a h] (* a h 0.5))})
```

Wir definieren nun eine Funktion `area`, die abhängig von der gegebenen Form die entsprechende Funktion auswählt und auf die Argumente anwendet.

```
(defn area [shape & args]
  (apply (shape2func shape) args))
```

Eine Funktion, die in Abhängigkeit von einem Wert eine geeignete Implementierung auswählt und anwendet – genau das nennt man eine polymorphe Funktion.

```
user=> (area :circle 2)
12.566370614359172
user=> (area :rectangle 2 3)
6
```

Das Wort „Polymorphie“ kommt aus dem Griechischen und setzt sich zusammen aus dem Präfix „poly“ für „viel“, „verschieden“, und dem Suffix „morph“, was soviel heißt wie „die Gestalt oder die Form betreffend“. Eine polymorphe Funktion (*polymorphic function*) ist eine verschiedengestaltige Funktion. In unserem Beispiel spiegelt sich das anschaulich

wieder in der via `shape`-Argument angezeigten „Gestalt“ der verwendeten geometrischen Form. Den verschiedenen „Gestalten“ wird `area` durch eine entsprechende, angepasste Funktion bei der Berechnung der Fläche gerecht.

Die Polymorphie ist mit einer Map wesentlich effizienter umgesetzt als man das mit `if`-Abfragen jemals tun könnte. Die Ermittlung eines Zielwertes zu einem Schlüsselwert einer Map erfolgt stets in konstanter Zeit. Würde man die Map durch eine Kette von `if`-Abfragen ersetzen (gleich einem `cond`), so würde die Zeit anwachsen mit der Anzahl der Schlüssel-/Zielwert-Paaren. Das ist unakzeptabel.

Genau darauf will die Anti-If-Campaign hinaus: Nutze Polymorphie wann immer möglich! Das ist nicht nur ein schnelles Auswahlverfahren von Optionen, es ist auch besonders flexibel. Soll zur polymorphen Funktion eine weitere Unterscheidung samt Implementierung hinzukommen, so muss nur die Map erweitert werden. Clojure bietet dazu einen passenden Definitionsmechanismus an, wie wir später noch sehen werden.

Lassen Sie uns noch eine kleine Erweiterung vornehmen und polymorphe Funktionen um eine Dispatch-Funktion erweitern – das Wort *dispatch* heißt so viel wie „Abfertigungsstelle“. Der Aufgabe der Dispatch-Funktion ist, aus den an die polymorphe Funktion übergeben Argumenten einen Wert zu ermitteln, der als Schlüssel für die Map gilt. Die Dispatch-Funktion macht die Ermittlung eines Map-Schlüssels vollkommen flexibel und frei konfigurierbar.

Gießen wir die Erweiterung in ein Makro `defnpoly` zur Definition einer polymorphen Funktion. Als Argumente sind festzulegen der Name der polymorphen Funktion, die Dispatch-Funktion, die Map mit ihren Assoziationen von Schlüsseln zu Funktionen und optional eine Default-Funktion, falls kein Schlüssel zutrifft.

```
(defmacro defnpoly
  ([name dispatch-fn map] `(defnpoly ~name ~dispatch-fn ~map nil))
  ([name dispatch-fn map default-fn]
    `(defn ~name [& args#]
      (apply (~map (apply ~dispatch-fn args#) ~default-fn) args#))))
```

Die Definition des Makros entspricht in ihrem Aufbau der `area`-Funktion, ergänzt um die Dispatch-Funktion. Zur Erinnerung: Das `#`-Zeichen in `args#` weist Clojure an, an dieser Stelle automatisch ein eindeutiges Symbol zu generieren, das für alle `args#`-Stellen dasselbe ist.

Die `area`-Funktion kann nun sehr viel angenehmer als polymorphe Funktion definiert werden. Die Dispatch-Funktion muss einzig das erste aus allen übergebenen Argumenten extrahieren. Aufgrund der Existenz der Dispatch-Funktion müssen die Funktionen in der Map mit einem zusätzlichen ersten Argument ausgerüstet werden.

```
(defnpoly area (fn [shape & args] shape)
  {:circle (fn [self r] (* r r Math/PI))
   :rectangle (fn [self a b] (* a b))
   :square (fn [self a] (area :rectangle a a))
   :triangle (fn [self a b] (* a b 0.5))})
```

Wie Sie sehen, ist der Code sehr gut lesbar: Die Funktion `area` ist in den entsprechenden Zeilen für einen Kreis, ein Rechteck usw. definiert. Die Definition für ein Quadrat (*square*) berücksichtigt den geometrischen

Zusammenhang, dass ein Quadrat ein gleichseitiges Rechteck ist. Auch so etwas geht völlig einfach und unproblematisch.

```
user=> (area :circle 2)
12.566370614359172
user=> (area :square 2)
4
```

Programmierer(innen), die mit einer objekt-orientierten Sprache vertraut sind, wird der Anblick des Codes für `area` einen gewissen Wiedererkennungswert haben. Statt auf den `shape` einen Dispatch zu machen, kann man genauso gut auf den Typ eines Objekts als alleiniges Kriterium dispatchen – man nennt das auch typ-basierten Dispatch (*type-based dispatch*). Viele OO-Sprachen sind nur auf diese Dispatch-Form ausgelegt. Ich habe absichtlich oben im Code `self` als Argument benutzt; in anderen Sprachen heißt es statt `self` auch `this`. Damit ist die Typ- bzw. Klasseninstanz gemeint.

Maps sind ein gerne gewähltes, da leichtgewichtiges Mittel, um Datenobjekte zu beschreiben. Zum Beispiel kann man einen Kreis mit seinem Durchmesser als Eigenschaft wie folgt definieren; entsprechend ist auch ein Rechteck beschreibbar.

```
(def shape1 {:type :circle :radius 2})
(def shape2 {:type :rectangle :a 2 :b 3})
```

Der Typ der jeweiligen Form wird einfach anhand eines `:type`-Schlüssels definiert. Der Dispatch auf den Typen ist einfach definiert als

```
(defn type-dispatch [self] (self :type))
```

Mit dieser Datenmodellierung ist die polymorphe Funktion `area` etwas anders zu gestalten.

```
(defnpoly area type-dispatch
  {:circle (fn [self] (* (self :radius) (self :radius) Math/PI))
   :rectangle (fn [self] (* (self :a) (self :b)))})
```

Dieser Art des Aufbaus zeigt, wie auf die Eigenschaften der Datenobjekte zugeriffen wird. Das sieht sehr ähnlich zum Punkt-Operator (*dot operator*) in objekt-orientierten Sprachen aus. Dort würde man `(self :radius)` schreiben als `self.radius` oder `this.radius`.

```
user=> (area shape1)
12.566370614359172
user=> (area shape2)
6
```

Mit polymorphen Funktionen kann man viele interessante Sachen machen – mehr als nur die übliche Objekt-Orientierung zu imitieren. So können wir beispielsweise eine Funktion `times` definieren, die zwei Argumente erwartet. Sind die Argumente Zahlen, so wird multipliziert. Bei einer Zahl und einer Zeichenkette, wird eine neue Zeichenkette erzeugt, die die Zeichenfolge der gegebenen Zeichenkette um das Viefache der Zahl wiederholt.

```
(defnpoly times (fn [x y] [(type x) (type y)])
  {[Long Long] (fn [x y] (* x y))
   [Long String] (fn [n s] (reduce str (repeat n s)))
   [String Long] (fn [s n] (times n s))})
```


Der Dispatch erfolgt hier anhand zweier Typwerte. Mit `if`-Abfragen wäre das nicht annähernd so klar und deutlich programmierbar.

```
user=> (times 4 3)
12
user=> (times 4 "hi")
"hihihihi"
user=> (times "hi" 0)
""
```

Für einen OO-Programmierer bzw. eine OO-Programmiererin ist es ungewohnt, alle Implementierungen einer polymorphen Funktion – Methoden (*methods*) genannt – unter dem Dach der polymorphen Funktion zu fassen. In der OO-Welt werden die Methoden verteilt und den einzelnen Datenstrukturen, den Klassen zugeordnet. Diese datenzentrische Sicht sprengt Clojure absichtlich – so tat es schon das Common Lisp Object System (CLOS). Mit CLOS stand Lisp-Programmierer(innen) bereits in den 1980er Jahren ein mächtiges und modernes OO-System zur Verfügung, das seiner Zeit weit voraus war.

▷ **Aufgabe 10.2** Wie lassen sich variadische Funktionen (*variadic functions*) mit einer polymorphen Funktion umsetzen? Zur Erinnerung: Variadische Funktionen haben verschiedene Impementierungen für unterschiedliche Anzahlen an Argumenten.

10.3 Multimethoden

Ein fünfzeiliges Makro genügt, um den Grundstein zu legen für polymorphe Funktionen; vielmehr noch: für Multimethoden, die über eine frei konfigurierbare Dispatch-Funktion auf ein beliebiges Kriterium hin ausgewählt werden können. Das verallgemeinert den OO-Ansatz, der den Dispatch üblicherweise einzig anhand des Datentypen durchführt.

Clojure bringt von Haus aus eine Unterstützung für Multimethoden mit und verbirgt – im Gegensatz zu unserem Makro – den Gebrauch einer Map. Der Einsatz einer Map in unserem Makro ist motiviert aus der Anti-If-Kampagne und hat uns bis hin zu den Multimethoden getrieben. Streng genommen ist die Map ein Implementierungsdetail, das Clojure mit Hilfe der Makros `defmulti` und `defmethod` ausblendet.

Die Makros `defmulti` und `defmethod` verhalten sich genau wie unser Makro. Mit `defmulti` wird eine neue Multimethode unter einem Namen und mit der Angabe einer Dispatch-Funktion „angemeldet“. Mit `defmethod` werden die einzelnen Methoden erzeugt. Es muss der Name der Multimethode angegeben werden, der Dispatch-Wert, die Argumente und der Funktionsrumpf.

Das folgende Beispiel ist so gut wie selbsterklärend und wird Ihnen den Bezug zu unserem Makro sofort herstellen. Setzen wir die letzte Realisierung der Multimethode `area` mit `defmulti` und `defmethod` um.

```
user=> (defmulti area type-dispatch)
#'user/area
user=> area
#<MultiFn clojure.lang.MultiFn@7e5619>
```

Multimethoden sind in Clojure als eigener Datentyp implementiert, was Ihnen auch ein `(type area)` verrät.

```
user=> (defmethod area :circle [self] (* (self :radius) (self :radius) Mat
#<MultiFn clojure.lang.MultiFn@7e5619>
user=> (defmethod area :rectangle [self] (* (self :a) (self :b)))
#<MultiFn clojure.lang.MultiFn@7e5619>
```

Die Definitionen der Methoden müssen den Namen der Multimethode ausweisen, was sozusagen die gemeinsame Map identifiziert. Man kann in der REPL auch an der Ausgabe zu den Rückgabewerten erkennen, dass Clojure nach jedem `defmethod` ein und die selbe Multimethode zurück gibt.

```
user=> (area shape1)
12.566370614359172
user=> (area shape2)
6
```

Der Aufruf der `area`-Multimethode ist unverändert.

▷ **Aufgabe 10.3** Setzen Sie das Beispiel der Multimethode `times` mit `defmulti` und `defmethod` um.

Eine Default-Methode kann in einer Methode mit dem standardmäßigen Dispatch-Wert `:default` angelegt werden. Ein alternativer Dispatch-Wert für den Default-Dispatch kann in `defmulti` angegeben werden: nach dem Argument `:default` ist der gewünschte Dispatch-Wert anzugeben, siehe `(doc defmulti)`.

10.4 Hierarchien

Clojure ergänzt die Multimethoden um ein pfiffiges Feature: Es berücksichtigt beim Dispatch eine Hierarchie von Schlüsselwerten. Damit kann man Vererbungsbezüge im Sinne der Objekt-Orientierung inklusive Mehrfachvererbung nachbilden.

Clojure pflegt eine globale Hierarchie, in der die Klassen der zugrunde liegenden Java-Implementierung von Clojure abgebildet sind samt Vererbungsbezügen. So ist z.B. eine `Long`-Ganzzahl eine Zahl (`Number`), ebenso wie ein `Double` eine `Number` ist. Das Prädikat `isa?` verrät solche Hierarchie-Beziehungen.

```
user=> (type 4)
java.lang.Long
user=> Long
java.lang.Long
user=> (isa? Long Number)
true
user=> (isa? Double Number)
true
```

Diese Hierarchiebeziehungen werden beim Dispatch berücksichtigt. Ist kein Dispatch-Wert für beispielsweise `Integer` vorhanden, dann wird gemäß Eintrag in der globalen Hierarchie zusätzlich geschaut, ob es einen Dispatch-Wert für `Number` gibt. So kann die `times`-Funktion (siehe oben) im Multiplikationsfall generell für die Arbeit mit Zahlen konfiguriert werden und nicht ausschließlich für `Long`.

```
(defmulti times (fn [x y] [(type x) (type y)]))
(defmethod times [Number Number] [x y] (* x y))
```

```
(defmethod times [Long String] [n s] (reduce str (repeat n s)))
(defmethod times [String Long] [s n] (times n s))
```

Auch beim Mehrfach-Dispatch, wie hier auf zwei Werte, werden die einzelnen Werte in der Hierarchie abgesucht.

Die Hierarchie ist in einer Map gespeichert. In Ergänzung zu den Java-Klassen, können beliebige Beziehungen mittels `derive` hinzugefügt werden. Es muss sich dabei jedoch um qualifizierte Schlüsselwörter (*keywords*) oder um qualifizierte Symbole handeln. Mit dem Reader-Makro `::` erspart Clojure einem die Ergänzung um den aktuellen Namensraum; z.B. löst der Reader ein `::x` auf zu `:user/x`.

Zum Beispiel fügen die folgenden Aufrufe der globalen Hierarchie-Map die Abbildungen `::human` zu `::creature` und `::student` zu `::human` hinzu; `derive` ist eine Funktion mit einem Seiteneffekt.

```
user=> (derive ::human ::creature)
nil
user=> (derive ::student ::human)
nil
```

Die `isa?`-Abfrage erkennt einen `::student` sowohl als `::human` wie auch als `::creature`, aber z.B. nicht als `String`.

```
user=> (isa? ::student ::human)
true
user=> (isa? ::student ::creature)
true
user=> (isa? ::student String)
false
```

Die globale Hierarchie kann zu jedem Wert wie folgt befragt werden: Wer sind die unmittelbaren Vorgänger, in Informatiksprech „Eltern“ (`parents`) in der Hierarchie? Wer sind alle Vorfahren (`ancestors`) und wer alle Nachfahren (`descendants`) eines Wertes in der Hierarchie. Alle drei Fälle liefern ihre Ergebnisse als Mengen (*sets*).

```
user=> (parents ::student)
#{:user/human}
user=> (ancestors ::student)
#{:user/creature :user/human}
user=> (descendants ::creature)
#{:user/student :user/human}
```

Diese Flexibilität in der Beziehung von Hierarchiewerten kann Mehrdeutigkeiten erzeugen, die der Dispatch nicht auflösen kann. Solche Fälle treten auf, wenn es mehrere Pfade zu Vorgängern gibt.

```
user=> (derive ::student ::consumer)
nil
```

Ein `::student` ist sowohl `::human` als auch `::consumer`, womit der Dispatch im Beispiel keine eindeutige Wahl mehr treffen kann.

```
(defmulti youare :role)
(defmethod youare ::consumer [obj] "You are a consumer")
(defmethod youare ::human [obj] "You are a human")
```

Beachten Sie das Keyword `:role` in `defmulti`. Eigentlich sollte dort eine Dispatch-Funktion stehen. Tatsächlich fungiert ein Keyword in einem Programmausdruck als Funktion, das den assoziierten Wert zu einer übergebenen Map abfragt:

```
user=> (:role {:role ::human})
:user/human
```

Mit diesem Wissen ist klar, wie die folgenden Aufrufe von `youare` funktionieren.

```
user=> (youare {:role ::consumer})
"You are a consumer"
user=> (youare {:role ::student})
IllegalArgumentException Multiple methods in multimethod 'youare' match ..
```

Ein Dispatch auf `::consumer` bzw. `::human` ist eindeutig. Für `::student` ist er es nicht mehr, worüber sich Clojure in Form einer Exception beschwert.

Mit `prefer-method` kann der Konflikt durch Angabe einer Präferenz aufgelöst werden.

```
user=> (prefer-method youare ::human ::consumer)
#<MultiFn clojure.lang.MultiFn@11d2572>
user=> (youare {:role ::student})
"You are a human"
```

Man kann auch einen Dispatchwert mittels `remove-method` aus einer Multimethode entfernen. Damit lässt sich der Dispatch ebenfalls wieder eindeutig machen.

```
user=> (remove-method youare ::human)
#<MultiFn clojure.lang.MultiFn@11d2572>
user=> (youare {:role ::student})
"You are a consumer"
```

Bislang haben wir alle Beziehungen in der globalen Hierarchie abgebildet. Mit der Funktion `make-hierarchy` ist ohne weitere Argumente eine eigenständige Hierarchie erzeugbar, die nicht von der globalen Hierarchie abhängt. Die Funktionen `derive`, `isa?` etc. erlauben die explizite Übergabe einer Hierarchie als erstem Argument.

Lokale Hierarchien sind immutabel – sie werden über Maps realisiert. Nach einem `make-hierarchy` muss die erzeugte Hierarchie z.B. einem `derive` übergeben werden. Ein weiteres `derive` benötigt die Hierarchie aus dem vorhergehenden `derive`. Hierfür bietet der sogenannte Thread-Operator, das Makro `->`, eine angenehme Hilfe an: er reicht Ergebnisse zum jeweils nächsten Ausdruck als ersten Argumentwert weiter; siehe auch `(doc ->)`.

```
(def organization
  (-> (make-hierarchy)
      (derive ::dean ::professor)
      (derive ::vice-dean ::professor)))
```

Multimethoden verwenden die globale Hierarchie. Die Verwendung einer lokalen Hierarchie erfordert ein optionales `:hierarchy`-Keyword in der `defmulti`-Form, gefolgt von der lokalen Hierarchie.

10.5 Diskussion

In der Datenstruktur einer Map ist das Sprachkonstrukt der Entscheidung – gewöhnlich durch `if` repräsentiert – und das Prinzip der Gleichheit (`=`) implizit untergebracht. Es ist vollkommen beliebig, was man als gegeben ansieht: Man kann mit `if`, dem Vergleich und einer einfachen zusammengesetzten Datenstruktur eine Map implementieren. Oder man nimmt eine Map als gegeben an und „extrahiert“ daraus die der Map innewohnenden Konzepte von Entscheidung und Vergleich. Genau diesen Weg haben wir in diesem Kapitel nachgezeichnet.

Von der Map haben wir uns darüber hinaus inspieren lassen zu einer Verallgemeinerung des einfachen `if`. Ein `if` kann nur zwei Möglichkeiten einer Entscheidung abbilden. Das ist der elementarste Fall der Entscheidung. Eine Map erlaubt die Berücksichtigung vieler Möglichkeiten zur Entscheidung. Die konsequente Übertragung der Arbeitsweise eines `if` unter Zuhilfenahme einer Map hat uns zur polymorphen Funktion und schlußendlich zur Mutimethode geführt. Der Zugewinn an Ausdrucksstärke ist beachtlich. Wie aus dem Nichts entsteht eine Möglichkeit der Modellierung, die die Konzepte der Objekt-Orientierung einschließt aber Freiheitsgrade eröffnet, die sich aus der freien Konfiguration der Dispatch-Funktion ergeben.

Man kann den gleichen Herleitungsweg auch mit Vektoren als grundlegende zusammengesetzte Datenstruktur gehen. Die Dispatch-Funktion muss dann, beginnend mit Null, auf eine direkte Folge von Zahlen abbilden. Im einfachsten Fall lassen sich damit variadische Funktionen nachbilden. Angenommen, es gäbe ein Makro `defnvari` („define variadic function“). Mit diesem Makro ist die `plus`-Funktion (in Nachbildung von `+`) für verschiedene Anzahlen von Argumenten leicht zu definieren.

```
(defnvari plus (fn [& args] (count args))
  [(fn [] 0) ; index is 0
   (fn [x] x) ; index is 1
   (fn [x y] (+ x y))] ; index is 2
   (fn [x y & more] (apply plus (plus x y) more))) ; default function
```

Abhängig von der Anzahl der Argumente wird entweder die erste Funktion im Vektor (Index 0), die zweite oder dritte (Index 1 und 2) ausgeführt. Ansonsten greift die außerdem übergebene Default-Funktion. Die `plus`-Funktion arbeitet genauso wie `+`.

```
user=> (plus)
0
user=> (plus 2)
2
user=> (plus 2 3)
5
user=> (plus 2 3 4 5 6)
20
```

▷ **Aufgabe 10.4** Definieren Sie das Makro `defnvari`; verwenden Sie dabei `nth`.

Im allgemeinen Fall ist es schwierig, ohne `if`-Formen in der Dispatch-Funktion eine Abbildung auf eine Zahlenfolge zu bewerkstelligen. Und

damit wäre der Versuch gescheitert, per Vektor ein `if` überflüssig zu machen. Um das zu vermeiden, muss man Dispatch-Funktionen zur Bestimmung von sogenannten Hash-Werten finden – und bildet damit im Grunde die Funktionsweise einer Map nach. Insbesondere große Maps werden intern mit Hash-Funktionen implementiert, um eine effiziente und konstant performante Ermittlung des assoziierten Werts zu gewährleisten.

Die Anti-If-Kampagne sollte Ihnen eines gezeigt haben: Versuchen Sie, Ihre Programmwelt vermehrt und kreativ aus der Perspektive von Multimethoden zu betrachten. In vielen Fällen ist das ein Gewinn für Ihren Programmcode. Sie gewinnen an Flexibilität und Erweiterbarkeit. In der Objekt-Orientierung geschulte Programmierer(innen) werden alle Möglichkeiten der Objekt-Orientierung wiederfinden und nutzen können. Mit Multimethoden können Sie jedoch noch mehr machen.

Allerdings – auch das haben wir gezeigt – ist es nicht sinnig, auf ein `if` gänzlich zu verzichten. Es ist ebenso nützlich wie die daraus abgeleiteten Formen wie z.B. `cond` und `when`.

10.6 Lösungen

Aufgabe 10.1 Mit Hilfe einer Map liest sich die Funktion `NOT` als programmatische Beschreibung wie Klartext.

```
user=> (defn NOT [x] ({false true nil true} x false))
#'user/NOT
```

Das Verhalten ist wie erwartet. Machen Sie den Test, indem Sie statt `NOT` die Funktion `not` verwenden.

```
user=> (NOT nil)
true
user=> (NOT 3)
false
```

Aufgabe 10.2 Mit einem Dispatch auf die Anzahl der übergebenen Argumente sind variadische Funktionen realisierbar. Die Dispatch-Funktion gestaltet sich wie folgt:

```
(fn [& args] (count args))
```

Aufgabe 10.3 Die Lösung erfordert praktisch nur syntaktische Anpassungen:

```
(defmulti times (fn [x y] [(type x) (type y)]))
(defmethod times [java.lang.Long java.lang.Long] [x y] (* x y))
(defmethod times [java.lang.Long java.lang.String] [n s] (reduce str (repe
(defmethod times [java.lang.String java.lang.Long] [s n] (times n s))
```

Dass Verhalten ist wie zuvor:

```
user=> (times 4 3)
12
user=> (times 4 "hi")
"hihihihi"
user=> (times "hi" 0)
""
```

Aufgabe 10.4 Das Makro ist eine geringfügige Anpassung des Makros `defnpoly`.

```
(defmacro defnvari
  ([name dispatch-fn coll] `(defnvari ~name ~dispatch-fn ~coll nil))
  ([name dispatch-fn coll default-fn]
    `(defn ~name [& args#]
       (apply (nth ~coll (apply ~dispatch-fn args#) ~default-fn) args#))))
```

Historie

16. Mai 2012 Rohfassung fertig

4. Mai 2012 Schreibbeginn

11 Nebenläufige Programmierung

Lange Zeit wurden Rechner dadurch leistungsfähiger, indem man sie schneller machte. Mittlerweile geraten die Hardware-Entwickler an die Grenzen des Machbaren. Schneller geht es mit den heutigen Technologien kaum mehr. Die Leistungszuwächse moderner Rechner begründen sich auf zunehmende Parallelisierung. Prozessoren kommen mit mehreren parallel arbeitenden Kernen (*cores*) daher. Hochleistungsrechner entfesseln ihre Rechenkraft im Verbund (*cluster*), der oft Tausende Prozessoren umfasst. Ein weiterer Trend ist die Verteilung von Rechenleistungen auf mehrere Rechner in einer netzbasieten Daten- und Dienstleistungsinfrastruktur (*cloud*).

Mit dieser Entwicklung haben sich verschiedene Begrifflichkeiten eingebürgert.

Nebenläufige Programmierung : Die nebenläufige Programmierung (*concurrent programming*) hat Programme im Blick, die weitgehend unabhängig voneinander arbeiten, sich jedoch die Ressourcen eines Rechners teilen müssen wie z.B. Rechenleistung, Speicher oder Netzzugang.

Parallele Programmierung : Die parallele Programmierung (*parallel programming*) befasst sich damit, eine Aufgabe in viele, möglichst unabhängige Teilaufgaben zu zerlegen, die dann auf viele parallele Rechenprozesse verteilt bearbeitet werden können. Die Zusammenführung der Teilergebnisse zu einem Gesamtergebnis gehört ebenfalls dazu. Die parallele Programmierung will parallele Rechenleistung nutzbar machen, um eine Aufgabe schneller oder genauer durchführen zu können.

Verteiltes Rechnen : Die nebenläufige und parallele Programmierung geht von verlustfreier und zuverlässiger Kommunikation zwischen Programmen aus. Sind miteinander kommunizierende Rechenprozesse in einem Netz verteilt, so kann die Kommunikation ungewollt abbrechen, sie ist unzuverlässig und die Übertragung von Nachrichten kostet Zeit. Das verteilte Rechnen (*distributed computing*) versucht den Aspekt der Verteilung einerseits zwar transparent zu machen, muss sich andererseits aber grundsätzlich mit unterbrochener Kommunikation und den zeitlichen Latenzen von Kommunikation auseinander setzen.

Die Grenzen zwischen diesen Themengebieten sind fließend, zumal das eine Teilgebiet das andere umfasst. Parallele Programmierung basiert auf Nebenläufigkeit und verteilte Programmierung ist zu einem guten Teil auch parallele Programmierung. In diesem Kapitel setzten wir uns mit der nebenläufigen Programmierung auseinander.

Zuvor noch eine weitere Begrifflichkeit: Bei der nebenläufigen Programmierung wird oft zwischen Threads und Prozessen unterschieden. Unter einem Prozess ist oft der unabhängige Ausführungsstrang gemeint, den das Betriebssystem zur Verfügung stellt, um Programme bzw. Applikationen unabhängig voneinander ausführen zu können. Threads meinen die nebenläufige Ausführung innerhalb eines Prozesses. Die Ausführungsverwaltung von Threads ist wesentlich günstiger als die von Prozessen. Das

Starten, Verwalten und Beenden eines Prozesses kostet wesentlich mehr Ressourcen, als das Starten, Verwalten und Beenden eines Threads. Ohne ein anwendungsorientiertes Betriebssystem macht die Unterscheidung von Threads und Prozessen wenig Sinn.

Wir beschäftigen uns in diesem Kapitel mit der Nebenläufigkeit durch Threads, die in einem Clojure-Prozess z.B. unter Windows, Linux oder OS X laufen.

11.1 Funktionale Betrachtung von Interaktion

Die funktionale Programmierung basiert auf zwei Grundprinzipien:

- Werte, gemeint sind sowohl primitive als auch zusammengesetzte Daten, sind unveränderlich (*immutable*). Oder anders ausgedrückt: Werte ändern sich nicht über die Zeit.
- Eine Funktion liefert bei gleichen Eingangswerten immer den gleichen Ausgangswert zurück. Das Ergebnis der Anwendung einer Funktion ist unabhängig vom Zeitpunkt ihrer Anwendung; man nennt das „referentielle Transparenz“ (*referential transparency*).

Ein funktionales Programm ist nicht mehr als eine große Funktion, die ihrerseits aus Funktionen zusammengesetzt (komponiert) ist. Die strikte Unabhängigkeit aller Funktionen von der Zeit, ihr eindeutiges Verhalten und ein durchgängiges Kompositionsprinzip machen es relativ einfach, funktionale Programme zu analysieren und zu verstehen. Das ist ein großer Vorteil der funktionalen Programmierung, der viel zu ihrer zunehmenden Popularität beiträgt. Doch wie ist es um die Abbildung zeitlicher Abläufe bestellt?

Prinzipiell kann der Aspekt der Zeit als zusätzliches Argument zu einer Funktion modelliert werden. Die zeitlose Funktion $f(x)$ kann die Zeit t ausdrücklich berücksichtigen durch $f(x, t)$ – Sie haben in der Schule auf diese oder ähnliche Weise zeitliche Vorgänge in der Physik abgebildet. Nur löst uns dieser Ansatz ein grundlegendes Problem nicht: das der Interaktion.

Interaktion bedeutet ein Wechselspiel von Ein- und Ausgaben, sprich von Funktionsaufrufen und Funktionsergebnissen. Interaktion erfordert eine Unterbrechung einer Funktion. Die explizite Zerlegung einer Funktion in Teilfunktionen ist eine Voraussetzung für die Modellierung interaktiver Vorgänge.

Machen wir es an einem Beispiel konkret. Eine Folge von Eingaben sei dargestellt durch die Sequenz $[i_1, i_2, i_3]$. Die Anwendung der Funktion f auf diese Folge, $f([i_1, i_2, i_3])$, muss nun zerlegt werden in Teilfunktionen f_1, f_2 und f_3, die jeweils eine Teileingabe verarbeiten. Es gilt:

$$f([i_1, i_2, i_3]) = f_3(f_2(f_1(i_1), i_2), i_3)$$

Die rechte Gleichungsseite modelliert den zeitlichen Ablauf über Teilanwendungen. Diese Teilanwendungen entsprechen den Ausgaben: $o_1 = f_1(i_1)$, $o_2 = f_2(o_1, i_2)$ und $o_3 = f_3(o_2, i_3)$.

Es fällt auf, dass f_1 eine einwertige Funktion ist, d.h. nur ein Argument entgegen nimmt, während f_2 und f_3 zweiwertig sind. Durch Einführung eines Initialwertes o_0 kann die Ausnahmestellung von f_1 vermieden werden.

$$f([o_0, i_1, i_2, i_3]) = f_3(f_2(f_1(o_0, i_1), i_2), i_3)$$

Wenn wir die Interaktion von drei auf beliebig viele Interaktionen $n > 0$ ausweiten, ergibt sich

$$o_n = f_n(o_{n-1}, i_n) \tag{11.1}$$

Der Ansatz von Clojure zur Nebenläufigkeit basiert genau auf dieser kleinen Formel. Die Formel beruht auf der Idee, dass eine Funktion f zum Zwecke der Interaktion in eine möglicherweise unendliche Anzahl von Teilfunktionen $f_1, \ldots, f_n$ zerlegt wird. Die Teilfunktionen bilden Interaktionspunkte zu den diskreten Zeitpunkten $1, 2, 3, \ldots n$ ab. Man bezeichnet die Teilfunktionen auch als Aktionen (*actions*). Obige Formel 11.1 zeigt die jeweiligen Ausgaben o_n als Ergebnis einer Eingabe i_n in Abhängigkeit von der unmittelbaren Vergangenheit o_{n-1}.

Der Wert o_{n-1} steht für den über die Aktionen f_1 bis f_{n-1} erreichten Rechenfortschritt in Abarbeitung der Gesamtfunktion f. Der zum Zeitpunkt der Anwendung der Aktion f_n aktuelle Wert o_{n-1} repräsentiert den erreichten Zustand (*state*) aufgrund der bisherigen Interaktionen. Die Modellierung der Interaktion bündelt sich in der Folge der Ergebniswerte bzw. Zustandswerte als Rechenfortschritt der Funktion f. Die Funktion f ist der gemeinsame Kontext.

Plötzlich gibt es ein Problem. Werte wie o_{n-1} oder o_n sind immutabel, unveränderlich. Aber eigentlich sind die Übergänge von o_{n-1} zu o_n ein Ausdruck des Wechsels eines Wertes, nämlich o. Mit dem Fortschreiten der Zeit wird o immer wieder aktualisiert. Faktisch repräsentiert o eine Identität, deren Zustand sich im Laufe der Zeit ändert. Die „Mutation“ einer Identität von einem Zustandwert zum anderen realisiert einen mutablen Zustand (*mutable state*).

Clojure führt für solche Identitäten einen eigenen Datentypen ein, sogenannte Referenztypen (*reference types*), die einen Datenwert referenzieren. Die Referenztypen heißen Var, Ref, Atom und Agent. Die vier Referenztypen stehen für vier verschiedene Arten des Umgangs mit Nebenläufigkeit.

Ich verwende die Begriffe „Referenz“ und „Identität“ synonym. Ich bevorzuge jedoch den Begriff der „Referenz“, da er deutlicher macht, worum es geht.

11.2 Der Agent-Referenztyp

Wenn wir mehrere Funktionen haben, die miteinander in Interaktion treten, macht dies nur Sinn, wenn die Funktionen unabhängig, d.h. nebenläufig ihren Rechenfortschritt verfolgen können. Da sich die Interaktionen in den Zuständen von Identitäten bündeln, ist vor allem die konkurrierende Mutation von Identitäten im Zentrum des Interesses.

11.2.1 Agenten im Einsatz

Eine Referenz (auch „Identität“) vom Typ eines Agenten setzt man mit der Funktion `agent` zusammen mit einem Initialwert auf; der Initialwert entspricht o_0, siehe Gleichung 11.1.

```
user=> (agent 0)
#<Agent@5bd6fbb3: 0>
```

Der Agent verweist auf den Wert `0`. Man sagt auch: Der Agent referenziert den Wert `0`. Um mit dem Agenten dauerhaft arbeiten zu können, ist eine Definition hilfreich.

```
(def counter (agent 0))
```

Der Zugriff auf den über den `counter`-Agenten referenzierten Wert ist über die Dereferenzierungsfunktion `deref` gegeben. Kürzer und gebräuchlich ist die Verwendung des Readermakros `@`.

```
user=> (deref counter)
0
user=> @counter
0
```

Soll der Agent auf einen neuen Wert verweisen, muss die Interaktion – wir haben es in der Einführung gelernt, siehe Formel 11.1 – über eine Funktion (Aktion genannt) geschehen, die den neu zu referenzierenden Wert berechnet. Mit der Aktion muss der „alte“, also der aktuell referenzierte Wert übergeben werden. Den aktuellen Referenzwert setzt Clojure selbstständig ein. Weitere Argumente als Input zur Interaktion dürfen folgen, was von der verwendeten Aktionsfunktion abhängt. Das Ergebnis der Funktionsanwendung ist der neue vom Agenten referenzierte Wert.

Diese Infrastruktur bietet die Funktion `send` an. Als erstes Argument folgt der Agent, als zweites die Aktion, dann gedacht der dereferenzierte Agentenwert, und es mögen weitere Argumentwerte folgen. Die Funktion `send` liefert als Rückgabe den Agenten zurück.

Ein Beispiel: Wir wollen den über `counter` referenzierten Wert um Eins erhöhen. Im Prinzip ist folgende Rechnung durchzuführen: `(+ @counter 1)`. Den referenzierten Wert von `counter`, `@counter`, setzt Clojure automatisch ein; das Ganze erinnert sehr an die `apply`-Funktion.

```
user=> (send counter + 1)
#<Agent@79f03d7: 1>
user=> @counter
1
```

Bachten Sie, dass `send` einen Agenten erwartet! Natürlich können wir auch die `inc`-Funktion verwenden.

```
user=> (send counter inc)
#<Agent@79f03d7: 1>
user=> @counter
2
```

Lassen Sie sich nicht von der Ausgabe in der Console irritieren. Es wird Ihnen manchmal passieren, wie mir an dieser Stelle, dass der Agent nach dem Aufruf von `send` noch den „alten“ Wert `1` zu haben scheint. Tatsächlich hat er den Wert `2`, wie die Abfrage von `@counter` zeigt. Wir sehen hier einen typischen Effekt von gleichzeitiger Programmausführung, besonders, wenn Ein- und Ausgaben eine Rolle spielen. Es kann passieren, dass die REPL noch den „alten“ Wert ausgibt, während im Hintergrund gleichzeitig das Update der Referenz läuft.

Agenten realisieren echte Nebenläufigkeit. Das ist das Besondere an Agenten! Wenn Sie eine Aktion mit `send` an einen Agenten übergeben, so

wird diese Aktion in einem mit dem Agenten assoziierten Thread ausgeführt. Alle mit `send` abgesetzten Aktionen wandern in der Reihenfolge des Versands in eine Warteschlange (*queue*) und werden nacheinander abgearbeitet.

Der Aufrufer bekommt die Ausführungskontrolle in seinem eigenen Thread unmittelbar nach Aufruf der `send`-Funktion zurück. Man spricht von „asynchroner Kommunikation“. Da die Funktionen `+` bzw. `inc` so schnell arbeiten, merken Sie von der Nebenläufigkeit praktisch nichts.

Führen wir zu Anschauungszwecken eine Funktion `pause` ein, um Zeit verstreichen zu lassen, bevor eine Aktion zur Anwendung kommt. In einem Funktionsrumpf werden bei mehreren Formen alle Formen hintereinander ausgeführt, wobei die letzte Form den Rückgabewert der Funktion bestimmt. Der Zugriff auf `sleep` via `Thread` ist Java-spezifisch.

```
(defn pause [agt msec f & args]
  (Thread/sleep msec) (apply f agt args))
```

Verzögern wir das Update des `counter`-Agenten um 8 Sekunden (8000 Millisekunden). Fragen Sie den Wert `@counter` kurz nach dem `send` ab. Warten Sie ein wenig, und Sie werden feststellen, dass `@counter` plötzlich einen neuen Wert hat.

```
user=> (send counter pause 8000 inc)
#<Agent@79f03d7: 2>
user=> @counter
2
user=> ; Warten Sie ein paar Sekunden
user=> @counter
3
```

Mit `await` können Sie auf die Abarbeitung der an einen Agenten geschickten Aktionen warten. Damit wird die asynchrone Interaktion mit dem Agenten sozusagen nachträglich synchron gemacht. An `await` können beliebig viele Agenten übergeben werden; `await` gibt immer `nil` zurück.

Schicken Sie zwei Pausen mit einer `inc`-Aktion an `counter`. Mit `await` blockiert der REPL-Thread solange, bis der `counter`-Agent nichts mehr zu tun hat.

```
user=> (send counter pause 8000 inc)
#<Agent@79f03d7: 3>
user=> (send counter pause 8000 inc)
#<Agent@79f03d7: 3>
user=> (await counter) ; die REPL blockiert
nil
user=> @counter
5
```

Mit `await-for` kann das Warten zeitlich beschränkt werden. Die Dauer bis zur Zeitüberschreitung (*timeout*) wird in Millisekunden angegeben. Überschreitet der Agent bzw. überschreiten die Agenten die Wartezeit, gibt `await-for` den Wert `false` zurück.[1]

```
user=> (send counter pause 8000 inc)
#<Agent@79f03d7: 5>
```

[1] Laut Online-Dokumentation zu Clojure sollte `nil` der Rückgabewert sein. Ein Fehler in der Doku.

```
user=> (await-for 1000 counter)
false
```

Sorgt der Agent für eine vorzeitige Beendigung, ist der Rückgabewert `true`. Beachten Sie, dass `await-for` jetzt einen Timeout von 9 Sekunden hat.

```
user=> (send counter pause 8000 inc)
#<Agent@79f03d7: 6>
user=> (await-for 9000 counter)
true
```

Neben `send` gibt es `send-off`, um einen Agenten mit Aktionen in die Pflicht zu nehmen; `send-off` wird genauso wie `send` aufgerufen. Der Unterschied liegt in der Anzahl der zur Verfügung stehenden Threads.

Alle mit `send` initiierten Aktionen werden durch eine begrenzte Anzahl an Threads aus einem Threadpool abgearbeitet. Die maximale Zahl an Threads im Pool ist an die gegebene Hardware angepasst. Die Größe des Threadpools begrenzt die Anzahl nebenläufiger Aktivitäten. Überzählige Aktionen müssen auf einen frei werdenden Thread warten.

Für `send-off` gilt diese Beschränkung nicht. Jede mit `send-off` initiierte Aktion bekommt dynamisch einen neuen Thread zugewiesen. Es lassen sich soviele Threads aufsetzen, wie es die Ressourcen der Laufzeitumgebung erlauben.

Die Threads aus dem Threadpools sind effizient für die JVM zu managen. Threads außerhalb des Pools sind „teurer“ aufzusetzen und zu beenden. Als Daumenregel gilt: Für kurze Aktionen sollte man `send` verwenden; Threads aus dem Threadpools sind schnell aufgesetzt und wieder frei gegeben, sie sind auf einen hohen Durchsatz an Aktionen ausgelegt. Langwierige Aktionen und Aktionen, die z.B. aufgrund von Ein- bzw. Ausgaben blockieren, sind mit `send-off` besser initiiert; mit `send` liefe der Threadpool Gefahr, keine freien Threads mehr zu haben und sämtliche ausstehenden Aktionen „verhungern“ zu lassen.

Jede Aktion hat innerhalb ihres Threads mittels `*agent*` Kenntnis über den zu ihr gehörigen Agenten. Damit kann eine Aktion eine Aktion an den eigenen Agenten schicken.

11.2.2 Wenn Agenten Fehler unterlaufen

Tritt während einer Aktion eine Ausnahmebdingung (*exception*) auf, muss sich ein Agent selber darum kümmern; Fehler innerhalb eines Threads sind eine interne Angelegenheit des Agenten.

Es gibt zwei Umgangsmodi, die festlegen, wie ein Agent mit Ausnahmebedingungen umgeht: `:fail` und `:continue`. Der aktuelle Umgangsmodus kann mit `error-mode` erfragt werden. In der Grundeinstellung ist der Fehlermodus `:fail`.

```
user=> (error-mode counter)
:fail
```

Man kann den Fehlermodus ausdrücklich setzen mit `set-error-mode!`.

```
user=> (set-error-mode! counter :fail)
nil
```

`:fail` : Im Fall einer Ausnahmebedingung beendet der Agent die Aktion, wechselt in den `:fail`-Modus und führt keine weiteren Aktionen aus; noch ausstehenden Aktionen bleiben in der Warteschleife erhalten. Jeder jetzt folgende Aufruf eines `send` bzw. `send-off` an den Agenten führt zu einer `RuntimeExeption` beim Aufrufer. Der Grund für die Exception kann mit `agent-error` abgefragt werden. Per `restart-agent` kann der Agent seine Arbeit wieder aufnehmen. Abhängig vom Parameter `:clear-actions` werden die Aktionen in der Warteschleife gelöscht (`true`) bzw. ihre Abarbeitung fortgesetzt (die Grundeinstellung `false`).

Das folgende Szenario veranschaulicht die Beschreibung. Wir setzten eine zeitverzögerte Division durch Null auf, die nach 8 Sekunden eine Exception erzeugt, schicken unmittelbar ein `inc` hinterher und warten auf den Ablauf der Pause. Ein weiteres an den Agenten gesendetes `inc` führt zur Exception. (Ebenso würde ein `await` zu einer Exception führen.)

```
user=> (def counter (agent 10))
#'user/counter
user=> (send counter pause 8000 / 0)
#<Agent@76996cca: 10>
user=> (send counter inc)
#<Agent@76996cca: 10>
user=> ; Warten Sie 8 Sekunden
user=> (send counter inc)
java.lang.RuntimeException: Agent is failed, needs restart (NO_SOURCE_FILE
user=> (agent-error counter)
#<ArithmeticException java.lang.ArithmeticException: Divide by zero>
```

Der Wert des Agenten hat sich nicht geändert, wohl hängt noch das erste `inc` in der Warteschleife. Der Restart erwartet neben dem Agenten einen Restart-Wert und optional das Schlüsselwort `:clear-actions` mit dem Wert `true` oder `false`; ohne Angabe ist der Defaultwert `false`, man könnte die Angabe nachfolgend auch weglassen.

```
user=> @counter
10
user=> (restart-agent counter @counter :clear-actions false)
10
user=> @counter
11
```

`:continue` : In diesem Umgangsmodus mit Fehlern bricht der Agent die laufende Aktion im Fall einer Ausnahmebedingung ab, so als hätte sie nie stattgefunden. Falls gesetzt, ruft der Agent eine Fehlerbehandlungsroutine auf. Der Agent setzt die Abarbeitung weiterer Aktionen aus der Warteschleife fort.

Hier folgt ein zum `:fail`-Betriebsmodus entsprechendes Szenario.

```
user=> (def counter (agent 10))
#'user/counter
user=> (set-error-mode! counter :continue)
nil
user=> (send counter pause 8000 / 0)
#<Agent@6c91f005: 10>
user=> (send counter inc)
#<Agent@6c91f005: 10>
```



```
user=> ; Warten Sie 8 Sekunden
user=> @counter
11
```

In beiden Fehler-Betriebsmodi `:fail` wie auch `:continue` kann eine Fehlerbehandlungsroutine für den Agenten registriert werden, die zwei Argumente entgegen nimmt: den Agenten und den aufgetretenen Fehler. Mit dieser Routine können als Reaktion auf einen Fehler Seiteneffekte realisiert werden. Eine Möglichkeit ist z.B. das Aufzeichnen (*logging*) aufgetretener Fehler und/oder die Ausgabe einer Meldung.

```
user=> (def counter (agent 10))
#'user/counter
user=> (set-error-handler! counter
         (fn [agt err] (println "Note:" agt "reports" err)))
nil
user=> (set-error-mode! counter :continue)
nil
user=> (send counter / 0)
#<Agent@14718242: Note: 10>
#<Agent@14718242: 10> reports #<ArithmeticExceptionuser=>  java.lang.Arithm
xception: Divide by zero>
```

Was Sie an diesem Beispiel sehr schön demonstriert bekommen, ist, wie sich die Rückmeldung über die Konsole und die Print-Ausgabe auf der Konsole in die Quere kommen. Beide konkurrieren gleichzeitig um den `*out*`-Strom. Das Ergebnis ist kaum lesbar. Wenn Sie das Beispiel an Ihrem Rechner ausprobieren, sieht das Resultat des Wettstreits um den Ausgabestrom vermutlich anders aus.

Mit `error-handler` können Sie die registrierte Fehlerbehandlungsroutine erfragen.

```
user=> (error-handler counter)
#<user$eval15$fn__16 user$eval15$fn__16@643f96ee>
```

11.3 Der Atom-Referenztyp

Angenommen, Sie und ich stehen am Fenster und schauen auf die Straße. Wir sehen beide *gleichzeitig* ein Auto vorbeifahren. Welche Farbe hat es? Sie sagen, es ist rot. Ich sage, es ist grün. Da hat wohl jemand nicht richtig hingeguckt! Wir wiederholen das Experiment und es bleibt verwirrend. Unsere Beobachtungen zur Farbe könnten unterschiedlicher nicht sein.

Sie würden an meinem Geisteszustand zweifeln – und ich an Ihrem. Grund für die Unterstellung des Irrsinns ist, dass wir ein Grundprinzip nicht in Frage stellen:

> Gleichzeitige Beobachtungen müssen übereinstimmende Ergebnisse liefern!

Wir wissen zwar seit Einstein von der Relativität der Gleichzeitigkeit, aber das ist „höhere Physik“. Unsere Erfahrungen sind in der vereinfachten Weltsicht der Newton'schen Mechanik verhaftet. Wir dürfen – und müssen – uns darauf verlassen: *Gleichzeitige Beobachtungen müssen übereinstimmende Ergebnisse liefern.*

Dieses Prinzip ist bei nebenläufigen Vorgängen in Gefahr. Was, wenn zwei Beobachtungen und eine Änderung am „Gegenstand“ der Beobachtung

gleichzeitig stattfinden? Es drohen Inkonsistenzen bei den Beobachtungen.

Die Auswirkungen gleichzeitiger Vorgänge sind schwer vorzustellen. Ein Hilfskonstrukt ist, sich die „gleichzeitigen“ Ereignisse als Folge von Ereignissen zu denken und zwar in allen möglichen Varianten ihrer Abfolge. Die Varianten spielen die Möglichkeiten dessen durch, was als Ergebnis der Gleichzeitigkeit in Frage kommt. So kann es passieren, dass Sie ein rotes Auto sehen und ich ein grünes. Gleichzeitig wurde „dazwischen“ das Auto umgespritzt. Genauso ist es möglich, dass wir beide zusammen ein rotes Auto „vor“ dem Umspritzen oder zusammen ein grünes Autos „nach“ dem Umspritzen sehen; vorausgesetzt, dass das Auto in der Vergangenheit rot war. Welche dieser Beobachtungen wir machen ist mehr oder minder zufällig.

Diese Mehrdeutigkeiten der Auswirkungen von Gleichzeitigkeit sind eine schwerwiegende Verletzung des funktionalen Prinzips. Unter gleichen Bedingungen sollte eine Wiederholung gleichzeitig stattfindender Beobachtungen und Änderungen dieselben Beobachtungsergebnisse liefern.

Clojure löst diese Problematik mit dem Referenztyp des Atoms. Ein Atom wird zusammen mit seinem Initialwert mittels `atom` erzeugt.

```
user=> (def color (atom "red"))
#'user/color
user=> color
#<Atom@359ecd80: "red">
```

Eine Beobachtung, sprich das Lesen bzw. Dereferenzieren eines referenziertes Atom-Werts, kann jederzeit stattfinden. Da das Dereferenzieren nicht blockiert, ist das Lesen eine schnelle und effiziente Operation. Der referenzierte Wert wird über `deref` bzw. `@` dereferenziert.

```
user=> @color
"red"
```

Wird die Atom-Referenz jedoch auf einen neuen Wert gesetzt, so blockiert das Atom für den Moment der Änderung alle lesenden Zugriffe. Konkurrieren mehrere Änderungen mehr oder weniger gleichzeitig um das Atom, werden die Update-Aktionen auf eine einfache Art und Weise synchronisiert: Stellt eine Update-Aktion fest, dass sich der Referenzwert während des Updateversuchs geändert hat, unternimmt die Update-Aktion so viele Wiederholungsversuche, bis sie erfolgreich ist.

Die wichtigste Update-Funktion ist `swap!`; `swap!` arbeitet im Sinne von Formel 11.1 und erwartet Argumente entsprechend zu `send` bei Agenten: nach der Atomreferenz folgt die Aktionsfunktion, die implizit den dereferenzierten Atom-Wert übergeben bekommt einschließlich aller weiterer Argumente.

```
user=> (swap! color str "dish")
"reddish"
user=> @color
"reddish"
```

Die Funktion `str` liefert bei zwei oder mehr Zeichenketten eine aus den Argumenten zusammengesetzte (konkatenierte) Zeichenkette zurück.

Eine weitere Update-Funktion ist `reset!`, die die Atom-Referenz auf den übergebenen Wert setzt.

```
user=> (reset! color "blue")
"blue"
user=> @color
"blue"
```

Zu guter Letzt gibt es noch die Update-Funktion `compare-and-set!`. Entspricht der aktuelle Wert des Atoms dem übergebenen Vergleichwert, dann wird die Referenz auf den neuen Wert gesetzt. Die Funktion gibt `true` im Erfolgsfall zurück und `false` bei Misserfolg.

```
user=> (compare-and-set! color "blue" "bluish")
true
user=> (compare-and-set! color "red" "blue")
false
user=> @color
"bluish"
```

11.4 Der Ref-Referenztyp

Atom-Referenzen synchronisieren Änderungen an *einer* Referenz. Was, wenn die Abstimmung mehr als eine Referenz betrifft?

Das klassische Beispiel ist der Geldtransfer zwischen zwei Bankkonten. Angenommen, der Kontostand von Konto A sei 1.500 €, der von Konto B 750 €. Es erfolge eine Überweisung in Höhe von 500 € von Konto A auf Konto B. Zeitgleich zur Überweisung erfolge eine Abhebung von 1.500 € von Konto A.

Bei diesen Vorgängen *kann*, wenn die Konten nicht koordiniert sind, eine Menge schief laufen. Zum Beispiel: Da zu Beginn der Kontostand mit 1.500 € angegeben ist, erscheinen sowohl die Abhebung als auch die Überweisung Konto A nicht zu überziehen (1. Fehler). Die Abhebung setzt Konto A auf 0 €, was von der Überweisung mit 1.000 € überschrieben wird (2. Fehler). Während der laufenden Vorgänge ermitteln wir die Summe der auf den Konten vorhandenen Gelder; in der Summe fehlen die 500 €, da sie zwar schon von Konto A abgebucht, Konto B aber noch nicht gutgeschrieben sind (3. Fehler).

Die Probleme lassen sich beseitigen, wenn die Überweisung zwischen den Konten abgesichert abläuft und konkurrierende Änderungsvorgänge synchronisiert werden.

Clojure regelt koordinierte Änderungen über den Ref-Referenztypen zusammen mit einem Transaktionssystem namens *Software Transactional Memory* (STM). Die Referenzen können nur innerhalb einer Transaktion verändert werden.

Eine Ref wird mit `ref` erzeugt und in bekannter Manier mittels `deref` oder `@` dereferenziert.

```
user=> (def account (ref 200))
#'user/account
user=> @account
200
```

Eine Änderung des referenzierten Werts ist mit `alter` möglich und folgt der Logik von Formel 11.1, wie es auch `swap!` bei Atomen und `send` bei Agenten tun. Allerdings, und das ist ein entscheidender Unterschied, muss die Änderung einer Ref in einer Transaktion vorgenommen werden. Dafür

ist sie einfach innerhalb der `dosync`-Form einzubetten, die den Rahmen für den Beginn und das Ende der Transaktion setzt.

```
user=> (alter account + 500)
java.lang.IllegalStateException: No transaction running (NO_SOURCE_FILE:0)
user=> (dosync (alter account + 500))
700
user=> @account
700
```

Schauen wir uns das Beispiel an, während einer Überweisung eine Bestandsaufnahme über die Konten zu machen. Definieren wir zunächst zwei Konten und die zwei Funktionen „abheben“ (*withdraw*) und „einzahlen“ (*deposit*). Geben Sie den Code entweder über die Konsole oder in ein File ein.

```
(def accountA (ref 1500))
(def accountB (ref  750))

(defn withdraw [account amount]
  (alter account - amount))

(defn deposit [account amount]
  (alter account + amount))
```

▷ **Aufgabe 11.1** Was halten Sie von der folgenden Umsetzung der Funktion `deposit`?

```
(defn deposit [account amount] (withdraw account -amount))
```

Für eine Überweisung erstellen wir zwei Versionen. Die erste Version sichert die Abhebung und die Einzahlung innerhalb einer Transaktion mittels `dosync` ab, die zweite Version isoliert die Vorgänge in je einer Transaktion. Zwischen Abhebung und Einzahlung sorgt eine Pause dafür, dass wir genügend Zeit für eine parallele Inventarisierung über die Konsole haben. Als erstes Argument erwarten die `transfer`-Funktionen einen Agenten, mit Hilfe dessen eine Überweisung asynchron in einem Thread ausgeführt werden kann.

```
(defn transferSTM [agt from to amount]
  (dosync
    (withdraw from amount)
    (Thread/sleep 8000)
    (deposit to amount)
    agt))

(defn transfer [agt from to amount]
  (dosync (withdraw from amount))
  (Thread/sleep 8000)
  (dosync (deposit to amount))
  agt)
```

Wir setzen zuerst einen mit dem Wert `"Thread"` initialisierten Agenten auf.[2] An diesen Agenten senden wir `transferSTM` und führen unmittelbar danach eine Inventarisierung aus. Die Inventarisierung summiert die aktuellen Beträge über die Konten.

[2] Man kann einen Thread auf per `future` erzeugen, doch haben wir dieses Makro noch nicht eingeführt.

```
user=> (def thread (agent "Thread"))
#'user/thread
user=> (send thread transferSTM accountA accountB 500)
#<Agent@689e8c34: "Thread">
user=> (+ @accountA @accountB)
2250
```

Der Betrag ist korrekt. Warten Sie ein paar Sekunden, bis die Überweisung wirksam ist. Wenn Sie es ausprobieren wollen: Solange die Überweisung noch nicht abgeschlossen ist, liefern `@accountA` und `@accountB` noch die Werte `1500` bzw. `750`.

Wenn Sie eine Überweisung mit `transfer` absetzen mit anschließender Inventarisierung, verflüchtigt sich der Überweisungsbetrag für die Dauer der Pause.

```
user=> (send thread transfer accountA accountB 500)
#<Agent@261a6518: "Thread">
user=> (+ @accountA @accountB)
1750
user=> ; Warten Sie die Pause ab
user=> (+ @accountA @accountB)
2250
```

Zu Beginn einer Transaktion wird ein Schnappschuss über die Werte aller Ref-Referenzen angelegt, die in der Transaktion involviert sind. Diese Ausgangswerte der Refs merkt sich die Transaktion bis zu ihrem Ende. Im Laufe der Transaktion stattfindende Änderungen an Ref-Werten gelten nur lokal innerhalb der Transaktion. Jede Transaktion arbeitet isoliert (*isolated*) von anderen Transaktionen. Am Ende einer Transaktion sollen die lokal geänderten Ref-Werte „öffentlich“ gemacht werden – man spricht bei Transaktionen von einem „Commit“ (engl. für überlassen, übergeben). Das geschieht nur dann, wenn die Schnappschuss-Werte während der Transaktion von „außen“ nicht geändert wurden und immer noch ihre Ursprungswerte haben. Ist das nicht der Fall, hat die Transaktion mit mittlerweile überholten Ref-Werten gearbeitet. Sie legt einen neuen Schnappschuss an und wiederholt ihre Arbeit (*restart*). Dieser Prozess wird so oft durchgeführt, bis die Offenlegung (*commit*) der lokalen Ref-Änderungen nicht in Konflikt steht zu mittlerweile geänderten Ausgangswerten.

Der Commit, die „Veröffentlichung“ der neuen Ref-Werte einer Transaktion, wird zeitlich atomar (*atomically*) ausgeführt, d.h. die Werte aller betroffenen Refs werden zeitgleich gesetzt. Faktisch müssen dafür für einen Moment die Lesezugriffe auf Ref-Referenzen unterbunden werden.

Transaktionen können verschachtelt werden. Der Restart einer inneren Transaktion löst den Restart der sie einbettenden Transaktion aus. Der Gesamtcommit wird ausschließlich von der äußersten Transaktion ausgelöst.

Es gibt zwei weitere Möglichkeiten, um die Werte von Ref-Referenzen zu ändern, `ref-set` und `commute`. Mit `ref-set` wird ein Wert für eine Ref innerhalb einer Transaktion einfach gesetzt.

```
user=> (dosync (ref-set account 1000))
1000
```

Ein `commute` arbeitet fast wie ein `alter`, es ändert den transaktionseigenen referenzierten Wert innerhalb einer Transaktion. Zum Zeitpunkt des

Commits gibt es jedoch einen entscheidenden Unterschied: Bei `alter` wird überprüft, ob der referenzierte Wert von außen seit dem Schnappschuss eine Änderung erfahren hat; falls ja, wird die Transaktion wiederholt. Bei `commute` wird der aktuelle „Außenwert“ der Referenz genommen und die `commute`-Funktion angewendet; dieser Wert wird dann mit dem Commit übergeben. Die `commute`-Funktion wird also immer zwei Mal angewendet. Es findet kein Abgleich mit dem Schnappschusswert statt, und ein `commute` löst keinen Restart aus.

Ändert sich der referenzierte Wert außerhalb der Transaktion nicht, verhält sich `commute` exakt wir `alter`:

```
user=> (dosync (commute account - 300))
700
```

Konstruieren wir ein Szenario, bei dem wir die Ref-Referenz während der Transaktion ändern, so ist der Unterschied beobachtbar:

```
user=> (def thread (agent "Thread"))
#'user/thread
user=>  (def account (ref 1000))
#'user/account
user=> (send thread
  (fn [agt aref]
    (dosync (commute aref + 100) (Thread/sleep 8000) agt))
  account)
#<Agent@5a0029ac: "Thread">
user=> (dosync (ref-set account 2000))
2000
user=> ; Wait some seconds
user=> @account
2100
```

▷ **Aufgabe 11.2** Für Fortgeschrittene: Wenn Sie in dem Beispiel `commute` durch `alter` austauschen, werden Sie feststellen, dass die zweite Transaktion die REPL solange blockiert bis die erste Transaktion fertig ist. Warum?

Sollen referenzierte, aber in der Transaktion nicht veränderte Werte ebenfalls bei Außenänderung einen Restart anstoßen, so ist `ensure` zusammen mit der Referenz zu verwenden. So wird auch der reine Lesezugriff auf Referenzen geschützt vor Änderungen, die die Wert-Annahmen der Transaktion zu Beginn ihrer Arbeit (beim Anlegen des Schnappschusses) kompromittieren.

Ein wichtiger Punkt im Umgang mit Transaktionen ist: Da Transaktionen durch Restarts wiederholt werden können, sollten Transaktionen keine Seiteneffekte haben. Insbesondere sollten Ein-/Ausgabeoperationen unterbleiben. Um niemals versehentlich eine Ein-/Ausgabe innerhalb einer Transaktion zu verwenden, empfiehlt sich die Verwendung des Makros `io!` für jede Ein- bzw. Ausgabe.

```
user=> (io! (println "Hey"))
Hey
nil
user=> (dosync (io! (println "Hey")))
java.lang.IllegalStateException: I/O in transaction (NO_SOURCE_FILE:0)
```

Generell führen Ausnahmebedingungen (*exceptions*) zu einem Abbruch der Transaktion. Es findet kein Commit statt.

11.5 Der Var-Referenztyp

Den vierten Referenztyp, den Var-Typ, kennen Sie bereits – wenngleich wir ihn namentlich noch nicht eingeführt haben. Wann immer Sie ein `def` verwenden, wird eine Var-Referenz erzeugt. Schauen wir uns das im Detail an!

Wenn Sie in der REPL `xyz` eingeben, versucht Clojure im aktuellen Namensraum – im Ausgangszustand ist das `user` – das Symbol `xyz` aufzulösen; das ist die Evaluationsvorschrift für Symbole. Da ein Namensraum nicht viel mehr als eine Map ist, sucht Clojure nach dem mit dem Symbol assoziierten „Objekt“. Existiert die Assoziation nicht, wirft Clojure eine Ausnahmebedingung.

```
user=> xyz
java.lang.Exception: Unable to resolve symbol: xyz in this context (NO_SOU
LE:0)
```

Mit `def` und einem Symbol als einzigem Argument legt Clojure eine Assoziation in der Map des aktuellen Namensraums an (hier `user`) und zwar *immer* zu einer Var-Referenz! Der Rückgabewert von `def` ist die Var-Referenz.

```
user=> (def xyz)
#'user/xyz
```

Var-Referenzen präsentieren sich in einer Schreibweise, mit der man sie auch direkt abrufen kann. `#'` ist ein Reader-Makro, das der Spezialform `var` entspricht. Die Spezialform erwartet ein Symbol. Unqualifizierte Symbolnamen beziehen sich auf den aktuellen Namensraum.

```
user=> (var xyz)
#'user/xyz
user=> #'xyz
#'user/xyz
```

Wenn wir jetzt das Symbol `xyz` von Clojure evaluieren/auflösen lassen, unternimmt Clojure einen weiteren Schritt: Clojure sucht aus der Map des Namensraums nicht nur die mit dem Symbol assoziierte Var-Ref heraus, sondern liefert den dereferenzierten Wert der Var-Referenz zurück. Da wir in unserem `def`-Ausdruck ein zweites Argument haben vermissen lassen, ist die Var-Referenz ohne Bezug, d.h. ungebunden (*unbound*).

```
user=> xyz
java.lang.IllegalStateException: Var user/xyz is unbound. (NO_SOURCE_FILE:
```

Mit `bound?` kann man feststellen (reflektieren), ob eine Var-Referenz gebunden ist oder nicht.

```
user=> (bound? (var xyz))
false
```

Setzen wir das `def` wie üblich mit zwei Argumenten auf, ist die Dereferenzierung der Var-Referenz erfolgreich.

```
user=> (def xyz 7)
#'user/xyz
```

```
user=> xyz
7
user=> (bound? (var xyz))
true
```

Da die Assoziation vom Symbol `xyz` zur Var-Referenz `'#user/xyz` in der Map des aktuellen Namensraums bereits existiert, wird mit diesem `def` lediglich die vorhandene Var-Referenz auf den neuen Wert gesetzt. Existiert die Assoziation noch nicht, falls es kein entsprechendes, vorausgehendes `def` gegeben hat, legt Clojure die Assoziation erstmalig an. In Clojure heißt dieser Vorgang *Interning*. Die Bindung der Var-Referenz an einen Wert nennt man Wurzelbindung (*root binding*).

▷ **Aufgabe 11.3** Beschreiben Sie, was `(def atm (atom 3))` an Assoziationen bzw. Bindungen aufsetzt.

▷ **Aufgabe 11.4** Es gibt in Clojure ein Makro, das wie `def` benutzt wird, die Var-Referenz jedoch nur bindet, wenn keine Wurzelbindung existiert. Wie lautet dieses Makro?

▷ **Aufgabe 11.5** Wie arbeitet das `declare`-Makro?

Anbei: Ein `deref` bzw. `@` dereferenziert auf bekannte Weise eine Var-Referenz. Alternativ kann `var-get` benutzt werden. `var?` testet, ob eine Var-Referenz vorliegt.

```
user=> @(var xyz)
7
user=> (var-get (var xyz))
7
user=> (var? #'xyz)
true
```

Mit `alter-var-root` kann die Wurzelbindung einer Var-Referenz auf genau dieselbe Weise geändert werden wie mit `send` bei Agenten, mit `swap!` bei Atomen und `alter` bei Ref-Referenzen. Als Argumente folgen die Var-Referenz, eine Funktion und gegebenenfalls weitere Argumente, die an die Funktion übergeben werden.

```
user=> (alter-var-root (var xyz) inc)
8
user=> xyz
8
```

Die per `def` aufgesetzten Wurzelbindungen sind global und für alle Threads sichtbar aber für vom Hauptthread aufgesetzte „Kind“-Threads nicht veränderbar. Jeder Thread kann jedoch im Rumpf einer `binding`-Form den Bindungswert der Var-Referenz lokal für den Thread verändern. Nach Verlassen der `binding`-Form gilt wieder der vormalige Var-Wert. Da `binding`-Formen verschachtelbar sind, überlagern sich die Var-Bindungen ähnlich einer Stapel-Datenstruktur (*stack*).

In Clojure 1.3 muss im `def`-Ausdruck die Var-Referenz explizit als dynamisch, d.h. per `binding` veränderbar ausgewiesen werden. Das geht über die Metainformation `^:dynamic`. Ohne diese Information gilt die Var-Referenz als statisch (unveränderlich). In Clojure 1.2 sind alle Var-Referenzen grundsätzlich dynamisch. Da die wenigsten Var-Referenzen

dynamisch verwendet werden, profitiert in Clojure 1.3 der Compiler von der Grundannahme statischer Var-Referenzen.

Das `binding`-Makro erwartet wie die `let`-Form Paare von Symbolen und auszuwertenden Ausdrücken innerhalb der Klammern `[` und `]`. Die Ausdrücke im Rumpf werden nacheinander ausgewertet und der letzte Auswertungswert zurückgegeben.

```
user=> (def ^:dynamic x 7)
#'user/x
user=> (list x (binding [x 3] x) x)
(7 3 7)
user=> (list x (binding [x 3] (binding [x 1] x)) x)
(7 1 7)
```

▷ **Aufgabe 11.6** Begründen Sie Ihre Antwort: Was liefert der Ausdruck `(list x (binding [x 3] (binding [x 1] x) x) x)`?

▷ **Aufgabe 11.7** Zeigen Sie mit `thread-bound?`, dass `binding` eine Bindung aufsetzt, die lokal für den Thread gültig ist.

Innerhalb einer `binding`-Form kann der aktuelle Var-Wert per `set!` oder alternativ per `var-set` geändert werden. `set!` erwartet als erstes Argument ein Symbol, `var-set` eine Var-Referenz; als zweites Argument kann ein beliebiger Ausdruck übergeben werden. Ein Beispiel:

```
user=> (list x (binding [x 3] (set! x (+ 2 3)) x) x)
(7 5 7)
user=> (list x (binding [x 3] (var-set #'x (+ 2 3)) x) x)
(7 5 7)
```

Mit dem Aufruf der Funktion `get-thread-bindings` ohne Argumente erhalten Sie eine Map mit allen Var/Wert-Paaren, die lokal in dem Thread gebunden sind.[3]

```
user=> (get-thread-bindings)
{#'clojure.core/*unchecked-math* false, #'clojure.core/*1 nil, ... }
```

Sie finden in der Map einige Bindungen, deren Namen mit einem oder zwei Sternen „`*`“ hervorgehoben sind. Diese Bindungen haben eine spezielle Funktion, wie z.B. `*unchecked-math*`. Mit dem `doc`-Marko erfahren Sie mehr über diese Var-Referenzen.

Das `binding`-Makro und die Spezialform `let` sind nicht nur in der Anwendung ähnlich, sie scheinen auch fast dasselbe zu bewirken. Die Unterschiede liegen in den Details. Während der Gebrauch von `let` unkritisch ist, sind dynamische Bindungen via `binding` mit Vorsicht zu genießen; sie können die Modularität d.h. die statische Betrachtung von Programmeinheiten empfindlich stören.

Ein Unterschied besteht darin, wie die Bindungen aufgesetzt werden. In einem `let` werden die Bindungen der Reihe nach aufgesetzt, wobei jede neue Bindung den Bindungswert der vorausgehenden Bindung „sehen“ kann. Bei einem `binding` werden alle Bindungen unabhängig voneinander aufgesetzt.

[3] Die entsprechende „Gegenfunktion“ `push-thread-binding` ist eine Low-Level-Funktion, die man nicht verwenden sollte.

Wesentlicher ist jedoch der Unterschied in der Auflösung einer Bindung, sprich der Evaluation eines Symbols. Grundsätzlich berücksichtigt die Auflösung einer Bindung (Evaluation einer dynamischen Var-Referenz) zuerst den lexikalischen und dann den dynamischen Kontext. Mit `let` aufgesetzte Bindungen finden ihren Rahmen und ihre Begrenzung in den lexikalischen Verschachtelungsstrukturen; die Auflösungsbezüge sind strikt lokal und statisch am Code ableitbar. Eine `binding`-Bindung setzt einen zeitlichen Bindungsrahmen. Der Wert einer dynamischen Var-Referenz hängt ab vom Zeitpunkt der Auswertung.

Ein einfaches Beispiel hilft, die Zusammenhänge zu erläutern.

```
user=> (def ^:dynamic x 7)
#'user/x
user=> ((fn [y] (+ x y)) 3)
10
```

Mit `def` wird das Symbol `x` mit einer dynamischen Var-Referenenz im Namensraum `user` assoziiert. Falls Sie Clojure 1.2 benutzen, lassen Sie `^:dynamic` weg. Die Var-Referenz hat den Wert `7` als Wurzelbindung.

In der Funktionsdefinition stellt `y` aus der Argumentenliste den lexikalischen Kontext für den Funktionsrumpf her. Somit ist klar, dass sich das `y` im Funktionsrumpf auf das Argument `y` bezieht. Bei Anwendung der Funktion ist darüber festgelegt, dass `y` im Funktionsrumpf mit dem Wert des `y`-Arguments übereinstimmt.

Das `x` im Funktionsrumpf ist eine sogenannte „freie Variable“, deren Kontext nicht innerhalb der Funktionsumgebung bestimmt ist. Zur Auflösung von `x` muss der Blick über den „Tellerrand“ der Funktion gewagt werden. Da die Funktion nicht in weiteren lexikalischen Strukturen eingebettet ist, muss `x` zur Laufzeit, d.h. im Moment der Auswertung der `+`-Funktion aufgelöst werden.

Mit `let` bzw. `binding` lassen wir nun die Funktionsanwendungen im lexikalischen bzw. dynamischen Kontext ablaufen: `let` stellt einen lokalen Kontext her, der `x` lexikalisch an `1` bindet, `binding` setzt einen zeitlichen Kontext auf, innerhalb dessen `x` an `1` gebunden ist. Die Auswertung der Funktion fällt in den zeitlichen Rahmen der `binding`-Form.

```
user=> (let [x 1] ((fn [y] (+ x y)) 3))
4
user=> (binding [x 1] ((fn [y] (+ x y)) 3))
4
```

Lösen wir die Funktionsanwendung aus dem Kontext des `let` bzw. `binding` heraus, werden die Konsequenzen deutlich. Achten Sie genau auf die Klammern.

```
user=> ((let [x 1] (fn [y] (+ x y))) 3)
4
user=> ((binding [x 1] (fn [y] (+ x y))) 3)
10
```

Die lexikalischen Strukturen binden im ersten Fall `x` an `1`. Im zweiten Fall findet die Anwendung der Funktion dann statt, wenn die zeitliche Bindung von `x` an `1` nicht mehr gültig ist, sondern die Wurzelbindung an den Wert `7` relevant ist.

Wie man an dem Beispiel schön sehen kann, hält die „verzögerte“ Evaluation der Funktion offen, welcher Wert für `x` anzusetzen ist. Das Ergebnis der Funktionsanwendung ist eben nicht mehr nur von dem Aufrufwert für `y` abhängig, sondern darüber hinaus von dem Wert der freien Variablen `x`. Das verletzt das Grundprinzip der funktionalen Programmierung („Gleicher Input, gleicher Output“). Mit der freien Variablen gibt es einen impliziten Input, der nicht aus der Argumentenliste einer Funktion erkenntlich ist.

Aus diesem Grund sind dynamische Var-Referenzen mit Vorsicht einzusetzen und zu verwenden. Wir Menschen sind mit der Analyse der Bindungsverhältnisse von freien Variablen außerhalb lexikalischer Kontexte intellektuell schnell überfordert. Die verzögerte Auswertung (wie etwa auch im Fall von verzögerten Listen) löst Ausdrücke sozusagen aus ihren lexikalischen Zusammenhängen und erfordert die Analyse von Bindungsbezügen außerhalb der lexikalischen Strukturen – man verliert dabei leicht den Überblick.

Es gibt neben `binding` zwei verwandte Formen, um lokal für einen Thread gültige Var-Referenzen in Stapelmanier einer Wurzelbindung zu überlagern. `with-bindings` erwartet eine Map an Var/Wert-Paaren, setzt die Bindungen auf und führt dann den Rumpf aus. `with-bindings*` erwartet ebenso eine Map, als zweites Argument jedoch eine Funktion und weitere, optionale Argumente.

```
user=> (with-bindings {#'x 1} ((fn [y] (+ x y)) 3))
4
user=> (with-bindings* {#'x 1} (fn [y] (+ x y)) 3)
4
```

Der Vollständigkeit halber sei erwähnt, dass das Makro `with-local-vars` Var-Referenzen and lokale Symbole binden kann, ohne dass die Var-Referenzen in der Map des aktuellen Namensraums assoziiert werden. Dieses Feature ist für die tägliche Programmierarbeit praktisch irrelevant.

11.6 Validierung von Referenz-Änderungen

Alle Referenztypen (Var, Ref, Atom, Agent) können vor einer Änderung der Referenz auf einen neuen Wert daraufhin überprüft (validiert) werden, ob der neu zu setzende Wert eine bestimmte Bedingung erfüllt. Erfüllt der neu zu setzende Wert die Bedingung, wird die Referenz auf den neuen Wert gesetzt. Erfüllt er sie nicht, bleibt die Referenz auf dem aktuellen Wert bestehen und eine Ausnahmebedingung (*exception*) wird geworfen.

Zur Überprüfung (Validierung) ist eine Funktion anzugeben, die nur ein Argument entgegen nimmt und keine Seiteneffekte hat. Bei einer Rückgabe von `false` oder `nil` gilt die Validierung als fehlgeschlagen. Tritt bei der Validierung eine Ausnahmebedingung auf, ist die Valisierung ebenfalls nicht erfolgreich. Die Validierungsfunktion wird zusammen mit der Referenz als erstem Argument über die Funktion `set-validator!` aktiviert.

Nehmen wir ein Beispiel mit einer Atom-Referenz. Der referenzierte Wert darf sich nur im Wertebereich von `1` bis `23` inklusive bewegen.

```
user=> (def hour (atom 7))
#'user/hour
```

```
user=> (set-validator! hour (fn [h] ((set (range 1 24)) h)))
nil
user=> (reset! hour 23)
23
user=> (reset! hour 24)
IllegalStateException Invalid reference state  clojure.lang.ARef.validate
java:33)
```

▷ **Aufgabe 11.8** Setzen Sie das gleiche Szenario mit einer Var-Referenz auf!

Die Validierungsfunktion kann, einmal gesetzt, mit `get-validator` abgefragt werden.

```
user=> (get-validator hour)
#<user$eval116$fn__117 user$eval116$fn__117@1f0dc656>
```

Die Validierungsfunktion kann mit `nil` wieder deaktiviert werden.

```
user=> (set-validator! hour nil)
nil
user=> (reset! hour 24)
24
```

Derzeit lassen sich mit `add-watch` und `remove-watch` sogenannte „Wachhunde“ (*watchdogs*) auf Referenzen ansetzen bzw. wieder entfernen. Bei Änderungen einer Referenz wird eine Überwachungsfunktion aufgerufen. Die Funktionalität ist derzeit im Alpha-Status und mag Änderungen erfahren. Aktuelle Auskünfte bietet die Dokumentation zu Clojure.

11.7 Anmerkungen

Einfacher kann man Nebenläufigkeit nicht haben: einen Agenten anfordern und ihn per `send` bzw. `send-off` mit Aktionen beschäftigen. Und schon erfüllen die Agenten „im Hintergrund“ ihre Mission. Diese Leichtigkeit im Umgang mit Nebenläufigkeit hebt sich wohltuend ab vom Thread-Management, mit dem man sich in vielen anderen Programmiersprachen beschäftigen muss.

Manch einem Entwickler mag zwar die Kontrolle über die Details des Thread-Managements fehlen. Doch in der Regel sollte man kaum mehr benötigen. Der Einsatz von Nebenläufigkeit sollte so einfach wie irgend möglich sein – sonst wird sie nicht genutzt. Clojure schlägt hier eine bedeutende Richtung ein, um der Nebenläufigen Programmierung ihren Schrecken soweit wie möglich zu nehmen.

Ein Referenztyp hat mehr Raum in diesem Kapitel eingenommen als die anderen Referenztypen: Die Var-Referenz ist wichtig unabhängig davon, ob Sie mit Nebenläufigkeit arbeiten wollen oder nicht. Die Mechanismen hinter einem `def` sind ein wichtiger Baustein, um die Auflösung von Symbolen zu verstehen.

11.8 Lösungen

Aufgabe 11.1 Die Funktion `withdraw` kann nun nicht mehr mit einer Vorbedingung versehen werden, die nur positive `amount`-Beträge zulässt. Negative Abbuchungsbeträge topedieren die Semantik der erwarteten Funktionsweise von `withdraw` und verhindern eine Überprüfung der semantischen Erwartungshaltung per Vorbedingung.

Aufgabe 11.2 Die Antwort findet sich über die Clojure-Dokumentation zu „Refs and Transactions“.[4] Clojure implementiert STM via *Multiversion Concurrency Control* (MVCC). Die Dokumentation verweist auf den englischen Wikipedia-Artikel zu MVCC, wo es heißt:[5]

> „If a transaction (T_i) wants to write to an object, and if there is another transaction (T_k), the timestamp of T_i must precede the timestamp of T_k (i.e., $TS(T_i) < TS(T_k)$) for the object write operation to succeed. Which is to say a write cannot complete if there are outstanding transactions with an earlier timestamp.“

Die erste Transaktion aus unserem Beispiel blockiert den Schreibzugriff der nachfolgenden Transaktion, die erst nach mehreren Wiederholungsversuchen mit Beendigung der ersten Transaktion zum Zug kommt. Mehr Details zu Clojures STM-Implementierung sind nachzulesen in [NK10].

Aufgabe 11.3 Der Ausdruck `(def atm (atom 3))` setzt, sofern nicht bereits vorhanden, in der Map des aktuellen Namensraus eine Assoziation vom Symbol `atm` zu einer Var-Referenz auf. Die Var-Referenz ist gebunden an das Evaluationsergebnis von `(atom 3)`, sprich an ein Atom, das den Wert `3` referenziert.

Aufgabe 11.4 Das Makro `defonce` erwartet zwei Argumente und ist einem `def` gleichzusetzen unter der Voraussetzung, dass keine Wurzelbindung existiert.

Aufgabe 11.5 `doc declare` verrät es fast: `declare` entspricht pro Symbolnamen je einem `def` ohne Wertargument. Es wird also keine Wurzelbindung aufgesetzt.

Aufgabe 11.6 Die Antwort lautet `(7 3 7)`. Das erste Binding hat zwei Ausdrücke im Rumpf. Der letzte Ausdruck bestimmt den Rückgabewert. Die Rückgabe des ersten Ausdrucks, das zweite Binding liefert für `x` den Wert `1`, wird verworfen und spielt hier keine Rolle.

Aufgabe 11.7 Die folgende Interaktion zeigt, wie eine Bindung innerhalb von `binding` *thread-local* ist.

```
user=> (def ^:dynamic x 7)
#'user/x
user=> (bound? (var x))
true
user=> (thread-bound? (var x))
false
user=> (list x (binding [x 3] (thread-bound? (var x))))
(7 true)
```

Aufgabe 11.8 Achten Sie darauf, der Funktion `set-validator!` die Var-Referenz und nicht den Wert der Var-Referenz zu übergeben. Sie erhalten ansonsten

[4] http://clojure.org/refs
[5] http://en.wikipedia.org/wiki/Multiversion_concurrency_control (5. Sep. 2011)

eine Ausnahmebedingung aufgrund eines missglückten Versuchs der Typanpassung (*casting*).

```
user=> (def hour 7)
#'user/hour
user=> (set-validator! (var hour) (fn [h] ((set (range 1 24)) h)))
nil
user=> (def hour 23)
#'user/hour
user=> (def hour 24)
IllegalStateException Invalid reference state  clojure.lang.ARef.validate
java:33)
```

Wie Sie sehen, muss die Var-Referenz nicht dynamisch sein, um validiert zu werden.

Historie

22. Mai 2012	Lösungskapitel erscheint im Inhaltsverzeichnis
24. Sep. 2011	1. Fassung
8. Aug. 2011	Schreibbeginn

Literaturverzeichnis

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[GJ08] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer, 2nd edition, 2008.

[NK10] Peder R. L. Nielsen and Patrick T. Kristiansen. Benchmarking contention management strategies in clojure's software transactional memory implementation. Master's thesis, Aalborg University, Computer Science Department, Database and Programming Technologies, 2010.