

# Konkatenative Programmierung mit Consize

Dominikus Herzberg\*  
Hochschule Heilbronn

18. Dezember 2012

\* Copyright © 2012, Dominikus Herzberg

Dieses Werk bzw. dieser Inhalt steht unter der Creative-Commons-Lizenz  
CC BY-NC-SA 3.0: [Namensnennung](#) – [keine kommerzielle Nutzung](#) – [Weitergabe unter gleichen Bedingungen](#).

---

Viele Leser und Leserinnen haben mir geholfen, kleinere und größere Mängel im Text zu beseitigen. Mein Dank geht an:<sup>1</sup> Marco Brinkmeier (1), Hasan Cakir (1), Dominik Föhl (1), Ghislain Lova (3), Felix Radzowski (2).

---

<sup>1</sup> Die Zahl hinter den Namen gibt die Anzahl gefundener „Fehler“ an.

---

# Inhaltsverzeichnis

---

<b>1</b>	<b>Warum Consize?</b>	<b>5</b>
1.1	Die Geschichte zu Consize . . . . .	5
1.2	Consize: Installation und Inbetriebnahme . . . . .	7
1.3	Ein Hinweis für Einsteiger . . . . .	8
<b>2</b>	<b>Erste Schritte mit Consize: Die Basics</b>	<b>9</b>
2.1	Programme sind Texte . . . . .	9
2.2	Wie Consize denkt: Die Ur-Grammatik von Consize . . . .	9
2.3	Das Parsen eines Programms . . . . .	11
2.4	Was Consize versteht: Die erweiterte Grammatik . . . . .	13
2.5	Datenstrukturen . . . . .	14
2.6	Datenstapel und Programmstapel . . . . .	16
2.7	Atomare und nicht-atomare Wörter . . . . .	17
2.8	Mappings . . . . .	19
2.9	Was sind Datenstrukturen? . . . . .	20
<b>3</b>	<b>Die VM von Consize: Der Kernel</b>	<b>23</b>
3.1	Kurzdokumentation mittels Stapeleffekten . . . . .	23
3.2	Stack Shuffler: <code>dup</code> , <code>swap</code> , <code>drop</code> , <code>rot</code> . . . . .	24
3.3	Typ und Vergleich: <code>type</code> , <code>equal?</code> , <code>identical?</code> . . . . .	24
3.4	Stapel: <code>emptystack</code> , <code>push</code> , <code>top</code> , <code>pop</code> , <code>reverse</code> , <code>concat</code> . .	25
3.5	Mappings: <code>mapping</code> , <code>unmap</code> , <code>assoc</code> , <code>dissoc</code> , <code>get</code> , <code>keys</code> , <code>merge</code>	26
3.6	Wörter: <code>unword</code> , <code>word</code> , <code>char</code> . . . . .	27
3.7	Konsole: <code>print</code> , <code>flush</code> , <code>read-line</code> . . . . .	28
3.8	Dateien und mehr: <code>slurp</code> , <code>spit</code> , <code>spit-on</code> . . . . .	29
3.9	Parsing: <code>uncomment</code> , <code>tokenize</code> , <code>undocument</code> . . . . .	30
3.10	Funktionen: <code>func</code> , <code>apply</code> , <code>compose</code> . . . . .	31
3.11	Continuations: <code>stepcc</code> , <code>call/cc</code> , <code>continue</code> , <code>get-dict</code> , <code>set--dict</code> , <code>\</code> . . . . .	31
3.12	Arithmetik: <code>+</code> , <code>-</code> , <code>*</code> , <code>div</code> , <code>mod</code> , <code>&lt;</code> , <code>&gt;</code> , <code>integer?</code> . . . . .	35
3.13	Extras: <code>current-time-millis</code> , <code>operating-system</code> . . . . .	36
3.14	Der Start der VM . . . . .	37
3.15	Referenzimplementierung . . . . .	37

<b>4</b>	<b>Von der Gleichheit und der Identität</b>	<b>39</b>
4.1	Gleich ist, was aus gleichen Werten bei gleicher Struktur besteht . . . . .	39
4.2	Identität ist Verfolgbarkeit . . . . .	40
4.3	Identitäten durch Referenzen . . . . .	41
4.4	Referenzielle Transparenz durch Immutabilität . . . . .	42
<b>5</b>	<b>Die Prelude</b>	<b>44</b>
5.1	Vorbereitungen: Was sein muss und was nützlich ist . . .	44
5.1.1	Consize-Lizenz . . . . .	44
5.1.2	Wie man Programme schreibt . . . . .	44
5.1.3	Grundverhalten von <code>read-word</code> und <code>read-mapping</code>	45
5.1.4	Mehr davon: Stack Shuffler . . . . .	46
5.1.5	Freunde und Helferlein . . . . .	47
5.2	Grundlegende Kombinatoren . . . . .	48
5.2.1	Quotierungen aufrufen . . . . .	48
5.2.2	Boolesche Werte und die binäre Wahl . . . . .	49
5.2.3	Entscheiden, wo's lang geht . . . . .	50
5.3	Kombiniere: Kombinatoren . . . . .	53
5.3.1	„Abtauch“-Kombinatoren: <code>dip</code> . . . . .	53
5.3.2	Erhaltungskombinatoren: <code>keep</code> . . . . .	54
5.3.3	Cleave-Kombinatoren: <code>bi</code> , <code>tri</code> und <code>cleave</code> . . . . .	55
5.3.4	Spread-Kombinatoren: <code>bi*</code> und <code>tri*</code> . . . . .	55
5.3.5	Apply-Kombinatoren: <code>bi@</code> und <code>tri@</code> . . . . .	56
5.3.6	Sequenzkombinatoren: <code>each</code> und <code>Co.</code> . . . . .	56
5.4	Erweiterung der Consize-Grammatik . . . . .	58
5.4.1	Literale für zusammengesetzte Datentypen . . . . .	58
5.4.2	Neue Wörter mit <code>:</code> bis <code>;</code> definieren . . . . .	60
5.4.3	Symbole . . . . .	62
5.5	Die Interaktion mit Consize . . . . .	63
5.5.1	Repräsentation von Daten und ihre Ausgabe . . .	63
5.5.2	Die REPL: Die Interaktion über die Konsole . . .	64
5.5.3	Dateien lesen und starten . . . . .	65
5.5.4	Reflexion und Debugging . . . . .	65
5.5.5	Unit-Testing . . . . .	67
5.6	Serialisierung, Consize-Dumps und Bootstrapping . . . . .	67
5.6.1	Die Serialisierung von Daten . . . . .	67
5.6.2	Der Dump: Ein Schnappschuss des Wörterbuchs .	68

5.6.3	Bootstrapping Consize . . . . .	69
5.7	Here we go! . . . . .	72
5.7.1	Consize meldet sich zur Arbeit . . . . .	72
5.7.2	Von der Dokumentation zum Code . . . . .	72
<b>A</b>	<b>Mathematische Grundlagen</b>	<b>73</b>
A.1	Der konkatenative Formalismus in denotationeller Semantik	73
A.2	Der Bezug zur operationalen Semantik, der Consize-VM .	74

---

# 1 Warum Consize?

---

## 1.1 Die Geschichte zu Consize

An manchen weiterführenden Schulen lernt man von Beginn an ein Instrument zu spielen. Ein beliebtes Instrument für Anfänger ist die Flöte. Warum Flöte? Das Instrument ist bei Kindern nicht unbedingt populär – viele kennen Flöten bestenfalls aus der Weihnachtszeit. Der „Sound“ einer Flöte strahlt nicht die unmittelbare Faszination eines rauen, schrillen Akkords auf einer E-Gitarre aus. Warum also Flöte?

Die Feinheit und Klarheit des Flötentons trainiert das Gehör. Und einen weiteren, immensen Vorteil hat die Flöte: Die bei jedem Instrument notwendige koordinative Leistung über die Hände ist reduziert auf das Wesentliche. Wer Nintendo, Playstation oder Xbox spielen kann, der kann auch Löcher zuhalten – ob er oder sie musikalisch ist oder nicht, das spielt erst einmal keine Rolle. Damit mich niemand missversteht: Eine Flöte ist kein Instrument einzig für musikdidaktische Spielchen mit Kindern. Könner trainieren über Jahre ihre Atemtechnik und Spielfertigkeit. Es bedarf der Meisterschaft, um dem Flötenspiel Leichtigkeit und Lebendigkeit abzugewinnen.

Ich habe lange nach einer „Flöte“ für meine Informatik-Studierenden gesucht. Ich suchte nach einer Programmiersprache, die – einer Flöte gleich – von bestechender Einfachheit und deshalb leicht zu erlernen ist, aber nichts an Ausdrucksfähigkeit, an „musikalischem Reichtum“ vermissen lässt. Ich wünschte mir ein „Programmierinstrument“, das über die Turing-Maschine hinausgeht. Die Turing-Maschine ist ein ganz primitiver Programmierapparat, der so etwas wie die Maultrommel der Informatik ist. Nett für die Effekthascherei, theoretisch von immenser Bedeutung, da im Prinzip ein vollwertiges Musikinstrument, aber aufgrund des begrenzten Tonumfangs nur wenig reizvoll im Musikalltag. Andererseits sollte die „Programmierflöte“ einfacher sein als der Lambda-Kalkül. Der Lambda-Kalkül bildet die Grundlage fast aller funktionalen Sprachen. Lambda-Ausdrücke sind wie Orgelpfeifen, die richtig zusammengestellt und aufeinander abgestimmt eine imposante Orgel erschaffen, mit der sich gewaltige Werke orchestrieren lassen. Nicht selten glänzen funktionalen Programmierern bei Haskell, ML, Scheme oder Lisp ebenso die Augen, wie einem Musikbegeisterten die Orgelwerke von Bach in Entzückung versetzen. Zu überwältigend für Anfänger!

Wovon ich die Finger lassen wollte, das sind die imperativen, objekt-orientierten Sprachen, die überall verbreitet sind und an sehr vielen Hochschulen aufgrund ihrer Praxisrelevanz gelehrt werden. Java und C# sind die elektronischen Keyboards mit allem nur erdenklichen Schnack. Im Zweifel muss man nur noch mit einem Finger die Tasten drücken, die gesamte Begleitband ertönt wie von selbst aus den Lautsprechern. Man kann sehr schnell beeindruckende Musikstücke hervorzubauern, ohne selbst allzu viel von Musik zu verstehen. Hier gewinnt schnell derjenige, der sich in der Bedienung der zahllosen Knöpfe auskennt. Aber wehe, man muss sich mit diesen oberflächlichen Kenntnissen in ein Orchester einfügen. Das kann gründlich daneben gehen!

Die sogenannten Skript-Sprachen, wie beispielsweise Python oder Ruby, haben einen so ursprünglichen Reiz wie Akustikgitarren – Ruby on Rails sicher den einer E-Gitarre. Wenige Akkorde und die eigene Stimme reichen, um mitreißende Lieder zu schmettern und rührende Balladen zu trällern. Wer schaut nicht neidvoll auf die Barden, die mit den immer gleichen Akkorden so mühelos einen ganzen Abend retten und in Lagerfeuerstimmung zu tauchen vermögen? Mancher Musikkennner mag für dieses profane Können nur Verachtung übrig haben. Aber die Ruby- und Python-Programmierer sind die großen Pragmatiker, die das, was sie können, auch wirklich beherrschen und damit eine beachtliche Vielfalt an Problemen gut und zielsicher bewältigen. Das verstört die „Keyboard-Programmierer“ bisweilen sehr. Meine Erfahrung zeigte mir aber, dass das Greifen von Akkorden auf der Gitarre nicht so leicht zu erlernen ist.

Als ich auf die Arbeiten von MANFRED VON THUN und die Programmiersprache Joy stieß, ahnte ich, bei den Blasinstrumenten angekommen zu sein. Die konzeptuelle Eleganz von Joy ist frappierend. Joy vereinfacht die Ideen, die JOHN BACKUS mit FP schon Ende der 1970er Jahre aufgeworfen hatte. Joy wie FP sind in meinen Augen Geniestreiche. Als ich dann noch Forth von CHARLES H. MOORE und Factor von SLAVA PESTOV entdeckte, war es um mich geschehen. Das konkatenative Programmierparadigma hatte mich in den Bann gezogen. Und meine ersten Experimente, Factor in der Lehre einzusetzen, verliefen überraschend gut.<sup>1</sup>

Allerdings wollte ich es noch etwas einfacher haben, so einfach wie möglich, ohne als Programmiersprache unpraktikabel zu werden. Und so beschloss ich, mir eine kleine Blockflöte an der Drehbank zu dreheln. Das Ergebnis ist Consize. Die Consize-VM (Virtuelle Maschine) besteht aus nicht einmal 150 Programmzeilen Clojure-Code; der Antriebsmotor von Factor umfasst rund 10.000 Zeilen C++-Code. Bei Consize reichen um die 500-600 Zeilen Consize-Code, um Consize selbst zu einer interaktiven Sprache zu erweitern. Bei Factor ist die Code-Basis um einiges größer und der Bootstrapping-Prozess deutlich komplizierter. Dafür ist Factor hochperformant und Consize eine lahme Ente, was für meine Einsatzzwecke jedoch unproblematisch ist.

Consize will zweierlei erreichen: Zum einen möchte ich Ihnen mit Consize aufzeigen und erläutern, wie einfach viele wichtige Programmierkonzepte der Informatik sind. Es macht einfach Spaß, sich einen Debugger mit Einzelschrittmodus in wenigen Zeilen Code zu bauen. Ebenso lässt sich ein Testwerkzeug für Unit-Tests in 10 Zeilen Consize-Code realisieren. Polymorphie und eine einfache Objekt-Orientierung mit Mehrfachvererbung sind in 20-30 Zeilen Consize-Code zu haben. Meta-Programmierung und der Umgang mit Continuations sind eine Fingerübung. Consize stellt sich gegen den Trend immer „größerer“ Programmiersysteme mit mächtigen Bibliotheken und Frameworks, die in ihrem Umfang und in ihrer Vielfalt kaum mehr zu überblicken und zu beherrschen sind. Consize möchte destillieren und vereinfachen, so dass viele grundlegende Programmierkonzepte im Rahmen eines einzigen Semesters erfassbar und verstehbar sind.

Für mich bleibt konkatenative Programmierung – und das ist der andere Aspekt – nicht bei den Programmierkonzepten stehen. Ich habe mich jahrelang mit der Architektur und der Modellierung von Softwaresystemen aus der Telekommunikation beschäftigt. Telekommunikationsnetze

<sup>1</sup> Siehe „Factor @ Heilbronn University“ im [denkspuren-Blog](#).

wie auch das Internet sind Meisterwerke der (Software-)Ingenieurskunst. Bei aller Kompliziertheit dieser Netze sind die dahinter liegenden Design-Prinzipien einfach, die sich bis auf die Ebene der Software niederschlagen. Sie sind der Garant für die beachtliche Stabilität, Robustheit und Erweiterbarkeit dieser Systeme. Als ich das Buch „[Thinking Forth](#)“ von LEO BRODIE las, musste ich zu meinem großen Erstaunen feststellen, dass sich die Grundprinzipien der Programmierung konkatenativer Systeme mit den Design-Prinzipien von Kommunikationssystemen weitgehend decken. Wer konkatenativ programmiert fällt automatisch in eine softwaretechnische Denke. In konkatenativen Sprachen stellt sich eine interessante Wechselwirkung zwischen Lesbarkeit (intellektueller Nachvollziehbarkeit) eines Programms und softwaretechnischer Programmkomposition (Programmaufbau) ein – und das schon bei kleinen Programmen. In meinen Augen kann man Softwaretechnik kaum anschaulicher und praktischer erfahrbar machen als mit konkatenativen Programmiersprachen.

Ich hoffe sehr, dass Ihnen der Umgang mit der „Flöte“ namens Consize „musikalische“ Grundfertigkeiten der Programmierung und der Softwaretechnik vermittelt. Sie werden eine Menge lernen. Und wie beim Flötenspiel gilt: Übung macht den Meister. Wenn Sie mit Consize das Programmierspiel verstanden haben, werden Sie sehr viel müheloser weitere Programmiersprachen erfassen und erlernen.

## 1.2 Consize: Installation und Inbetriebnahme

Da Consize in der Programmiersprache Clojure implementiert ist, müssen Sie zunächst Clojure installieren, siehe [clojure.org](http://clojure.org). Die Consize-Implementierung samt der unabdingbar notwendigen Datei `prelude.txt` finden Sie unter [code.google.com/p/consizel](http://code.google.com/p/consizel). Am einfachsten ist es, die beiden Dateien in das Clojure-Verzeichnis zu kopieren.

Wenn Sie Clojure über die Windows-Kommandozeile starten, meldet sich Clojure mit:

```
Clojure 1.4.0
user=>
```

Geben Sie (`require 'consizel`) ein, und nach kurzer Zeit startet Consize.

```
This is Consize -- A Concatenative Programming Language
>
```

Haben Sie insbesondere bei kleinen, langsamen Rechnern Geduld, das Laden der Prelude kann ein wenig dauern.

Sollten Sie Consize mit einem `exit` beenden, kommen Sie zurück zur Clojure-Umgebung. Wollen Sie Consize erneut starten, geben Sie Folgendes ein; nach kurzer Zeit wird sich Consize wieder mit der Begrüßungszeile melden.

```
user=> (consizel/-main)
This is Consize -- A Concatenative Programming Language
>
```

Unter Windows kann eine [Batchdatei](#) sehr hilfreich sein, um sich Tipparbeit beim Starten von Clojure zu sparen.<sup>2</sup> Wenn Sie Consize direkt aufrufen

<sup>2</sup> Siehe z.B. „[Clojure per Kommandozeile in Windows: CLASSPATH-Setting](#)“ auf dem [denkspuren-Blog](#).



fen lassen möchten, geben Sie Folgendes in die Windows-Kommandozeile ein; die Zeile erwartet, dass die Umgebungsvariable CLOJURE auf das aktuelle Verzeichnis der Clojure-Dateien verweist.

```
java -cp %CLOJURE%\*;*; clojure.main consize.clj %2 %3 %4 %5 %6 %7 %8
```

### 1.3 Ein Hinweis für Einsteiger

Wenn Sie ein Neuling in der [Programmierung](#) und in der [Informatik](#) sind, dann werden Ihnen viele Begriffe, Konzepte und Ideen fremd sein. Aus diesem Grund verweise ich an etlichen Stellen im Text auf die Einträge in der [Wikipedia](#). Die Wikipedia bietet eine exzellente Möglichkeit, sich unkompliziert und in kurzer Zeit in verschiedenste Themen und Gebiete einzuarbeiten und an Wissen aufzuholen.

Die Verlinkungen im Text verweisen fast durchgängig auf die [deutschsprachige Wikipedia](#). Wenn Sie auf diese Weise einen Begriff nachschlagen, möchte ich Ihnen sehr ans Herz legen, auch die entsprechende englischsprachige Wikipedia-Seite zu lesen. Das Englische hat in der Informatik aus historischen Gründen eine lange Tradition. Nicht nur sollten Sie die englischsprachigen Fachbegriffe kennen, in vielen Fällen empfinde ich die englischsprachigen Wikipedia-Seiten auch als informativer und umfassender. Wenn Ihnen der Einstieg in die Informatik jedoch neu ist, dann sind meist die Wikipedia-Seiten in der eigenen Muttersprache der bessere Startpunkt. Aus diesem Grund biete ich Ihnen hauptsächlich die deutschsprachigen Wikipedia-Seiten an.

Lassen Sie sich von der Menge der Informationen nicht überfluten. Gerade am Anfang des Textes finden Sie zahlreiche Verweise in die Wikipedia, weil viele für Sie möglicherweise neue Begriffe eingeführt werden. Schon nach wenigen Seiten nimmt die Häufigkeit der Verlinkungen deutlich ab. Haben Sie Geduld mit sich, man kann nicht alles auf einmal lernen. Das Lernen geschieht mehr in Zyklen, phasenweise. Darum sollten Sie nach einer Weile Kapitel oder Textpassagen durchaus noch einmal lesen und punktuell Begriffe nachschlagen, wenn es passt. Sie werden feststellen, nach und nach beginnen die Sachen immer mehr Sinn zu machen und zusammen zu passen. Was anfangs kompliziert und unklar war, ist später, wenn Sie zu der Textstelle zurückkehren, plötzlich sonnenklar. Auch die Erläuterungen in der Wikipedia bekommen Hand und Fuß, und Sie lernen einzuordnen, was dort steht.

---

## 2 Erste Schritte mit Consize: Die Basics

---

### 2.1 Programme sind Texte

Wenn Sie mit einer [Programmiersprache](#) etwas anfangen möchten, dann müssen Sie ein [Programm](#) schreiben, das von den Ausdrucksmitteln der Programmiersprache Gebrauch macht. Ein Programm ist nichts weiter als Text, sprich eine Folge von Einzelzeichen.

Praktisch alle Programmiersprachen verarbeiten Programme in Form von [Textdateien](#). Das heißt, Sie schreiben den Programmtext (auch [Quelltext](#), Quellcode oder kurz nur Code bezeichnet) mit Hilfe eines sogenannten Texteditors und speichern den Programmtext als Datei ab. Ein [Texteditor](#) ist eine Anwendung, die Ihnen das Schreiben und Bearbeiten von Textdateien ermöglicht. Bekannte und weit verbreitete Editoren sind zum Beispiel [jEdit](#) und [Notepad++](#). Wenn Sie Programme in z.B. [Java](#) oder [C#](#) schreiben, dann wird Ihnen ein einfacher Editor meist nicht mehr genügen. Sie greifen dann auf eine sogenannte „[Integrierte Entwicklungsumgebung](#)“ (*Integrated Development Environment*, IDE) wie [Eclipse](#) oder [Visual Studio](#) zurück. Eine IDE ist im Grunde eine Art erweiterter Editor, der zusätzliche, die Programmierarbeit unterstützende Anwendungen integriert. Für Consize reicht jedoch ein einfacher Texteditor zum Schreiben von Programmtexten vollkommen aus.

Consize verarbeitet nicht nur Programme in Form von Textdateien, Sie können mit Consize auch direkt über die [Konsole](#) interagieren. Als Konsole bezeichnet man eine Ein- und Ausgabeeinheit, die eine Schnittstelle zu einer Anwendung herstellt. Unter den verschiedensten [Betriebssystemen](#) wird Ihnen eine Konsole meist in Form eines [Fensters](#) bereit gestellt. In der Regel ist die Interaktion über eine Konsole rein textuell, sprich über einen Fensterausschnitt auf dem Bildschirm und die Tastatur. Die [Maus](#) oder andere Eingabegeräte spielen dabei praktisch keine Rolle. Mittlerweile gibt es viele sehr populäre Programmiersprachen, mit denen Sie direkt über die Konsole interagieren können. Dazu gehören [Python](#), [Ruby](#) und auch [JavaScript](#). Diese Sprachen werden gerne als [Skriptsprachen](#) bezeichnet, was historische Gründe hat.

Ganz gleich, ob Sie Consize eine Textdatei als Programm lesen und verarbeiten lassen oder über die Konsole mit Consize interagieren: es geht immer um Text. Und so ist es wichtig zu wissen, welche Texte der reinen Form nach – der Fachbegriff lautet Syntax – für Consize gültige Programmtexte darstellen. Jede Programmiersprache hat ihre eigene [Syntax](#). Typischerweise sind die Syntax-Regeln (die Regeln, welche Zeichenfolgen gültige Programmtexte sind) in Form einer [formalen Grammatik](#) angegeben.

### 2.2 Wie Consize denkt: Die Ur-Grammatik von Consize

Die Grammatik legt anhand einer Reihe von [Produktionsregeln](#) genauestens fest, welche Zeichen wie aufeinander folgen dürfen. Da allein die Berücksichtigung der Produktionsregeln genügt, um zu entscheiden, ob ein Text (sprich, eine Zeichenfolge) ein gültiges Programm ist oder nicht,

spricht man auch von einer „[kontextfreien Grammatik](#)“. Ob das Programm denn auch ein funktionsfähiges Programm ist, das ist eine Frage der [Semantik](#), der Bedeutung von Programmausdrücken. Ein syntaktisch korrektes Programm muss kein semantisch gültiges Programm sein.

Grammatikregeln werden üblicherweise in der [Erweiterten Backus-Naur-Form](#) (EBNF) oder einer verwandten Schreibweise notiert. Die Notation ist einfach zu verstehen.

Die erlaubten, tatsächlich in einem Programmtext verwendbaren Zeichen heißen [Terminalsymbole](#) und werden durch einfache oder doppelte Anführungszeichen ausgewiesen. So meint ‘%’ das Prozentzeichen, so wie es über die Tastatur eingegeben werden kann. Einige [Steuerzeichen](#) wie z.B. das über die [Tabulator-Taste](#) erzeugte Steuerzeichen werden oft besonders notiert; das Tabulator-Steuerzeichen etwa als ‘\t’.

[Nichtterminalsymbole](#) werden aus einem Mix von Terminal- und Nichtterminalsymbolen definiert. In einer Produktionsregel steht links vom Gleichheitszeichen das zu definierende Nichtterminalsymbol (es wird gerne zur deutlichen Unterscheidung von Terminalsymbolen in spitze Klammern gesetzt), rechts davon eine oder mehrere Alternativen (ein senkrechter Strich „|“ trennt die Alternativen) von Terminalen und/oder Nichtterminalen. Runde Klammern dienen zur Gruppierung. Eckige Klammern weisen ein Terminal- oder Nichtterminalsymbol als optional aus (es darf vorkommen, muss es aber nicht), geschweifte Klammern lassen beliebige Wiederholungen zu (auch keinmal). Jede Regel endet mit einem Semikolon. Den Anfang eines Kommentars leitet ein „(\*“ ein, sein Ende ein „\*)“.

Das Ganze ist am Beispiel leichter nachzuvollziehen. Die Grammatik von Consize ist extrem einfach.

```

<whitespace> = ‘ ’ | ‘\t’ | ‘\n’ | ‘\r’ ;
<separator> = <whitespace> { <whitespace> } ;
<symbol> = ‘a’ | ... | ‘Z’ | ‘0’ | ... | ‘9’ ; (* more generally, any character
      except <whitespace> characters *)
<word> = <symbol> { <symbol> } ;
<program> = [ <word> ] { <separator> <word> } [ <separator> ] ;

```

Das Nichtterminal  $\langle whitespace \rangle$  ([Leerraum](#)) ist definiert, entweder ein Leerzeichen (der Deutlichkeit halber als Unterstrich dargestellt) oder ein Tabulator-Steuerzeichen oder ein Zeilenvorschub ‘\n’ oder ein Wagenrücklauf ‘\r’ zu sein. Der Strich trennt die Alternativen voneinander.

Das Nichtterminal  $\langle separator \rangle$  repräsentiert ein [Trennzeichen](#) und ist definiert als ein Leerraum  $\langle whitespace \rangle$  gefolgt von beliebig vielen weiteren  $\langle whitespace \rangle$ s. Mit anderen Worten: Ein  $\langle separator \rangle$  besteht aus mindestens einem  $\langle whitespace \rangle$ .

Als  $\langle symbol \rangle$  gilt jedes beliebige Terminalsymbol von ‘a’ bis ‘Z’ und von ‘0’ bis ‘9’. Die Punkte „...“ sind hier nur als verkürzende Schreibweise gedacht, statt alle Terminalsymbole ausdrücklich hinschreiben zu müssen. Der Kommentar weist darauf hin, dass die Produktionsregel für  $\langle symbol \rangle$  sogar noch weiter zu fassen ist: Als  $\langle symbol \rangle$  kommt jedes Terminalsymbol in Frage, das nicht als  $\langle whitespace \rangle$  gilt.

Ein Wort  $\langle word \rangle$  in Consize besteht aus mindestens einem  $\langle symbol \rangle$ . Und ein Programm  $\langle program \rangle$  beginnt optional mit einem  $\langle word \rangle$ , es schließt

optional mit  $\langle separator \rangle$  und erlaubt dazwischen beliebig viele Wiederholungen aus  $\langle separator \rangle$  und  $\langle word \rangle$ .

Wenn Sie sich die Regel für ein Programm  $\langle program \rangle$  durch den Kopf gehen lassen, so werden Sie eine interessante Beobachtung machen: Es ist unmöglich, ein syntaktisch ungültiges Programm für Consize zu verfassen. Welchen Programmtext auch immer Sie Consize vorsetzen: Ein Programm wird schlicht an den Grenzen von Leerräumen in Wörter zerlegt. Im Extremfall besteht ein Consize-Programm aus nicht einmal einem einzigen Zeichen.

Da Consize einzig an den Wörtern eines Programms interessiert ist, fällt die Analyse und Zerlegung eines Programmtextes in Wörter sehr leicht aus. Es gibt kaum eine andere Programmiersprache, die eine derart primitive Syntax hat. Man könnte die Syntax auch als extrem robust bezeichnen, gemäß der zweiten Hälfte des von [JONATHAN POSTEL](#) formulierten [Robustheitsgrundsatzes](#): „be liberal in what you accept from others“. Ganz so „liberal“ wird Consize jedoch nicht bleiben, weil zu viel Freiheit keine Strukturen bietet.

## 2.3 Das Parsen eines Programms

Wenn Consize eine Textdatei lädt und den dort enthaltenen Text als Programm ausführen soll, dann geht das nicht sofort. Ein paar Vorarbeiten sind notwendig. Der Text muss daraufhin untersucht werden, ob er der Grammatik entspricht. Und erst wenn das der Fall ist, dann kann der Programmtext weiter analysiert und verarbeitet werden bis er schlussendlich als Programm ausgeführt wird. Die Aufgaben von der Grammatikanalyse bis hin zur Vorbereitung der Programmausführung werden als „Parsen“ (*parsing*) bezeichnet. Der dafür verantwortliche Programmteil heißt „Parser“.

Nun ist die Grammatik von Consize so einfach, dass sich das Parsen auf zwei Schritte beschränkt. Im ersten Schritt entfernt Consize aus dem Programmtext Kommentare. Ein [Kommentar](#) ist ein für die Programmausführung vollkommen irrelevanter Textteil, den ein Programmierer bzw. eine Programmiererin nutzen kann, um Anmerkungen für sich und andere Leser(innen) des Programmtexts zu hinterlassen.

Im zweiten Schritt zerlegt Consize den Programmtext in logische Einheiten, sogenannte „Token“. Diesen Vorgang übernimmt ein Programm, das „Tokenizer“ heißt; der Tokenizer ist eine ganz einfache Form eines Parsers. Im Fall von Consize zerlegt der Tokenizer den Programmtext einfach an den Leerstellen ( $\langle separator \rangle$ ) in eine Folge von Wörtern ( $\langle word \rangle$ ). Die Token sind Wörter.

Machen Sie folgende einfache Übung: Legen Sie eine Textdatei mit nachstehendem Inhalt an und speichern Sie den Text in einer Datei namens `test.txt` und zwar in dem Verzeichnis, in dem sich auch Consize befindet. Wenn Sie wollen, fügen Sie beginnende Leerzeichen hinzu oder verändern Sie den Text beliebig.

```
I'm a syntactically valid line of code! % though meaningless
Bye bye
```

Starten Sie Consize und geben Sie folgendes ein:

```
> clear \ test.txt slurp
I'm a syntactically valid line of code! % though meaningless
```

Bye bye

Consize liest („schlüpft“, *slurp*) den Inhalt der Datei `test.txt` ein. Das Ergebnis sieht exakt so aus, wie das, was Sie mit dem Editor in die Datei geschrieben haben.

Nun entfernen wir mit `uncomment` die Kommentare. In Consizes beginnt ein Kommentar mit dem Prozentzeichen „%“ und endet am Ende der Zeile.

```
> uncomment
I'm a syntactically valid line of code!
```

Bye bye

Lassen Sie sich von der eingefügten „Extrazeile“ nicht irritieren; je nach verwendetem Betriebssystem ist die Extrazeile möglicherweise bei Ihnen auch nicht zu sehen.<sup>1</sup> Entscheidend ist, dass der Kommentar verschwunden ist.

Die Zerlegung des entkommentierten Programms geschieht mit `tokenize`.

```
> tokenize
[ I'm a syntactically valid line of code! Bye bye ]
```

Was Sie hier sehen, ist das Ergebnis der Zerlegung des entkommentierten Programms in eine Folge von neun Wörtern. Sie bekommen das Ergebnis in Form einer Datenstruktur präsentiert, die sich Stapel nennt; das zeigen die eckigen Klammern an.

Sie können das erste Wort `I'm` mit `unpush` vom Stapel holen und dahinter ablegen.

```
> unpush
[ a syntactically valid line of code! Bye bye ] I'm
```

Mit `type` können Sie den Datentypen von `I'm` ermitteln; `type` entfernt das Wort `I'm` und gibt das Ergebnis mit der Angabe `wrđ` bekannt. Es handelt sich also tatsächlich um ein Wort!

```
> type
[ a syntactically valid line of code! Bye bye ] wrđ
```

Doch ich greife vor. Wir werden uns in Kürze mit den Datentypen beschäftigen, die Consizes anbietet.

Hätte der Programmtext nicht den Grammatikregeln entsprochen, so hätte Consizes den Text als syntaktisch ungültig zurückgewiesen. Wie Sie aber wissen, ist die Grammatik von Consizes derart primitiv und grundlegend, dass es keine syntaktisch ungültigen Programmtexte geben kann. Consizes wird den Inhalt jeder Datei erfolgreich mit `tokenize` verarbeiten. Bei so gut wie allen anderen Programmiersprachen ist das nicht so. Die Grammatiken sind komplizierter und verlangen nach Strukturen, die eingehalten werden wollen. Fordert eine Grammatik beispielsweise, dass einer öffnenden Klammer ‘(’ im Programmtext auch immer eine schließende Klammer ‘)’ folgt, dann ist ein Kurzprogramm wie z.B. „( 1 2 3“ ungültig: die schließende Klammer fehlt.

In der Tat ist die Grammatik von Consizes so einfach, dass sie schon wieder problematisch ist: Ohne strukturbildende Ausdrucksmittel wie z.B.

---

<sup>1</sup> Ich verwende Consizes zusammen mit Microsoft Windows.

Klammern, sind Consize-Programme kaum für Menschen lesbar. Und auch das Schreiben von Consize-Programmen ohne Strukturmittel ist wenig Spaßig.

Doch es gibt einen netten Ausweg aus dieser Situation: Wohl aber können wir Klammern, wie z.B. [ und ] oder { und } als Wörter in Consize verwenden. Wenn Sie sonst eine geklammerte Zahlenfolge als „[1 2 3]“ schreiben würden, müssen Sie jetzt nur Leerräume nutzen und die Klammern in „[ 1 2 3 ]“ werden zu eigenständigen Wörtern.

Damit können wir uns eines Tricks bedienen: Wir triggern mit diesen Wörtern, wie z.B. bei einer öffnenden eckigen Klammer [, ein Consize-Programm, das die passende schließende eckige Klammer sucht, die dazwischen liegenden Wörter als Daten interpretiert und in eine geeignete Datenstruktur packt. Damit simulieren wir eine Grammatikregel für eckige Klammern.

Wie man einen „richtigen“ Parser schreibt, statt mit einem „Hack“ eine Grammatikregel zu simulieren, das lernen Sie in einem anderen Kapitel – das ist ein Thema für fortgeschrittene Consize-Programmierer(innen).

## 2.4 Was Consize versteht: Die erweiterte Grammatik

Obwohl die Ur-Grammatik von Consize so überaus primitiv ist, können wir mit dem eben erwähnten Trick nachträglich Grammatikregeln simulieren. Dieser Trick ist im sogenannten „Präludium“ (Vorspiel) zu Consize programmiert. Wir verwenden fortan den englischen Begriff „Prelude“.

In der Prelude ist eine Vielzahl an kleinen Consize-Programmen abgelegt, die das Arbeiten mit Consize praktischer und angenehmer machen – Consize-Programme, die Consize erweitern. Darunter eben auch die „Erweiterungen“ der Grammatik. Kap. 5 befasst sich ausführlich mit der Prelude.

Sie dürfen sich die erweiterte Grammatik ungefähr wie folgt vorstellen; die Grammatik ist nicht vollständig und vereinfacht, aber sie enthält wichtige Regeln von Consize.

$$\begin{aligned}\langle sequence \rangle &= '[ \{ \langle separator \rangle \langle item \rangle \} \langle separator \rangle ]' ; \\ \langle mapping \rangle &= '{ \{ \langle separator \rangle \langle item \rangle \langle separator \rangle \langle item \rangle \} \langle separator \rangle }' ; \\ \langle item \rangle &= \langle word \rangle \mid \langle sequence \rangle \mid \langle mapping \rangle ; \\ \langle program \rangle &= [ \langle item \rangle ] \{ \langle separator \rangle \langle item \rangle \} [ \langle separator \rangle ] ;\end{aligned}$$

In der von Consize simulierten Grammatik gibt es Folgen ( $\langle sequence \rangle$ ) von Elementen ( $\langle item \rangle$ ), einmal mit eckigen, einmal mit Runden Klammern, und Mappings mit geschweiften Klammern. Mappings unterscheiden sich von Folgen dadurch, dass sie paarweise Items gruppieren. Überall sorgen Leerräume ( $\langle separator \rangle$ ) dafür, dass weder die Klammern noch die Elemente sich „berühren“ können. Die Logik der Ur-Grammatik bleibt damit erhalten: Alles kann letztlich als eine Folge von Wörtern interpretiert werden.

Die Grammatik hat eine Besonderheit: in ihr gibt es gegenseitige Abhängigkeiten – man spricht von wechselseitiger [Rekursion](#). So bezieht sich die Regel zu  $\langle sequence \rangle$  auf  $\langle item \rangle$ ,  $\langle item \rangle$  wiederum kann eine  $\langle sequence \rangle$  sein. Auf diese Weise beschreibt die Grammatik Verschachtlungen. Die Zeichenfolge

```
[ 1 2 [ { 3 4 x y } z ] ]
```

ist eine gültige *sequence*, die ihrerseits aus zwei Wörtern und einer weiteren Folge mit eckigen Klammern besteht, die wiederum ein Mapping und ein Wort beinhaltet.

Ein Programm in Consize besteht zwar nach wie vor – wie in der Ur-Grammatik – aus einer Folge von Wörtern, doch die Grammatik-Erweiterungen fordern den balancierten Gebrauch der Wörter ein, die Klammern darstellen. Öffnende und schließende Klammern müssen Verschachtelungen beschreiben. Bei geschweiften Klammern ist sogar stets eine gerade Anzahl an eingebetteten Elementen gefordert.

Sie wissen noch nicht, was mit den eckigen und den geschweiften Klammern gemeint ist, aber Namensgebungen wie *sequence* und *mapping* sind nicht zufällig, sondern absichtlich so gewählt. Wenn Sie Programmiererfahrung haben, werden Sie eher eine Idee haben, was Sequenzen und Mappings sein können, als wenn Consize Ihre erste Programmiersprache ist. Ein Programmierprofi wird immer einen Blick in die Grammatikregeln einer ihm neuen Programmiersprache werfen, um sich zu orientieren.

Das nächste Unterkapitel wird Sie in die Datenstrukturen von Consize einführen. Die Datentypen werden genauso dargestellt, wie Sie sie in Consize eintippen können – sofern die Prelude geladen ist. Die Notationen orientieren sich an den Grammatiken für Sequenzen und Mappings!

## 2.5 Datenstrukturen

Obwohl Consize laut „Ur-Grammatik“ nur Wörter und keine Klammern kennt (Klammern rüstet die Prelude nach), kann man dennoch Strukturen aufbauen. Eben nicht auf direkte Weise, sondern mit Wörtern, die Datenstrukturen erzeugen.

Beginnen wir mit der wichtigsten Datenstruktur in Consize, dem Stapel (*stack*). Der Name dieser Datenstruktur ist der Vorstellung entlehnt, die wir mit einem Stapel z.B. von Büchern, Zeitschriften oder Tellern verbinden. Auf einem Stapel kann man etwas ablegen oder wieder entfernen und zwar immer nur „von oben“.

Diese Anschauung wird auf wenige elementare Operationen reduziert: Mit **emptystack** wird ein leerer Stapel erzeugt, mit **push** ein Element auf dem Stapel abgelegt, mit **pop** der Stapel um das oberste Element reduziert, **top** gibt das oberste Element zurück.

```
| item4 | <- top of stack
| item3 | \
| item2 | > rest of stack (pop)
| item1 | /
+-----+
```

Schauen wir uns die Befehle einzeln an in der Interaktion mit Consize.

**emptystack** erzeugt einen leeren Stapel. Consize stellt einen Stapel mit Hilfe eckiger Klammern dar. Ist der Stapel leer, so trennt einzig ein Leerzeichen die öffnende von der schließenden Klammer.

```
> emptystack
[ ]
```

Man muss wissen, dass das „obere“ Ende des Stapels links ist. Wenn wir ein Element auf den Stapel **pushen**, wird es von links her auf dem Stapel abgelegt.

```
> 3 push
[ 3 ]
> 4 push
[ 4 3 ]
> hello push
[ hello 4 3 ]
```

**pop** entfernt das oberste, sprich das Element ganz links vom Stapel. Das kann man so lange tun, bis kein Element mehr auf dem Stapel ist. **top** holt das oberste („linkeste“) Element vom Stapel, was den Stapel vernichtet.

```
> pop
[ 4 3 ]
> top
4
```

Vielleicht haben Sie sich schon diese Frage gestellt: Wie bekommt man eigentlich ein Wort wie **push** selbst als Element auf einen Stapel abgelegt?

```
> emptystack \ push push
4 [ push ]
```

Das Wort **\** hat eine besondere Aufgabe: Es hebt die Funktion auf, die das nachfolgende Wort möglicherweise hat. So kann **push** auf einen leeren Stapel abgelegt werden.

Da Sie hier bereits mit der über die Prelude erweiterten Version von Consize arbeiten, gibt es auch die Möglichkeit, Stapel direkt in der Notation mit den eckigen Klammern einzugeben.

```
> [ hello 4 3 ]
4 [ push ] [ hello 4 3 ]
```

Zwei Stapel lassen sich mit **concat** „konkatenieren“ (verbinden, zusammenfügen).

```
> concat
4 [ push hello 4 3 ]
```

Consize ist eine konkatenative Programmiersprache. Hinter der Konkatenation, dem Zusammenfügen zweier Stapel verbirgt sich ein wichtiges Grundprinzip der Arbeitsweise von Consize. Wie werden darauf in Kap. 2.7 näher eingehen.

Mit **reverse** lassen sich die Inhalte des Stapels umkehren: Das letzte Element im Stapel wandert nach oben, das vorletzte an die zweitoberste Stelle usw. Das ändert nichts daran, dass der Stapel immer noch von links befüllt wird. Es dreht sich also nicht der Stapel um, sondern sein Inhalt wird „umgekehrt“.

```
> reverse
4 [ 3 4 hello push ]
```

Sie haben eine Menge gelernt: Sie erzeugen mit **emptystack** einen leeren Stapel, können dem Stapel mit **push** Elemente hinzufügen und mit **pop** wieder entfernen; **top** liefert das oberste Element eines Stapels. Mit **concat** werden zwei Stapel miteinander verbunden (konkateniert), mit **reverse** sein Inhalt „umgekehrt“.



## 2.6 Datenstapel und Programmstapel

Consize hat eine denkbar einfache Virtuelle Maschine (VM). Sie besteht aus zwei Stapeln und einem Wörterbuch (*dictionary*). Eine sehr einfache Abarbeitungsvorschrift regelt das Zusammenspiel der Stapel und den Gebrauch des Nachschlagewerks.

Die beiden Stapel heißen Datenstapel (*data stack*) und Aufruf- oder Programmstapel (*call stack*). Dabei gibt es eine Konvention: Aus praktischen Gründen werden die beiden Stapel „liegend“ dargestellt, wobei der Datastack sein oberes Ende rechts und der Callstack sein oberes Ende links hat. Wenn der Datastack und der Callstack gemeinsam dargestellt werden, stoßen die Kopffenden aneinander; links ist dann der Datastack, rechts der Callstack.

```
+-----+ +-----+
| Datastack   Callstack |
+-----+ +-----+
```

Wenn wir ein Programm aus Wörtern schreiben, dann stellt die Folge von Wörtern die Situation auf dem Callstack dar. Das Programm

```
emptystack 2 push 3 push
```

stellt sich bei leerem Datastack wie folgt dar:

```
+-- +-----+
|   emptystack 2 push 3 push |
+-- +-----+
```

Der Einfachheit halber notieren wir den Data- und den Callstack ohne die umgrenzenden Linien und nutzen einen Trennstrich „|“, um den Übergang vom Datastack zum Callstack anzuzeigen:

```
| emptystack 2 push 3 push
```

Das Programm wird schrittweise abgearbeitet. Ein Wort nach dem anderen wird vom Callstack genommen, bis dass der Callstack leer ist. Dem entspricht in der Darstellung eine Abarbeitung der Wörter auf dem Callstack von links nach rechts.

Jedes Wort, das vom linken Ende des Callstacks „genommen“ wird, hat eine Auswirkung auf die VM von Consize. Die Bedeutung eines Wortes wird im Wörterbuch nachgeschlagen. **emptystack** zum Beispiel legt einen leeren Stapel oben auf dem Datastack ab. Nach **emptystack** stellt sich die Situation auf dem Data-/Callstack wie folgt dar:

```
[ ] | 2 push 3 push
```

Das Wort 2 (nicht Zahl, sondern Wort!) ist ein „neutrales“ Wort. Es wandert direkt vom Callstack rüber auf den Datastack.

```
[ ] 2 | push 3 push
```

Das Wort **push** erwartet oben auf dem Datastack irgendein Element und darunter einen Stapel. **push** legt das Element oben auf dem Stapel ab. Denken Sie daran, dass ein Stapel auf dem Datastack mit seinem „offenen“ Ende nach links liegt.

```
[ 2 ] | 3 push
```

Das Wort 3 wandert wie 2 direkt auf den Datastack.

```
[ 2 ] 3 | push
```

Nun legt das letzte `push` das Wort `3` auf dem Stapel `[ 2 ]` als oberstes Element ab.

```
[ 3 2 ] |
```

Der Callstack ist nun vollständig abgearbeitet und der Datastack beinhaltet das Ergebnis der Abarbeitung des Programms auf dem Callstack. Voilà! Sie haben einen ersten Programmdurchlauf im Einzelschrittmodus durch Consize mitgemacht. Schwieriger wird es kaum mehr.

Ist Ihnen etwas aufgefallen? Wenn Sie interaktiv mit Consize über die Console arbeiten, geht all das, was Sie an Wörtern eingeben auf den Callstack. Als Ergebnis der Abarbeitung zeigt Ihnen Consize den Datastack an.

Und noch etwas ist Ihnen sicher aufgefallen. Es macht für Consize oftmals keinen Unterschied, ob Sie in der Konsole Wörter einzeln eingeben und direkt mit einem ENTER die Abarbeitung anstoßen oder ob Sie mehrere Wörter durch Leerzeichen getrennt in einer Zeile an Consize übergeben. Das Ergebnis ändert sich deshalb nicht. Es gibt Ausnahmen von dieser Regel wie z.B. das Wort `\`, dem immer ein Wort unmittelbar nachfolgen muss. Von solchen Ausnahmen abgesehen ist es ein Leichtes, ein Programm Wort um Wort einzugeben und die Abarbeitung so schrittweise zu verfolgen. So lässt sich selbst das komplizierteste Programm gleichermaßen im Einzelschrittmodus nachvollziehen.

## 2.7 Atomare und nicht-atomare Wörter

Wenn Consize ein Wort vom offenen Ende des Callstacks nimmt, schlägt Consize die Bedeutung des Wortes in einem Wörterbuch (*dictionary*) nach. Findet sich das Wort nicht im Wörterbuch, so landet das Wort unversehens auf dem Datastack – die Details erfahren Sie am Ende dieses Kapitelabschnitts.

Findet Consize das Wort im Wörterbuch, so gibt es zwei Möglichkeiten: Entweder ist das Wort ein atomares Wort (*atomic word*). Dann ist im Wörterbuch für das Wort eine Funktion hinterlegt, die – angewendet auf den Datastack – die Bedeutung des Wortes umsetzt. Oder das Wort ist ein nicht-atomares Wort (*non-atomic word*). Das Wörterbuch hat zu dem Wort einen Stapel als Eintrag, dessen Inhalt die Bedeutung des Wortes definiert. Wir nennen diesen Stapel auch „Quotierung“ (*quotation*). Generell bezeichnen wir ein in einem Stapel „verpacktes“ Programm als Quotierung.

Schauen wir uns das am Beispiel an. Das Wort `rot` verändert die Position der obersten drei Werte auf dem Datastack. Das dritte Element von oben „rotiert“ an die führende Stelle ganz oben auf dem Stapel, was die vormalig obersten zwei Elemente absteigen lässt.

```
> clear x y z
x y z
> rot
y z x
```

Das einleitende `clear` räumt den Datastack auf und lässt einen leeren Datastack zurück. Wir werden sehr oft bei den Beispielen `clear` verwenden, um eine definierte Situation auf dem Datastack zu haben.

Mittels `\ rot source` können Sie das Wort `rot` im Wörterbuch nachschlagen. Das Ergebnis landet nicht auf dem Datastack, sondern wird über die Konsole ausgegeben, bevor der Inhalt des Datastacks angezeigt wird. Der Datastack bleibt unverändert.

```
> \ rot source
<fct>
y z x
```

Die Ausgabe `<fct>` besagt, dass zu `rot` eine Funktion im Wörterbuch eingetragen ist; `rot` ist also ein atomares Wort, man sagt auch primitives Wort.

Erinnern Sie sich noch, warum Sie bei `source` dem Wort `rot` ein `\` voranstellen müssen? Das Quotierungswort `\` „zieht“ das nachfolgende Wort direkt auf den Datastack und verhindert damit ein Nachschlagen im Wörterbuch samt Ausführung des Wortes.

Ein Beispiel für ein nicht-atomares Wort ist `-rot`. Es setzt sich aus zwei `rot`-Wörtern zusammen.

```
> \ -rot source
[ rot rot ]
y z x
```

Wann immer im Wörterbuch eine Quotierung die Bedeutung eines Wortes angibt, wird die Quotierung mit dem Callstack konkateniert (aneinander gefügt) und als neuer Callstack betrachtet. Dem Augenschein nach sieht es so aus als ob `-rot` auf dem Callstack durch `rot rot` ersetzt wird; der Fachausdruck lautet „substituiert“.

Die Auswirkungen von `-rot` kann man sich durch ein zweifaches `rot` veranschaulichen. Die Eingabe von `-rot`

```
> -rot
x y z
```

wird von Consize durch `rot rot` ersetzt. Nur atomare Wörter kann Consize direkt ausführen. Die Quotierungen zu nicht-atomaren Wörtern werden mit dem Callstack konkateniert.

Nicht-atomare Wörter stellen „benannte Abstraktionen“ dar. Das Wort `-rot` abstrahiert die Wortfolge `[ rot rot ]`. Wenn man so möchte ist `-rot` der Name für die Abstraktion `[ rot rot ]`; darum die Bezeichnung als „benannte Abstraktion“.

Wenn Sie ein Wort nachschlagen, das nicht im Wörterbuch verzeichnet ist, meldet Ihnen Consize das.

```
> \ x source
word-not-found
x y z
```

Wörter, die nicht im Wörterbuch stehen, legt Consize standardmäßig auf dem Datenstack ab. Das ist jedoch nur die halbe Wahrheit. Außerdem – und das haben wir eben unterschlagen – legt Consize das Wort `read-word` auf dem Callstack ab. Was nun mit dem Wort auf dem Datastack passiert hängt davon ab, welche Abstraktion für `read-word` im Wörterbuch hinterlegt ist.

```
> \ read-word source
[ ]
x y z
```

Mit `read-word` ist ein leerer Stapel assoziiert, der konkateniert mit dem Callstack den Callstack unverändert lässt. Die Definition von `read-word` belässt folglich unbekannte Wörter auf dem Datastack. Sie können das Wort `read-word` anpassen, um besondere Wörter gesondert behandeln zu können.

## 2.8 Mappings

Das Wörterbuch der Consizes-VM ist nur ein Sonderfall einer Datenstruktur, die wir allgemeiner als Mapping (Abbildung) bezeichnen. Ein Mapping assoziiert einen Schlüsselwert (*key value*) mit einem Zielwert (*target value*). Jeder Schlüsselwert kann nur genau einmal vorkommen, somit ist die Abbildung auf einen Zielwert immer eindeutig. Wir nennen ein Mapping dann Wörterbuch, wenn alle seine Schlüsselwerte Wörter sind.

Ein Mapping wird in Consize mit geschweiften Klammern dargestellt. Innerhalb der Klammern stehen in freier Abfolge Paare von Schlüssel- und Zielwerten; das Leerzeichen dient in gewohnter Manier als Trennzeichen.

Es gibt in Consize kein direktes Wort, um ein leeres Mapping anzulegen. Man muss zunächst einen leeren Stapel auf dem Datastack erzeugen, den ein nachfolgendes `mapping` in ein Mapping verwandelt.

```
> emptystack mapping
{ }
```

Um einen Schlüsselwert mit einem Zielwert zu assoziieren, muss zunächst der Zielwert, dann der Schlüsselwert und zu guter Letzt das Mapping auf dem Datastack liegen, das die Assoziation aufnehmen soll. Dabei kommt uns `rot` zu Hilfe.

```
> 1 mon rot
1 mon { }
> assoc
{ mon 1 }
```

Und so lassen sich weitere Paare von Schlüssel- und Zielwerten hinzufügen.

```
> 3 wed rot assoc
{ wed 3 mon 1 }
```

Auch ein nicht-leerer Stapel kann, sofern er eine gerade Anzahl an Werten hat, mit `mapping` in ein Mapping überführt werden.

```
> emptystack 2 push tue push
{ wed 3 mon 1 } [ tue 2 ]
```

Um eine vollständige Aufführung der Wochentage einer Arbeitswoche zu haben, fügen wir noch die fehlenden Tage hinzu, bevor wir aus dem Stapel ein Mapping machen.

```
> 5 push fri push 4 push thu push
{ wed 3 mon 1 } [ thu 4 fri 5 tue 2 ]
> mapping
{ wed 3 mon 1 } { thu 4 fri 5 tue 2 }
```

Die beiden Mappings lassen sich mit `merge` zu einem zusammenführen. Die Reihenfolge der Auflistung der Schlüssel/Ziel-Paare ist ohne Bedeutung und kann abweichen von dem, was Sie hier gezeigt bekommen.

```
> merge
{ tue 2 fri 5 thu 4 wed 3 mon 1 }
```

Man kann nun das Mapping befragen, welches Wort beispielsweise mit dem Wort `wed` assoziiert ist. Dazu dient das Wort `get`, das erst einen Schlüsselwert, dann ein Mapping und zuoberst auf dem Stapel einen Wert erwartet, der das Ergebnis ist, falls der Schlüsselwert nicht im Mapping vorhanden ist. `rot` wird uns wieder helfen, die gewünschte Ordnung auf dem Datastack herzustellen.

```
> key-not-found wed -rot
wed { tue 2 fri 5 thu 4 wed 3 mon 1 } key-not-found
> get
3
```

Das über die Prelude erweiterte Consize erlaubt die Eingabe von Mappings auch direkt über geschweifte Klammern.

```
> { 1 a 2 b 3 c }
3 { 1 a 2 b 3 c }
```

Mit `dissoc` lässt sich eine Schlüssel-/Zielwert-Bindung entfernen. Dazu muss das Mapping zuoberst auf dem Datastack liegen, der betreffende Schlüssel darunter. Hier nutzen wir gleich das verbliebene Wort `3` auf dem Stapel.

```
> dissoc
{ 1 a 2 b }
```

Das Wort `keys` liefert alle Schlüsselwerte eines Mappings in einem Stapel zurück – ohne irgendeine Garantie, in welcher Reihenfolge die Schlüssel im Stapel aufgeführt sind.

```
{ 1 a 2 b }
> keys
[ 1 2 ]
```

Mappings sind eine sehr leistungsfähige und bedeutsame Datenstruktur. In manchen Programmiersprachen tragen sie einen anderen Namen und heißen dort etwa *Map* oder assoziative Arrays. In JavaScript sind Mappings die Grundlage für Objekte, ähnlich in Python.

Wenn Sie sich das Wörterbuch der VM von Consize anschauen wollen – es ist nichts anderes als ein Mapping –, dann legt Ihnen `get-dict` das Wörterbuch auf dem Datastack ab. Seien Sie nicht erschreckt über den unübersichtlichen Datenwust. Sie werden im Laufe der Zeit sehr genau verstehen, was sich so alles warum in dem Wörterbuch der VM befindet.

Übrigens sind als Schlüssel- wie auch als Zielwerte beliebige Werte für Mappings erlaubt. So darf sogar ein Stapel oder ein Mapping als Schlüsselwert verwendet werden.

## 2.9 Was sind Datenstrukturen?

Sie haben nun die drei Arten von Daten kennengelernt, die Consize unterstützt: Wörter und in dem Zusammenhang Funktionen, Stapel und

Mappings – man spricht auch von [Datentypen](#). Dazu kommt noch ein Datentyp namens *Nil* (für „Nichts“). Mehr Datentypen kennt Consize von Haus aus nicht.

Wörter und Funktionen sind Vertreter der einfachen oder auch primitiven Datentypen (*primitive datatypes*). Sie repräsentieren ein Datum, sie stehen sozusagen für sich selbst. Stapel und Mappings dagegen sind zwei Vertreter der zusammengesetzten Datentypen (*compound datatypes* oder auch *composite datatypes*); mit ihnen wird Daten eine Struktur gegeben, weshalb man auch von [Datenstrukturen](#) spricht.

Die Struktur der Daten deutet sich in den verwendeten Notationen an. Die eckigen Klammern für Stapel markieren den Anfang und das Ende der Stapel-Datenstruktur; ebenso markieren die geschweiften Klammern Anfang und Ende eines Mappings. Wie Consize die Daten jedoch intern im Speicher organisiert, das ist Ihnen verborgen. Darauf haben Sie keinen Zugriff. Sie wissen lediglich, das Ihnen für den Umgang mit Stapel und Mappings ein paar Worte zur Verfügung stehen.

Diese Abschottung von den Interna der eingebauten Datenstrukturen ist in vielen Programmiersprachen so gewollt. Sie sind Teil der Sprache und man möchte verhindern, dass Sie damit irgendwelchen Schindluder treiben.

Was Sie jedoch wissen sollten, ist, wie „teuer“ Ihnen der Gebrauch der Datenstrukturen kommt. Es gibt immer einen Preis zu zahlen und zwar in der Währung „Zeit“ und in der Währung „Speicherverbrauch“. Da Ihnen in vielen Programmiersprachen mehr Datenstrukturen als Stapel und Mappings zur Verfügung stehen, haben Sie nicht selten die Qual der Wahl: Welche Datenstruktur wollen Sie wofür nehmen? Das ist immer eine Frage nach: Wie schnell bedient mich die Datenstruktur für meinen Einsatzzweck, und wieviel Speicher frisst sie mir weg?

Mit den einzelnen Namen wie Stapel, Liste, Array, Queue – um nur einige zu nennen – sind verschiedene Kosten für Zeit und Speicher verbunden. Dabei lassen sich all diese Datenstrukturen prinzipiell über exakt die gleichen Operationen (Wörter) ansprechen.

Zum Beispiel können Daten von einem Stapel immer nur „von oben“ abgegriffen werden. Egal wie groß der Stapel ist, das oberste Element ist immer „sofort“ erhältlich, d.h. mit einem `top` erreichbar. Das „unterste“ Element in einem Stapel kann nur sukzessive über eine Reihe von `pops` und einem abschließenden `top` erreicht werden. Es werden genau so viele Wörter benötigt, um an das unterste Element zu kommen, wie es Elemente auf dem Stapel gibt. Für den Zugriff auf das  $n$ -te Element auf dem Stapel (von oben gezählt) benötigt man genau  $n$  Wörter, um an dieses Element heranzukommen.

Bei einer Liste ist der Zugriff auf das erste und das letzte Element gleichermaßen schnell, ebenso auf das zweite und vorletzte usw. Eine Liste ist eine Art Stapel, der von beiden Seiten gleichermaßen gut zugreifbar ist, was in manchen Fällen von Vorteil ist.

Bei einem Array (manchmal auch Vektor genannt) kann man über einen Index auf die Elemente zugreifen. Das erste, zweite, dritte Element, allgemein das  $n$ -te Element, ist immer in der gleichen Zeit abrufbar.

Dies soll Ihnen nur einen Eindruck geben, wie sehr unterschiedliche Datenstrukturen unterschiedliche Zugriffszeiten auf die durch sie organisierten Daten mit sich bringen. Und dies, ohne dass Sie etwas über die Interna

der Datenhaltung zu wissen brauchen.

Jede Datenstruktur hat ihre Vor- und Nachteile. Consize ist eine stapelbasierte Programmiersprache. Es ist das Auszeichnungsmerkmal von Consize, dass es zur Daten- und Programmhaltung nicht mehr und genau nur zwei Stapel benötigt – und ein Mapping als Wörterbuch. Für die stapelbasierten Operationen sind Stapel, es mag kaum verwundern, eben optimal.

Ein Mapping kann man auch über einen Stapel simulieren, aber dann werden die Zugriffszeiten auf die über die Schlüssel assoziierten Zielwerte sehr ungünstig. Consize würde Ihnen zu langsam werden, sie hätten keinen Spaß daran. Darum habe ich mich entschieden, Mappings in ihrer Reinform in Consize mit aufzunehmen. Der Zugriff auf assoziierte Werte ist sehr schnell und günstig. Aber nicht nur deshalb. Mit Mappings hat man eine sehr hilfreiche Datenstruktur, mit der sich so manch nettes Feature in Consize umsetzen lässt.

Ein Rat an dieser Stelle: Lernen Sie in jeder neuen Programmiersprache die verfügbaren Datentypen kennen und finden Sie heraus, welche Vorzüge und welche Nachteile jede Datenstruktur mit sich bringt. Informatiker geben diese „Kosten“ in Sachen Laufzeit ([Zeitkomplexität](#)) und Speicherbedarf ([Platzkomplexität](#)) mit Hilfe der „[Big-O-Notation](#)“ an. Zu den Standard-Datenstrukturen gehören Stapeln, Listen, Arrays (Vektoren), Mappings (Dictionaries, Assoziative Arrays), Queues, Bäume und Graphen – darüber sollten Sie Bescheid wissen.

Eine wichtige Anmerkung noch: Der Stapel ist der einzige Datentyp in Consize für eine geordnete Ansammlung (*collection*) von Elementen. Und oftmals wird er auch genau dafür gebraucht: Für eine Folge (*sequence*) von Elementen. Da ist es eher unerheblich, ob dafür ein Stapel verwendet wird oder nicht – Consize ist da alternativlos. Stünden weitere Datenstrukturen zur Verfügung, dann wären beispielsweise Listen oder Arrays oft eine geeignetere Wahl für Sequenzen.

Darum wundern Sie sich bitte nicht: Wenn ich von Folgen oder Sequenzen rede, dann ist mir die Tatsache nicht entscheidend, dass die Folge bzw. Sequenz in Consize durch einen Stapel abgebildet wird. Dann abstrahiere ich von der konkreten, zugrunde liegenden Datenstruktur. Und ein anderer Name wird immer wieder fallen: der der Quotierung. Eine Quotierung ist eine Sequenz, deren Elemente ein Programm darstellen.

Ein Mapping realisiert ebenfalls eine Datensammlung (*collection*), aber eine ungeordnete. Die Ordnung der Schlüssel/Ziel-Paare ist ohne Bedeutung, darauf kommt es bei Mappings nicht an.

---

## 3 Die VM von Consize: Der Kernel

---

Dieses Kapitel legt Ihnen die Details des Consize-[Interpreters](#) offen. Ein Interpreter kann als [Virtuelle Maschine](#) (VM) bezeichnet werden, weshalb auch immer wieder von der Consize-VM gesprochen werden wird.

Die Consize-VM hat um die 50 atomare Wörter, die alle auf die eine oder andere Weise mit den Elementen des Datastacks arbeiten und/oder dort neue Werte ablegen. Es gibt lediglich vier Wörter, die sich nicht allein durch Manipulationen des Datastacks verstehen lassen: `get-dict`, `set-dict`, `call/cc`, `continue`. Diese vier Wörter offenbaren die Maschinerie der Virtuellen Maschine aus Datastack, Callstack und Wörterbuch. Die Logik dieser Maschinerie legt Consize jedoch vollständig mit dem Wort `stepcc` offen; dieses Wort ist der Dreh- und Angelpunkt der Consize-VM und ist zentral für das Verständnis des Interpreters.

Die Consize-VM ist in diesem Kapitel positiv spezifiziert. Damit ist gemeint: In der Spezifikation steht das Verhalten eines Wortes beschrieben, wenn man es unter den genannten Voraussetzungen verwendet. Hält man sich nicht an die Vereinbarungen, dann ist Consize frei gestellt, wie es mit der Verletzung der Vereinbarung umgehen möchte. In der Regel versucht Consize, die Interaktion über die Konsole aufrecht zu halten und auf einen Fehler hinzuweisen; mehr dazu in Kap. 3.15.

Die Spezifikation gibt pro Wort in der Regel ein Beispiel der Verwendung des Wortes an. Die Beispiele setzten die geladene Prelude voraus, die Consize automatisch beim Programmstart einliest.

### 3.1 Kurzdokumentation mittels Stapeleffekten

Zu den rund 50 Wörtern gibt es eine dokumentierende Kurzform, die Ihnen hilft sich zu erinnern, welche Auswirkungen, welche Effekte das Wort auf den Datastack hat. Man spricht vom Stapeleffekt (*stack effect*) eines Wortes. Der Stapeleffekt wird in runden Klammern notiert. Er gibt an, was das Wort auf dem Stack erwartet und von dort entfernt und – durch Doppelstriche „--“ getrennt angegeben – welche Situation das Wort nach Anwendung seiner mit ihm assoziierten Funktionalität auf dem Datastack hinterlässt.

Ein Beispiel: Das atomare Wort `dup` steht für *duplicate* und erzeugt ein Duplikat des obersten Werts auf dem Datastack. Der Stapeleffekt ist `( x -- x x )` und besagt: `dup` erwartet mindestens einen Wert auf dem Datastack, es entfernt ihn und bindet ihn gedanklich an `x` – das ist die Angabe links vom Doppelstrich. Sind mehrere Werte auf dem Datastack, so bleiben diese von `dup` unberührt. Die rechte Seite vom Doppelstrich sagt aus, dass `dup` zweimal das entfernte und gedanklich mit `x` assoziierte Element oben auf dem Datastack ablegt. Wenn nicht mindestens ein Element auf dem Datastack ist, kann `dup` seine Arbeit nicht verrichten.

```
> clear x y z dup
x y z z
```



Lassen Sie sich nicht von den Namen im Stapeleffekt verwirren. Sie sind gedankliche Stellvertreter für die tatsächlichen Elemente auf dem Datastack. In dem Beispiel ist **z** das oberste zu duplizierende Element auf dem Datastack!

Obwohl die Notation für Stapeleffekte informell ist, folgt sie gewissen Vereinbarungen. Meist sind die Namen nach einem der möglichen Datentypen benannt.

Konventionen:

- Wenn die Implementierung als Stapel nicht entscheidend ist, reden wir auch vom Stapel mit seinen Elementen als „Folge“ oder „Sequenz“ (*sequence*); bei der Notation der Stapeleffekte wird dann oft das Kürzel **seq** verwendet.
- Repräsentiert ein Stapel ein Programm, dann sprechen wir von einer „Quotierung“ (*quotation*), abgekürzt in den Stapeleffekten als **quot**.
- Ein Mapping kürzen wir mit **map** ab, den Spezialfall eines Wörterbuchs (*dictionary*) mit **dict**.
- Handelt es sich um ein beliebiges Element (*item*), dann kürzen wir es in den Stapeleffekten als **itm** ab.

Nicht immer passen diese Konventionen, manchmal erweisen sich andere Namen als hilfreicher, die Funktion eines Wortes zu erfassen.

### 3.2 Stack Shuffler: dup, swap, drop, rot

Das Tauschen, Verschieben, Duplizieren und Entfernen von Elementen auf dem Datastack wird als *stack shuffling* bezeichnet. Ganze vier Wörter reichen aus, um jedes gewünschte Arrangement der obersten drei Elemente auf dem Datastack herzustellen.

**dup ( x -- x x )** : dupliziert das oberste Element auf dem Datastack.

```
> clear x y z dup
x y z z
```

**swap ( x y -- y x )** : vertauscht die obersten Elemente auf dem Datastack.

```
> clear x y z swap
x z y
```

**drop ( x -- )** : entfernt das oberste Element auf dem Datastack.

```
> clear x y z drop
x y
```

**rot ( x y z -- y z x )** : rotiert die obersten drei Elemente auf dem Datastack, wobei das „unterste“ Element nach oben gebracht wird.

```
> clear x y z rot
y z x
```

### 3.3 Typ und Vergleich: type, equal?, identical?

Consize kennt insgesamt fünf Datentypen. Ein Datentyp steht für eine Menge von Datenwerten. So steht der Datentyp **word** für die Menge aller Wörter, **fact** für die Menge aller Funktionen, **stack** für die Menge aller Stapel und **mapping** für die Menge aller Mappings. Eine Sonderstellung nimmt der Typ **nil** ein. Er repräsentiert einen einzigen Wert, der „Nichts“ heißt

(englisch *nil*) und dann eingesetzt wird, wenn statt eines Fehlers das Resultat einer Operation als erfolglos ausgewiesen werden soll im Sinne von „es ist ‘nichts’ dabei herumgekommen“. Mehr dazu in Kap. 4.

`type ( itm -- wrd )`: ermittelt den Datentypen des obersten Elements auf dem Datastack. Der Typ ist entweder das Wort `wrd`, `stk`, `map`, `fct` oder `nil`.

```
> clear hi type
wrd
> [ 1 2 3 ] type
wrd stk
```

Sie haben ein intuitives Verständnis davon, was Gleichheit (*equality*) von Werten bedeutet. Dass die Stapel `[ 1 x 2 ]` und `[ 1 x 2 ]` gleich sind, leuchtet Ihnen unmittelbar ein. Wir werden in Kap. 4 genauer definieren, was mit Gleichheit gemeint ist. In dem Kap. 4 wird auch das Konzept der Identität (*identity*) ausführlich zur Sprache kommen.

`equal? ( itm1 itm2 -- t/f )`: testet die Gleichheit der obersten zwei Elemente vom Datastack; liefert entweder `t` (für *true*, wahr) oder `f` (für *false*, falsch) zurück.

```
> clear [ 1 2 3 ] [ 1 2 3 ] equal?
t
```

Das Wort `identical?` ist einzig aus didaktischen Gründen in Consize vorhanden, siehe Kap. 4. Es hat in einer funktionalen Sprache keine Verwendung und ist in einer Consize-Implementierung nicht erforderlich.

`identical? ( itm1 itm2 -- t/f )`: testet die obersten zwei Elemente auf dem Datastack auf Identität. Legt `t` oder `f` auf dem Datastack ab.

Wörter, die als Rückgabe auf dem Datastack entweder ein `t` (für *true*) oder `f` (für *false*) zurücklassen, schließen ihren Namen gerne mit einem Fragezeichen ab. In vielen Fällen ist diese Konvention hilfreich und dient als Gedächtnisstütze: Endet ein Wort mit einem Fragezeichen, ist das Resultat auf dem Stack entweder `t` oder `f`.

### 3.4 Stapel: `emptystack`, `push`, `top`, `pop`, `reverse`, `concat`

Der Stapel ist *die* allgegenwärtige Datenstruktur in Consize.

`emptystack ( -- [ ] )`: legt auf dem Datastack einen leeren Stapel ab.

```
> clear emptystack
[ ]
```

`push ( stk itm -- stk' )`: erzeugt einen neuen Stapel `stk'`, der das Ergebnis des Vorgangs ist, das Element `itm` auf dem Stapel `stk` abzulegen.

```
> clear [ 1 2 3 ] 4 push
[ 4 1 2 3 ]
```

`top ( stk -- itm )`: legt das oberste Element `itm` vom Stapel `stk` auf dem Datastack ab. Bei einem leeren Stapel oder `nil` liefert `top` als Ergebnis `nil`.

```
> clear [ 4 1 2 3 ] top
4
> [ ] top
4 nil
> nil top
4 nil nil
```

**pop ( stk -- stk' )** : legt den um das oberste Element reduzierten Stapel auf dem Datastack ab.

```
> clear [ 1 2 3 ] pop
[ 2 3 ]
> [ ] pop
[ 2 3 ] [ ]
> nil pop
[ 2 3 ] [ ] [ ]
```

Der Wert für „Nichts“ (*nil*), der von **top** erzeugt und auch von **pop** akzeptiert wird, hat eine Sonderfunktion. Er weist das Resultat von **[ ] top** als erfolglos aus; es macht keinen Sinn, von einem leeren Stapel den obersten Wert zu verlangen. Mit *nil* als Wert dennoch bei **top** und **pop** weiterarbeiten zu können, ermöglicht die effiziente Zerlegung von Stapeln, ohne stets überprüfen zu müssen, ob der Stapel inzwischen leer ist. Abseits dieses Einsatzzwecks sollten Sie auf *nil* als Datenwert verzichten.

**reverse ( stk -- stk' )** : kehrt die Reihenfolge der Element in einem Stapel um.

```
> clear [ 1 2 3 4 5 ] reverse
[ 5 4 3 2 1 ]
```

**concat ( stk1 stk2 -- stk3 )** : verbindet die Elemente der beiden Stapel **stk1** und **stk2** zu einem neuen Stapel **stk3**. Die Reihenfolge der Elemente wird gemäß der Lesart von links nach rechts beibehalten. Ein leerer Stapel konkateniert lediglich seinen „leeren“ Inhalt.

```
> clear [ 1 2 3 ] [ 4 5 ] concat
[ 1 2 3 4 5 ]
> [ ] concat
[ 1 2 3 4 5 ]
```

Hinweis: Die Wörter **reverse** und **concat** bietet die Consize-VM aus Gründen der Performanz an. Sie können als optional betrachtet werden. Man könnte diese Wörter auch in der Prelude definieren.

### 3.5 Mappings: mapping, unmap, assoc, dissoc, get, keys, merge

**mapping ( stk -- map )** : wandelt einen Stapel in ein Mapping um. Die Elemente auf dem Stapel werden paarweise als Schlüssel- und Zielwert interpretiert. Der Stapel muss eine gerade Anzahl an Elementen haben. Ein leerer Stapel liefert ein leeres Mapping.

```
> clear [ mon 1 tue 2 wed 3 thu 4 fri 5 ] mapping
{ thu 4 tue 2 mon 1 wed 3 fri 5 }
> clear [ ] mapping
{ }
```

**unmap ( map -- stk )** : wandelt ein Mapping in einen Stack, wobei die Assoziationen eine Folge von Schlüssel- und Zielwerten bilden. Die Reihenfolge von Schlüssel-/Zielwertpaaren ist unbestimmt. Ein leeres Mapping führt zu einem leeren Stack.

```
> clear { mon 1 tue 2 wed 3 thu 4 fri 5 } unmap
[ thu 4 tue 2 mon 1 wed 3 fri 5 ]
> clear { } unmap
[ ]
```

`assoc ( val key map -- map' )` : fügt der Map `map` die Assoziation aus Schlüsselwert `key` und Zielwert `val` hinzu und liefert die neue Map `map'` zurück. Existiert ein Schlüsselwert in `map` bereits, wird der Zielwert „überschrieben“.

```
> clear 3 :radius { :type circle } assoc
{ :radius 3 :type circle }
```

`dissoc ( key map -- map' )` : legt ein Mapping `map'` auf dem Datastack ab, das alle Assoziationen des Mappings `map` übernimmt bis auf die Assoziation, die über den Schlüsselwert `key` identifiziert ist. Existiert `key` in `map` nicht, bleibt das Mapping unverändert.

```
> clear c { a b c d } dissoc
{ a b }
> clear c { a b } dissoc
{ a b }
```

`get ( key map default -- val/default )` : liefert den mit dem Schlüsselwert `key` im Mapping `map` assoziierten Zielwert `val` zurück. Existiert die Assoziation nicht, liefert `get` stattdessen den `default`-Wert zurück.

```
> clear thu { mon 1 tue 2 wed 3 thu 4 fri 5 } _|_ get
4
> clear sat { mon 1 tue 2 wed 3 thu 4 fri 5 } _|_ get
_|_
```

`keys ( map -- seq )` : gibt alle Schlüsselwerte eines Mappings `map` als Sequenz `seq` zurück. Die Reihenfolge der Schlüsselwerte in `seq` kann beliebig sein.

```
> clear { mon 1 tue 2 wed 3 thu 4 fri 5 } keys
[ thu tue mon wed fri ]
```

`merge ( map1 map2 -- map3 )` : fasst die zwei Mappings `map1` und `map2` zu einem neuen Mapping `map3` zusammen. Bei gleichen Schlüsselwerten in `map1` und `map2` werden die Assoziationen aus `map2` in `map3` übernommen.

```
> clear { a b c d } { c x y z } merge
{ a b c x y z }
```

Hinweis: Nicht alle Wörter zu Mappings sind unabdingbar notwendig. Die Wörter `unmap`, `dissoc` und `merge` bietet die Consize-VM aus Gründen der Performanz an. Sie können als optional betrachtet werden, da sie mit den übrigen Wörtern der Consize-VM nachgebildet werden können.

### 3.6 Wörter: `unword`, `word`, `char`

Mit `unword` und `word` kann man Wörter in ihre Einzelzeichen zerlegen und mehrere Wörter zu neuen Wörtern zusammen setzen. Nützlich ist das für die Arbeit mit Ein- und Ausgaben (die mit Wörtern arbeiten) und das automatische Generieren von Wörtern.

`unword ( wrd -- seq )` : zerlegt ein Wort in seine Einzelzeichen (Symbole) in Form einer Folge von Wörtern. Jedes Wort in der Folge entspricht einem Einzelzeichen.

```
> clear \ push unword
[ p u s h ]
```

`word ( seq -- wrd )` : erwartet eine Folge von ausschließlich Wörtern und fügt diese Wörter zu einem neuen Gesamtwort zusammen. Die Folge muss mindestens ein Wort enthalten.

```
> clear [ it's me ! ] word
it'sme!
```

`char ( wrd -- wrd' )`: legt auf dem Datastack ein Wort `wrd'` ab, das lediglich aus einem Einzelzeichen (*character*) besteht. Das Einzelzeichen wird durch das Wort `wrd` kodiert. Die Kodierung erfolgt als [Unicode](#) mit dem Präfix „\u“ und vier folgenden Stellen im [Hexadezimalsystem](#) (z.B. \u0040 für das Wort `@`) oder im [Oktalsystem](#) mit dem Präfix „\o“ (z.B. \o100 ebenfalls für `@`). Zusätzlich stehen als Kodierung für `wrd` die Wörter

- `\space` (Leerzeichen, *space*),
- `\newline` (Zeilenvorschub, *line feed*),
- `\formfeed` (Seitenvorschub *form feed*),
- `\return` (Wagenrücklauf, *carriage return*),
- `\backspace` (Rückschritt, *backspace*) und
- `\tab` (horizontaler Tabulator, *tab*)

zur Verfügung, denen die entsprechenden Zeichen bzw. [Steuerzeichen](#) als Wörter für `wrd'` entsprechen. Beachten Sie, dass `\space` etc. mit `char` „unsichtbare“ Wörter erzeugen, die z.B. bei der Ausgabe über die Konsole dennoch Auswirkungen als [Leerraum](#) haben.

```
> clear \u0040 char
@
```

### 3.7 Konsole: print, flush, read-line

Zur Ein- und Ausgabe über die Konsole stellt Consize drei Wörter bereit. In der Regel erfolgen Ausgaben über die Konsole nicht direkt, sondern über den Umweg eines Puffers.

`print ( wrd -- )`: gibt das Wort auf der Konsole aus. Da die Ausgabe über einen [Puffer](#) erfolgt, kann die Ausgabe möglicherweise nicht direkt sondern zu einem späteren Zeitpunkt erfolgen. Die sofortige Ausgabe erzwingt `flush`.

```
> clear \ Hello print \newline char print
Hello
```

`flush ( -- )`: leert den Ausgabepuffer und bringt alles, was noch im Ausgabepuffer ist, zur Ausgabe in der Konsole. Das Wort „*flush*“ heißt soviel wie „ausspülen“.

```
> clear \ Hi print \newline char print flush
Hi
```

`read-line ( -- wrd )`: liest eine Zeile über die Konsole ein. Sobald die Eingabe mit der [Eingabetaste](#) abgeschlossen ist, legt `read-line` die Eingabezeile als Wort auf dem Datastack ab.

Geben Sie in dem nachstehenden Beispiel nach `read-line` über die Tastatur „Hello you“ ein und schließen Sie die Eingabe mit der Eingabetaste ab.

```
> clear read-line
Hello you
Hello you
```

Die Ausgabe lässt den Eindruck aufkommen, als handele es sich um zwei Wörter auf dem Datastack. Tatsächlich ist `Hello you` ein Wort, das ein Leerzeichen beinhaltet! Das wird offensichtlich, wenn Sie ein `dup` eingeben.

```
> dup
Hello you Hello you
```

### 3.8 Dateien und mehr: slurp, spit, spit-on

Consize unterstützt nur sehr rudimentär die Arbeit mit dem [Dateisystem](#): das Lesen von Dateien mit `slurp` und das Schreiben von Daten in Dateien mit `spit` und `spit-on`. Consize ist absichtlich nicht mit weiteren Fähigkeiten zum Löschen von Dateien und zum Navigieren durch das Dateisystem ausgestattet.

`slurp ( wrd -- wrd' )`: interpretiert das Wort `wrd` als Datenquelle, liest die Daten von dort ein und legt die Daten als Wort `wrd'` auf dem Datastack ab. Nur Datenquellen mit Textdaten können sinnvoll verarbeitet werden. Bei einfachen Wortnamen für `wrd` liest Consize die Daten von einer namensgleichen Datei ein, die sich im Aufrufverzeichnis von Consize befindet. Die Konventionen für Dateipfade folgen [java.io](#). So können auch Daten aus dem World Wide Web gelesen werden, siehe Beispiel.

Das folgende Beispiel zeigt, dass Sie auch Daten über das Hypertext Transfer Protokoll (HTTP) lesen können, sofern Sie online sind. Das gelieferte Ergebnis ist der Inhalt eines einzigen Wortes. Auch hier gilt wie bei `read-line`: Das Wort `wrd'` kann Leerzeichen, Sonder- und Steuerzeichen enthalten. Die beispielhafte Ausgabe ist natürlich abhängig vom aktuellen Inhalt der Webseite. Der besseren Lesbarkeit wegen habe ich Zeilenumbrüche hinzugefügt

```
> clear http://m.twitter.com slurp
<html><body>You are being
<a href="https://mobile.twitter.com/signup">redirected</a>
.</body></html>
```

`spit ( data-wrd file-wrd -- )`: schreibt das Wort `data-wrd` in eine Datei unter dem Namen `file-wrd` in das Dateisystem. Für `file-wrd` gelten die Konventionen wie unter `slurp` erwähnt. Existiert die Datei nicht, wird sie neu angelegt. Existiert die Datei bereits, wird ihr bisheriger Inhalt überschrieben.

Nach Eingabe des Beispiels sollten Sie im Startverzeichnis von Consize eine Datei namens `dummy.txt` finden. Öffnen Sie die Datei mit einem Editor, um sich den Inhalt anzuschauen.

```
> clear \ Hello \ dummy.txt spit
```

`spit-on ( data-wrd file-wrd -- )`: hängt das Wort `data-wrd` an den Inhalt der Datei unter dem Namen `file-wrd` an. Für `file-wrd` gelten die Konventionen wie unter `slurp` erwähnt.

Wenn Sie das Beispiel zu `spit` ausgeführt haben, hängt Ihnen das folgende Beispiel ein `You` an. In der Datei steht anschließend `HelloYou`.

```
> clear \ You \ dummy.txt spit-on
```

### 3.9 Parsing: uncomment, tokenize, undocument

Die Wörter `uncomment`, `undocument` und `tokenize` bearbeiten den durch ein Wort repräsentierten Inhalt seiner Einzelzeichen. Die Wörter sind insbesondere dafür gedacht, über `read-line` oder `slurp` eingelesene Consize-Programme einer Vorverarbeitung unterziehen zu können.

`uncomment ( wrd -- wrd' )` : entfernt aus einem Wort alle Kommentare. Ein Kommentar beginnt mit dem `%`-Zeichen und geht bis zum Ende einer Zeile. Das Zeilenende wird durch einen [Zeilenumbruch](#) markiert. Je nach Betriebssystem markieren die [Steuerzeichen](#) „[Wagenrücklauf](#)“ (*carriage return*, CR) und/oder „[Zeilenvorschub](#)“ (*line feed*, LF) den Zeilenumbruch.

Da `read-line` im folgenden Beispiel keinen Marker für das Zeilenende setzt, müssen wir ein `\newline char` zu dem eingelesenen Wort hinzufügen, damit `uncomment` den Kommentar entfernen kann.

```
> read-line
This line % has a comment
This line % has a comment
> [ ] \newline char push swap push word uncomment
This line
```

`tokenize ( wrd -- seq )` : zerlegt das Wort an vorhandenen Leerraum-Stellen in eine Folge von Wörtern gemäß dem [regulären Ausdruck](#) „`\s*%.*[(\r\n)\r\n]`“. Als [Leerraum](#) (*whitespace character*, „`s`“ im regulären Ausdruck) gilt eine nicht leere Folge bzw. Kombination der folgenden Zeichen: Leerzeichen, horizontaler Tabulator, vertikaler Tabulator, Zeilenvorschub, Seitenvorschub und Wagenrücklauf. (Im regulären Ausdruck steht ein „`\n`“ für einen Zeilenvorschub und ein „`\r`“ für einen Wagenrücklauf.) Diese Definition eines Leerraums folgt dem [POSIX](#)-Standard für reguläre Ausdrücke.

```
> read-line tokenize
This line % has a comment
[ This line % has a comment ]
```

`undocument ( wrd -- wrd' )` : extrahiert aus einem Wort lediglich die „Zeilen“, die mit einem „`>>` “ bzw. „`%>` “ (jeweils mit einem Leerzeichen) beginnen, verwirft dabei jedoch diese Anfänge. Die Extraktion folgt dem regulären Ausdruck „`[(\r\n)\r\n]?>> (.*)[(\r\n)\r\n]`“. Die extrahierten Anteile werden per Wagenrücklauf und Zeilenvorschub miteinander verknüpft und als neues Wort `wrd'` zurück gegeben.

Der Nutzen von `undocument` zusammen mit `slurp` liegt darin, aus einem Textfile lediglich die mit „`>>` “ bzw. „`%>` “ beginnenden Textzeilen herauszufiltern und nur noch diese Zeilen (ohne die Präfixe) als Programmzeilen für Consize zu betrachten. Das erlaubt eine schwache Form des [Literate Programming](#), wie sie auch die funktionale Programmiersprache [Haskell](#) unterstützt. Das Literate Programming zielt darauf ab, nicht den Code zu dokumentieren, sondern die Dokumentation mit Code anzureichern. Das Wort `uncomment` entfernt Kommentare aus Consize-Code,

das Wort `undocument` entfernt die Dokumentation aus dokumentiertem Consize-Code. In beiden Fällen bleibt der reine Consize-Code übrig.

Hinweise: Alle Parsing-Wörter sind nicht strikt erforderlich. Sie können allesamt durch Programme aus den übrigen Wörtern der Consize-VM nachgebildet werden und sind deshalb als optional zu betrachten. Aus Gründen der Performanz sind sie jedoch als fester Bestandteil der Consize-VM empfohlen.

### 3.10 Funktionen: `func`, `apply`, `compose`

Die Bedeutung atomarer Wörter ist über fest vorgegebene Funktionen definiert (*primitive Funktionen*). Darüber hinaus kann man neue Funktionen erstellen, Funktionen miteinander komponieren und auf einen Stapel anwenden.

`func ( quot dict -- fct )` : erzeugt eine Funktion und initialisiert sie mit einer Quotierung `quot` als Programm und einem im Kontext der Funktion gültigen Wörterbuch `dict`. Die Funktion kann mit `apply` aufgerufen werden. Die Semantik der mit `apply` aufgerufenen Funktion ist die der wiederholten Anwendung von `stepcc` auf die Continuation, die durch `dict`, den über `apply` übergebenen Stapel als Datastack und `quot` als Callstack gegeben ist, und zwar solange, bis der Callstack (`quot`) leer ist. Das Ergebnis der Funktionsanwendung mit `apply` ist dann der Datastack der resultierenden vollständigen Continuation.

```
> clear [ rot swap ] get-dict func
<fct>
```

`apply ( stk fct -- stk' )` : wendet die Funktion `fct` auf den Stapel `stk` an. Mit anderen Worten: Der Stapel `stk` ist der Datastack, mit dem die Funktion `fct` aufgerufen wird. Das Ergebnis `stk'` ist der resultierende Datastack aus der Funktionsanwendung.

Beachten Sie: Der Datastack, der Ihnen bei der Interaktion mit der Konsole angezeigt wird, hat sein offenes Stapelende rechts. Ein Stapel auf dem Datastack hat sein offenes Stapelende links.

```
> clear [ x y z ] [ rot swap ] get-dict func apply
[ x z y ]
```

`compose ( fct1 fct2 -- fct3 )` : liefert Funktion `fct3` als Ergebnis der Funktionskomposition von `fct1` und `fct2` zurück.

```
> clear \ rot get-dict nil get \ swap get-dict nil get compose
<fct>
> [ x y z ] swap apply
[ x z y ]
```

### 3.11 Continuations: `stepcc`, `call/cc`, `continue`, `get-dict`, `set-dict`, `\`

Ein Mapping und zwei Stapel heißen vollständige Continuation, sofern das Mapping als Wörterbuch, der eine Stapel als Datastack und der andere Stapel als Callstack interpretiert werden. Eine vollständige [Continuation](#) repräsentiert über das Wörterbuch und den Datastack nicht nur den aktuellen Zustand eines Rechenprozesses, sondern über den Callstack auch das zu bearbeitende Programm, sozusagen die gesamte rechnerische „Zukunft“. Als (verkürzte) Continuation bezeichnen wir die Kombination von Data- und Callstack ohne Wörterbuch.



Das Wort `stepcc` definiert einen einzelnen Rechenschritt einer vollständigen Continuation. Das Wort `stepcc` erwartet also die drei Anteile einer Continuation auf dem Datastack und liefert auch wieder die drei Anteile einer Continuation zurück. Über `stepcc` sind darüber hinaus vier weitere Wörter definiert, die den expliziten Umgang mit einer verkürzten Continuation (`call/cc` und `continue`) und dem Wörterbuch (`get-dict` und `set-dict`) innerhalb eines „normalen“ Programms erlauben.

`stepcc ( dict ds cs -- dict' ds' cs' )`: erwartet auf dem Datastack ein Wörterbuch `dict` und zwei Stapel `ds` und `cs`. Die drei Datenwerte werden als vollständige Continuation aufgefasst. `stepcc` definiert einen Rechenschritt in Consize und ermittelt die resultierende vollständige Continuation `dict'`, `ds'` und `cs'`. Zum leichteren Verständnis vereinbaren wir folgende Substitutionen (Ersetzungen):<sup>1</sup>

- `itm = cs top` – ist das erste Element auf `cs`
- `rsc = cs pop` – ist der Rest von `cs`
- `head = ds top` – ist das erste Element auf `ds`
- `tail = ds pop` – ist der Rest von `ds`
- `res = itm dict nil get` ist der mit `itm` assoziierte Wert in `dict`

Die resultierende Continuation wird ebenfalls durch Substitutionen für `dict'`, `ds'` bzw. `cs'` angegeben. Sofern in den Fallunterscheidungen nichts anderes angegeben ist, gilt grundsätzlich `dict' = dict`, `cs' = rsc`, `ds' = ds`.

Wort auf `cs` führt zu ...

... Konkatenation

... Funktionsanwendung

... Spezialwortvergleich

→ `call/cc`

→ `continue`

→ `get-dict`

→ `set-dict`

... `read-word`

Mapping auf `cs` führt zu ...

... `read-mapping`

1. Ist `itm` ein Wort und

- (a) `res` ein Stack: `cs' = res rsc concat`
- (b) `res` eine Funktion: `ds' = ds res apply`
- (c) weder/noch: Ist `itm` gleich dem Wort
  - i. `call/cc`:  
`ds' = [ ] tail push rsc push,`  
`cs' = head`
  - ii. `continue`:  
`ds' = tail top,`  
`cs' = head`
  - iii. `get-dict`:  
`ds' = ds dict push`
  - iv. `set-dict`:  
`dict' = head,`  
`ds' = tail`
  - v. ansonsten:  
`ds' = ds itm push,`  
`cs' = rsc \ read-word push`

2. Ist `itm` ein Mapping:

`ds' = ds itm push,`

<sup>1</sup> Bitte verwechseln Sie das Gleichheitszeichen auf keinen Fall mit einer Zuweisung. Wann immer die linke Seite vom Gleichheitszeichen erscheint, wie z.B. `itm`, ist sie „gedanklich“ zu ersetzen mit der rechten Seite vom Gleichheitszeichen; bei `itm` mit `cs top`.

Sonstiges Element auf `cs` ...  
... wandert auf `ds`

3. Ist `itm` weder ein Wort noch ein Mapping:  
`ds' = ds itm push`

In dieser formalen Darstellung stellt sich die Spezifikation des Verhaltens von `stepcc` ein wenig komplizierter dar, als sie es wirklich ist. Freisprachlich formuliert arbeitet `stepcc` wie folgt:

Befindet sich auf `cs` ein Wort und findet sich zu diesem Wort im Wörterbuch `dict` eine Quotierung, dann „ersetzt“ der Inhalt der Quotierung das Wort. Die Ersetzung geschieht durch Konkatenation der Quotierung `res` und dem Rest von `cs`, `rscs`.

```
> clear { \ -rot [ rot rot ] } [ z y x ] [ -rot swap ] stepcc  
{ -rot [ rot rot ] } [ z y x ] [ rot rot swap ]
```

Findet sich im Wörterbuch stattdessen eine Funktion, dann wird die Funktion auf `ds` angewendet. Mit `get-dict` wird das aktuell gültige Wörterbuch auf dem Datastack abgelegt.

```
> clear { \ rot dup get-dict nil get }  
{ rot <fct> }  
> [ z y x ] [ rot swap ] stepcc  
{ rot <fct> } [ x z y ] [ swap ]
```

Ist das Wort auf dem `cs` weder mit einer Quotierung noch mit einer Funktion assoziiert, überprüft `stepcc`, ob das Wort `call/cc`, `continue`, `get-dict` oder `set-dict` ist – die Effekte dieser Wörter sind in eigenen Abschnitten noch einmal beschrieben. Trifft keiner dieser Vergleiche zu, dann ist das Wort unbekannt und `stepcc` entfernt das Wort von `cs` und legt es auf `ds` ab. Das Wort `read-word` wird auf dem verbliebenen `cs`, `rscs`, abgelegt.

```
> clear { } [ z y x ] [ rot swap ] stepcc  
{ } [ rot z y x ] [ read-word swap ]
```

Auf diese Weise kann mittels `read-word` ein Verhalten für unbekannte Wörter vom Anwender selbst definiert werden. Anders ausgedrückt: `read-word` ist ein Meta-Wort zur Definition des Umgangs mit unbekannten Wörtern. Ein Meta-Wort verweist auf die Existenz einer [Metaebene](#), die es erlaubt, das Verhalten von Elementen auf der „Sachebene“ zu beschreiben.

Ähnlich ist der Umgang mit Mappings. Ist das oberste Element auf `cs` kein Wort, sondern ein Mapping, dann entfernt `stepcc` das Mapping von `cs`, legt es auf `ds` ab und legt das Wort `read-mapping` auf dem Rest von `cs`, `rscs`, ab.

```
> clear { } [ z y x ] [ swap ] { a b } push stepcc  
{ } [ { a b } z y x ] [ read-mapping swap ]
```

Das Wort `read-mapping` ist ein Meta-Wort, mit dem der Anwender das Verhalten im Umgang mit Mappings definieren kann.

Ist das oberste Element auf `cs` weder ein Wort noch ein Mapping, dann legt `stepcc` es auf dem `ds` ab.

```
> clear { } [ z y x ] [ [ 1 2 3 ] swap ] stepcc  
{ } [ [ 1 2 3 ] z y x ] [ swap ]
```

! → Die Bedeutung von **stepcc** ist, dass es gemeinsam mit **func** den „Antriebsmotor“ von Consize definiert. Eine mit **func** erzeugte und mit **apply** auf einen Stapel angewendete Funktion ist in ihrer Ausführung zu verstehen als fortwährende Anwendung von **stepcc** auf die via **func** und **apply** gegebene vollständige Continuation – und zwar solange, bis der Callstack der vollständigen Continuation abgearbeitet, sprich leer ist. Das Ergebnis der Abarbeitung ist der Datastack.

**call/cc ( quot -- ds' cs' ) :** legt den Datastack (ohne **quot**) und den Callstack (ohne **call/cc**) der aktuellen Continuation auf dem Datastack ab und ersetzt anschließend den aktuellen Callstack durch die Quotierung **quot**. Da **call/cc** ohne **continue** kaum vernünftig einzusetzen ist, ist hier der Stackeffekt in Anlehnung an die Definition unter **stepcc** angegeben mit **ds' cs'**. Dieser Stapeleffekt präsentiert die Regel, erfasst aber nicht die Ausnahme von der Regel.

→ **stepcc**

„**call/cc**“ steht übrigens für „*call with current continuation*“: rufe die Quotierung als Programm auf mit der aktuellen (verkürzten) Continuation auf dem Datastack.

Das Beispiel verwendet das Wort **printer** aus der Prelude; **printer** gibt den aktuellen Stand des Datastacks auf der Konsole aus. Zu **continue** siehe die Spezifikation des Wortes.

```
> clear x y z [ printer continue ] call/cc rot
[ z y x ] [ rot printer repl ]
y z x
```

Wenn das Wort **call/cc** durch **stepcc** in der Bearbeitung ist, dann sieht die Situation auf dem Data- und Callstack wie folgt aus:

```
x y z [ printer continue ] | call/cc rot printer repl
```

Dass sich hinter dem **rot** ein **printer** und **repl** anschließen, ist in der Prelude programmiert; so wird der interaktive Modus über die Konsole realisiert. Das Wort **call/cc** verwandelt die Situation auf Data- und Callstack folgendermaßen:

```
[ z y x ] [ rot printer repl ] | printer continue
```

Es wird das Wort **printer** ausgeführt, das den Datastack auf der Konsole ausgibt, dann das Wort **continue**, das die verkürzte Continuation auf dem Datastack wieder aktiviert und die Programmausführung fortsetzt.

**continue ( ds cs -- ds | cs ) :** lässt die Stapel **ds** und **cs** als Datastack und Callstack die Programmausführung übernehmen. Das Wort **continue** ist im Zusammenhang mit **stepcc** als Spezialwort definiert. Der Stapeleffekt auf der rechten Seite soll andeuten, dass sich die Situation auf dem Datastack ergibt aus der Abarbeitung der Continuation **ds cs**.

→ **stepcc**

Mit **continue** können zwei beliebige Stapel als Datenstack und Callstack interpretiert die Programmausführung übernehmen.

```
> clear [ z y x ] [ rot printer repl ] continue
y z x
```

**get-dict ( -- dict ) :** hinterlässt auf dem Datastack das Wörterbuch der aktuellen, in der Abarbeitung befindlichen Continuation. Das Wort **get-dict** ist im Zusammenhang mit **stepcc** als Spezialwort definiert.

→ **stepcc**

Wenn Sie `get-dict` an der Konsole eingeben dauert die Aufbereitung der Ausgabe eine Weile. Haben Sie ein wenig Geduld.

```
> clear get-dict
% output deliberately omitted
```

`set-dict ( dict -- )`: nimmt das Mapping `dict` vom Datastack und setzt das Wörterbuch der aktuellen Continuation gleich `dict`. Das Wort `set-dict` ist im Zusammenhang mit `stepcc` als Spezialwort definiert.

→ `stepcc`

Das Beispiel fügt ein neues Wort `2pop` mit der Bedeutung `pop pop` zum aktuellen Wörterbuch hinzu.

```
> clear [ pop pop ] \ 2pop get-dict assoc set-dict
```

```
> [ x y z ] 2pop
[ z ]
```

`\ ( -- itm )`: holt das dem Wort `\` folgende Element auf dem Callstack vom Callstack und legt es auf dem Datastack ab. Das Wort `\` wird auch als „Quote“ oder „Escape“ bezeichnet. Es verhindert die Interpretation von Daten auf dem Callstack. Das Wort ist definiert über das folgende Programm:

```
[ dup top rot swap push swap pop continue ] call/cc
```

Sie können die Definition von `\` direkt überprüfen, indem Sie es im Wörterbuch nachschlagen.

```
> \ \ get-dict nil get
[ [ dup top rot swap push swap pop continue ] call/cc ]
```

Ohne ein Quote-Wort ist es nicht möglich, ein Wort wie z.B. `rot` als Datum zu behandeln. Es würde sonst immer ausgeführt werden.

```
> clear \ rot
rot
```

### 3.12 Arithmetik: +, -, \*, div, mod, <, >, integer?

Consize bietet ein paar Wörter an, die Wörter als Zahlen interpretieren und damit das Rechnen mit Zahlen ([Arithmetik](#)) sowie einfache Zahlenvergleiche (größer, kleiner) ermöglichen. Nachfolgend sind die Angaben `x`, `y` und `z` in den Stapeleffekten Wörter, die [ganze Zahlen](#) repräsentieren. Allerdings sind Ganzzahlen nicht als eigenständiger Datentyp in Consize vertreten, Zahlen sind Wörter.

`+ ( x y -- z )`: liefert mit `z` die Summe `x+y` zurück; `+` realisiert die [Addition](#).

```
> 2 3 +
5
```

`- ( x y -- z )`: liefert mit `z` die Differenz `x-y` zurück; `-` realisiert die [Subtraktion](#).

```
> 2 3 -
-1
```

`* ( x y -- z )`: liefert mit `z` das Produkt `x*y` zurück; `*` realisiert die [Multiplikation](#).

```
> 2 3 *
6
```

`div ( x y -- z )` : liefert mit `z` den ganzzahligen Wert des Quotienten `x:y` zurück; `div` realisiert die ganzzahlige [Division](#). Der Divisor `y` muss von 0 verschieden sein.

```
> 7 3 div
2
```

`mod ( x y -- z )` : liefert mit `z` den Rest der Division `x:y` zurück; `mod` realisiert den Divisionsrest, „[Modulo](#)“ genannt. Die Zahl `y` muss von 0 verschieden sein.

```
> 7 3 mod
1
```

Die Vergleichsoperatoren `<`, `>` etc. nutzen die [Ordnungsrelation](#) unter den Zahlen, die man sich z.B. anhand eines [Zahlenstrahls](#) veranschaulichen kann.

`< ( x y -- t/f )` : liefert als Ergebnis des Vergleichs `x<y` entweder `t` (für wahr, *true*) oder `f` (für falsch, *false*) zurück.

```
> 7 3 <
f
```

`> ( x y -- t/f )` : liefert als Ergebnis des Vergleichs `x>y` entweder `t` oder `f` zurück.

```
> 7 3 >
t
```

`integer? ( x -- t/f )` : testet, ob das Wort `x` eine Ganzzahl (*integer*) repräsentiert, und liefert als Ergebnis entweder `t` oder `f` zurück.

```
> -7 integer?
t
> x integer?
f
```

Hinweis: Alle Wörter zur Arithmetik sind optional und können durch Consize-Programme in der Prelude vollständig ersetzt werden. Sie sind lediglich aus Performanzgründen in der Consize-VM enthalten. Die Referenzimplementierung von Consize rechnet mit Ganzzahlen beliebiger Größe. Notwendig ist jedoch lediglich die Unterstützung von Ganzzahlen mit mindestens 16 Bit Genauigkeit ([Integer](#)), d.h. mit einem Wertebereich von  $-32.768$  bis  $+32.767$ . Für Berechnungen mit `current-time-millis` sind jedoch mindestens 32 Bit erforderlich.

### 3.13 Extras: `current-time-millis`, `operating-system`

Die zwei Wörter `current-time-millis` und `operating-system` erlauben einerseits z.B. die Messung der Laufzeit von Programmen oder die Initialisierung von Zufallsgeneratoren und andererseits z.B. den korrekten Umgang mit Zeilenumbrüchen je nach Betriebssystem.

`current-time-millis ( -- wrd )` : liefert als Wort die Anzahl der [Millisekunden](#) zurück, die seit dem 1. Januar 1970 ([UTC](#), Universal Time Coordinated), der so genannten [Unixzeit](#), verstrichen sind.

Natürlich hängt die beispielhafte Ausgabe von Ihrer aktuellen Zeit ab.

```
> clear current-time-millis
1347630655230
```

`operating-system ( -- wrd )`: gibt als Wort das verwendete [Betriebssystem](#) zurück, unter dem Consize läuft. Das Wort kann Leerzeichen im Namen enthalten.

Im Beispiel läuft Consize unter dem Betriebssystem Windows 7.

```
> clear operating-system
Windows 7
```

Hinweis: Die Wörter `current-time-millis` und `operating-system` sind optional. Man kann, wenn man möchte, Laufzeitmessungen von Programmen auch durch die Anzahl der Aufrufe von `stepcc` vornehmen.

### 3.14 Der Start der VM

Der Start der VM sei definiert anhand eines fiktiven Wortes (`main`). Der Start von Consize entspricht der Semantik von (`main`).

`(main) ( file -- | quot )`: versteht das auf dem Datastack liegende Wort `file` als Dateinamen (in der Grundeinstellung lautet das Wort `prelude.txt`), es lädt die Datei (`slurp`), entfernt Kommentare (`uncomment`) und zerlegt den Dateitext in einzelne Worte (`tokenize`). Die resultierende Folge von Wörtern ist die Quotierung `quot`.

Alle Wörter dieses Kapitels bilden zusammen mit den Bedeutungsdefinitionen das Ur-Wörterbuch `root-dict`. Consize definiert über `quot` und `root-dict` mittels `func` eine Funktion, die per `apply` auf einen leeren Stapel, der den Datastack repräsentiert, angewendet wird.

Das fiktive Wort (`main`) ist folglich definiert durch:

```
[ ] \ prelude.txt
slurp uncomment tokenize root-dict func apply
```

Der rechte Anteil des Stapeleffekts zeigt an, dass die Situation auf dem Datastack das Ergebnis einer Abarbeitung der Quotierung `quot` bei leerem Datastack ist.

! → In der von Consize gelesenen Datei, der Prelude, sind in der Regel die zwei Meta-Wörter `read-word` und `read-mapping` zu definieren. Ansonsten führt das Auftreten eines unbekannten Wortes zu `read-word`, was seinerseits ein unbekanntes Wort ist – und der Vorgang findet keine Ende. Ähnliches gilt bei einem nicht definierten `read-mapping`.

### 3.15 Referenzimplementierung

Für Consize liegt eine Referenzimplementierung in der funktionalen Sprache [Clojure](#) vor, die die Consize-VM komplett umsetzt. Das Clojure-Programm ist weniger als 150 Programmzeilen lang! Es läuft unter der [Java Virtual Machine](#) (JVM). Die Clojure-Implementierung ist im Zweifel der freisprachlichen Spezifikation vorzuziehen – allerdings ist dafür ein Verständnis der Sprache Clojure erforderlich.

Alternative Implementierungen der Consize-VM müssen das Verhalten der Referenzimplementierung nachbilden, wenn sie sich als „Consize-VM“ bezeichnen wollen; ausgenommen ist das Wort `identity?`. Als Basis der Verhaltensgleichheit können die Test-Suites für die Consize-VM und die Prelude herangezogen werden.

Optionale Wörter der Consize-VM müssen, sofern sie nicht Teil einer Consize-Implementierung sind, in einer angepassten Prelude bereitgestellt werden. Von dieser Regelung ausgenommen sind die Wörter aus Kap. 3.13.

Im Fall eines Fehlers versucht Consize die Interaktion aufrecht zu halten. Consize legt auf dem Callstack das Wort `_l_` (sprich „*bottom*“) ab und auf dem Datastack den Zustand der Continuation beim Auftreten des Problems. Eine Continuation besteht hier aus dem Abbild von Datastack und Callstack.

---

## 4 Von der Gleichheit und der Identität

---

Zwischen dem Konzept der Gleichheit und dem der Identität gibt es einen wichtigen Unterschied. Was gleich ist, muss nicht identisch sein. Aber was identisch ist, muss gleich sein. Zwei Zwillinge mögen sich bis aufs Haar gleichen, identisch (d.h. ein und dieselbe Person) sind sie deshalb nicht. Zwei Fotos ein und derselben Person zeigen nicht die gleiche Person (sofern das eine Bild keinen Doppelgänger darstellt), sondern dieselbe Person. Die gezeigte Person ist identisch.

Aber wie ist Gleichheit definiert? Und wie die Identität?

### 4.1 Gleich ist, was aus gleichen Werten bei gleicher Struktur besteht

Gleich ist, was gleich aussieht. Diese Definition ist einfach verständlich und intuitiv. Etwas formaler ausgedrückt: Zwei Datenwerte sind gleich, wenn sich ihre Repräsentationen auf dem Datastack nicht voneinander unterscheiden. Der Test auf Gleichheit ist ein syntaktischer Vergleich: Ist die Folge der die Daten repräsentierenden Zeichen gleich, dann sind auch die Daten gleich. Für Wörter trifft diese Definition von Gleichheit zu, für Stapel anscheinend auch – doch sie greift zu kurz.

```
> clear hello hello equal?  
t  
> [ a b c ] [ a b c ] equal?  
t t
```

Bei zusammengesetzte Datentypen spielen die strukturellen Bedingungen, die die Struktur organisieren, eine entscheidende Bedeutung bei der Frage, was gleich ist. Ganz deutlich wird das bei Mappings. Da die Anordnung der Assoziationspaare in einem Mapping unerheblich ist, müssen zwei Mappings nicht gleich aussehen, um gleich zu sein.

```
> { 1 2 3 4 } { 3 4 1 2 } equal?  
t
```

Wir müssen die Definition der Gleichheit erweitern: Zwei Datenstrukturen vom gleichen Typ (es macht keinen Sinn etwa Mappings mit Stapeln zu vergleichen) sind dann gleich, wenn ihre Strukturelemente unter Beachtung der strukturgebenden Eigenschaften gleich sind.

Das klingt kompliziert, ist es aber nicht. Zwei Wörter sind gleich, wenn sie gleich aussehen. Zwei Stapel sind gleich, wenn alle ihre Elemente in der selben Reihenfolge (von „oben“ bis „unten“) gleich sind. Zwei Mappings sind gleich, wenn sie die gleiche Menge an Schlüsselwerten haben und die Schlüsselwerte jeweils mit den gleichen Zielwerten assoziiert sind.

Die Definition ist rekursiv. Die Elemente eines Stapels oder eines Mappings sind jeweils auf Gleichheit zu testen, die ihrerseits Stapel oder Mappings sein können usw. Irgendwann muss die Untersuchung auf Gleichheit auf Wörter oder leere Stapel oder leere Mappings stoßen, damit die Gleichheit zweier Datenstrukturen entschieden werden kann.



## 4.2 Identität ist Verfolgbarkeit

Wenn Sie ein Element auf dem Datenstack „mit den Augen“ eindeutig verfolgen können, dann handelt es sich um ein und dasselbe, um das identische Element. Ein **swap** oder **rot** mag die Position eines Elements auf dem Datastack verändern, die Operation verändert aber nicht das Element und erst recht nicht seine Identität. Diese Operationen sind identitätserhaltend.

Doch der Sachverhalt des Identitätserhalts geht über ein einfaches Datengeschiebe auf dem Datastack hinaus. Es macht Sinn, auch die folgende Wortfolge als identitätserhaltend zu definieren: Wenn Sie ein Element auf einen Stack **pushen** und dann mit **top** von dort wieder erhalten, auch dann haben Sie das Element eindeutig verfolgen können. Wir können den Identitätserhalt ausdrücken über den folgenden Zusammenhang: ein **push top** hat dasselbe Ergebnis wie ein **swap drop**; letzteres gibt uns die Möglichkeit zu sagen, dass die Identität des Elements durch den Vorgang **push top** nicht verloren geht, da **swap** identitätserhaltend ist und **drop** den „lästigen“ Stapel einfach „vernichtet“. Sie können sich an der Konsole davon überzeugen, dass beide Programme ergebnisgleich sind.

```
> clear emptystack hi
[ ] hi
> push top
hi
> clear emptystack hi
[ ] hi
> swap drop
hi
```

Das Wort **identical?** dient zur Abfrage, ob die zwei obersten Elemente auf dem Datastack identisch sind. Sind sie es, so legt **identical?** ein **t** für *true* als Ergebnis auf dem Datastack ab, ansonsten **f** für *false*. Der Stackeffekt ist ( **itm1 itm2 -- t/f** ).

In unserem Fall heißt das: Das Wort **hi** *vor* Anwendung von **push top** muss mit dem Wort **hi** *nach* Anwendung von **push top** identisch sein. Wir müssen für die Erzeugung der Vorher/Nachher-Situation ein **dup** verwenden, um einen Vergleich überhaupt möglich zu machen. In Consize ausgedrückt heißt das: Sind **[ ] hi** auf dem Datenstack gegeben, dann *muss* das nachstehende Programm ein **t** ergeben. Und so ist es auch, wie Sie an der Konsole ausprobieren können – vergessen Sie nicht, zuvor ein **[ ] hi** auf den Datastack zu bringen.

```
> dup rot swap push top identical?
t
```

Dieses Ergebnis kann nur zustande kommen, wenn **dup** ein identitätserhaltendes Wort ist. Wir können das sogar beweisen: Da sich **push top** durch **swap drop** ersetzen lassen, ist das obige Programm ergebnisgleich mit:

```
> dup rot swap swap drop identical?
t
```

Da ein doppeltes **swap** den Tausch auf dem Stack rückgängig macht, kann das Programm verkürzt werden zu:

```
> dup rot drop identical?
t
```

Die Wörter `rot` `drop` „vernichten“ lediglich das dritte Element auf dem Datenstack von oben. Das Wort `identical?` vergleicht deshalb das durch `dup` duplizierte Element auf dem Datastack. Das Programm

```
dup identical?
```

kann nur `t` als Ergebnis haben, wenn `dup` identitätserhaltend ist.

Die Erkenntnis ist entscheidend. Das Konzept der Identität, d.h. der Verfolgbarkeit fordert dem Wort `dup` eine identitätserhaltende Wirkung ab. Das ergibt sich zwingend aus der Überlegung, dass die Identität eines Elements dadurch gegeben ist, dass man es gleichermaßen mit einem Zeiger auf seinem Weg auf dem Datastack und als Element beim `pushen` und `topen` auf einem Stapel verfolgen kann. Unter diesen Annahmen *muss* `dup` die Identität beim Anlegen eines Duplikats erhalten.

### 4.3 Identitäten durch Referenzen

Auch wenn Sie argumentativ verstanden haben, dass `dup` ein identisches Element auf dem Datastack ablegt, so ist doch schwer anschaulich nachzuvollziehen, warum gleiche Elemente auf dem Datastack mal identisch sind und mal nicht.

```
> clear hi hi identical?
f
> hi dup identical?
f t
```

Wie erklärt sich, dass die gleichen Worte nicht identisch sind, ein geduptes Wort hingegen schon?

Wir haben das Konzept der Identität eingeführt mit der Idee, ein Datum auf seinem Weg auf dem Datenstack eindeutig verfolgen zu können. In der Tat ist das Konzept des Verfolgers – der Fachbegriff lautet „Referenz“ oder auch „Zeiger“ – eine Alternative, um sich zu veranschaulichen, wann zwei Elemente identisch sind bzw. wann sie es nicht sind.

Wenn Sie das Wort `hi` eingeben, erzeugt Consize auf dem Datenstack ein *neues* Wort. Jedes Mal, wenn Sie `hi` eingeben, wird ein neues Wort erzeugt. Die Wörter gleichen einander zwar, sind aber nicht identisch.

Mit dem Wort `dup` wird genau das nicht getan: `dup` erzeugt kein neues, gleiches Wort, sondern es wird nur eine Referenz des Wortes auf dem Stack abgelegt. Es ist wie mit den zwei Fotos, die ein und dieselbe Person zeigen – die Fotos sind gleichsam Referenzen auf ein und dieselbe Person; die Fotos sind nicht die Person.

Die Daten, die sich auf dem Datastack befinden, tummeln sich tatsächlich irgendwo im Speicher Ihres Computers. Der Datastack besteht ausschließlich aus Referenzen, die auf die Daten im Speicher zeigen. Wann immer Sie `hi` eingeben, wird das Wort im Speicher angelegt und eine Referenz verweist vom Datastack auf dieses Wort. Das Wort `dup` erzeugt kein neues Wort, sondern dupliziert nur die Referenz. Somit zeigen zwei Referenzen auf dasselbe Datenelement. Da man bei der Ausgabe des Datastacks die Referenzen nicht sieht, sondern nur die Repräsentationen der referenzierten Daten, ist die Täuschung perfekt: Man kann nicht erkennen, welche Daten auf dem Datastack identisch sind.

Den Blick hinter die Kulissen erlaubt einzig `identical?`. `identical?` schaut nämlich, ob zwei Referenzen auf ein und dasselbe Datum verwei-

sen. Ist dem so, dann liefert `identical?` ein `t` zurück, ansonsten `f`. Das Wort `equal?` wird also immer erst einmal auf `identical?` testen, um sich den Test auf Gleichheit der Daten möglicherweise zu ersparen.

#### 4.4 Referenzielle Transparenz durch Immutabilität

Das Konzept der Referenz ist ein Alternativkonzept, das die Idee der Identität (der Verfolgbarkeit von Daten) anschaulicher erfasst. Tatsächlich arbeiten Implementierungen funktionaler Sprachen intern sehr oft mit Referenzen. Referenzen optimieren die interne Datenhaltung. Es ist deutlich schneller und speichergünstiger, bei einem `dup` eine Referenz zu kopieren statt die Kopie eines Datums anzulegen, was bei Stapeln und Mappings sehr aufwendig sein kann. Abseits solcher Effizienzüberlegungen hat das Konzept der Referenz bzw. der Identität keine Auswirkungen oder Effekte bei der Ausführung von funktionalen oder deklarativen Programmen. Man spricht deshalb auch von referenzieller Transparenz. Mit anderen Worten: Ob man intern mit Referenzen arbeitet oder nicht, ohne das Wort `identical?` ist überhaupt nicht auszumachen, ob mit Referenzen gearbeitet wird oder nicht. Das Referenzkonzept hat keine Bedeutung in der funktionalen Programmierung.

Die referenzielle Transparenz wird in funktionalen Sprachen – und damit auch in Consize – durch die Unveränderlichkeit (Immutabilität) der Daten gewährleistet. Ein Datum kann niemals und unter keinen Umständen verändert (mutiert) werden. Wenn wir beispielsweise ein Element auf einen Stack `pushen`, wird der Stack niemals verändert, sondern ein neuer Stack erzeugt, der alle Element des bisherigen Stacks inklusive des `gepushten` Elements enthält. Der Stack *vor* dem `push` kann niemals mit dem Stack *nach* dem `push` identisch sein.<sup>1</sup> In imperativen Programmiersprachen werden Sie feststellen, dass das Verändern von Daten über die Referenz die Identität erhält.

Wenn Mutabilität zur referentiellen Transparenz führt, ist das Konzept der Referenz und damit der Identität eigentlich hinfällig, oder? Wozu gibt es dann das Wort `identical?` in Consize?

Der Einwand ist vollkommen berechtigt. Man kann das Konzept der Identität in einer funktionalen Sprache wie Consize zwar definieren und einführen, die Programmausführung kann und darf davon jedoch nicht abhängen.

Der Grund, warum ich das Wort `identical?` dennoch in Consize aufgenommen habe, ist ein didaktischer: Sie sollen sich des Unterschieds von Gleichheit und Identität sehr wohl bewusst sein. Insbesondere in imperativen Programmiersprachen wie Java oder C# macht es einen erheblichen Unterschied, ob Sie mit einer Referenz arbeiten oder nicht; dort sind die meisten Datentypen mutabel und über die Referenz veränderbar. In Consize brauchen Sie nur in Daten zu denken, in vielen imperativen Sprachen müssen Sie in Daten und Referenzen denken. Das heißt: In Consize ist `equal?` alles, was sie brauchen, um Daten zu vergleichen. Mit `identical?` haben Sie den Luxus, die Verfolgbarkeit eines Datums auf dem Datenstack zu testen – darüber hinaus ist das Konzept (in einer funktionalen Sprache) nutzlos. Weder darf noch kann ein Programm in Consize von `identical?` abhängen – weshalb das Wort konsequenterweise auch gar nicht erst genutzt werden, besser noch: gar nicht erst existieren sollte.

---

<sup>1</sup> Wie ein `push pop` dennoch identitätserhaltend sein kann, ist eine Sache der intern verwendeten Datenstrukturen. Die Datenstruktur des Stapels kann sehr gut in Form einer [einfach verketteten Liste](#) realisiert werden.

Darum seien Sie an dieser Stelle wieder in eine Welt ohne `identical`?  
entlassen.

---

## 5 Die Prelude

---

Consize ist eine sehr primitive Sprache, die allein mit rund 50 atomaren Wörtern auskommt. Damit kann man zwar programmieren – aber das Programmieren mit Consizes einzig mit den Wörtern der VM ist zu umständlich und macht wenig Spaß. Viel schlimmer noch: Consizes ist anfänglich nicht einmal in der Lage, mit Ihnen zu interagieren. Wenn Sie Consizes starten, möchte Consizes eine Datei namens `prelude.txt` verarbeiten, die sogenannte Prelude. Fehlt die Datei oder steht dort Unsinn drin, macht Consizes entweder gar nichts – oder eben eine Menge Unsinn. So oder so, wir können mit Consizes im Urzustand kaum sinnvoll arbeiten.

Der Clou an Consizes ist: Die Sprache lässt sich erweitern. Machen wir uns die Sprache komfortabel. Wir werden etliche neue Wörter einführen, die hilfreiche Abstraktionen bieten, wir werden die Syntax erweitern und Consizes interaktiv machen. Die dazu nötigen Programme stehen in der Prelude.

In diesem Kapitel sind alle Programmzeilen der Prelude durch ein vorangestelltes „>>“ (inkl. Leerzeichen) ausgezeichnet. Diese Markierung soll Ihnen helfen, in dem gesamten Text dieses Kapitels mit all seinen Erläuterungen und Beispielen die entscheidenden Programmzeilen zu identifizieren. Übrigens helfen die Markierungen auch Consizes, um den Quelltext aus der Dokumentation zu filtern.

Auch wenn der Einstieg in die Prelude gleich ein unvermittelter Einstieg in die Programmierung mit Consizes ist: Sie werden sehen, Consizes ist nicht wirklich schwer zu verstehen. In Consizes zerlegt man Programme systematisch in kleine Miniprogramme, die zu schreiben vergleichbar ist mit der Herausforderung von Rätselaufgaben. Und Sie haben einen immensen Vorteil bei der Arbeit mit Consizes: Ihnen stehen die Consizes-Erweiterungen direkt zur Verfügung. Sie arbeiten mit einer geladenen Prelude, um die Prelude zu verstehen. Das ist einfacher und unkomplizierter als sich das anhört. Sie können ein paar Hilfsmittel, wie z.B. den Debugger nutzen, um sich die Arbeitsweise von Consizes zu veranschaulichen.

### 5.1 Vorbereitungen: Was sein muss und was nützlich ist

#### 5.1.1 Consizes-Lizenz

Die Prelude ist [Open-Source-Software](#) (OSS). Consizes soll als Bildungsgut allen Interessierten frei zur Verfügung stehen.

```
>> %%% A Prelude for Consizes in Consizes
>> %%% Copyright (c) 2012, Dominikus Herzberg, Heilbronn University
>> %%% New BSD License: http://opensource.org/licenses/BSD-3-Clause
```

#### 5.1.2 Wie man Programme schreibt

In Consizes schreibt man keine Programme durch endlos lange Wortfolgen – damit bliebe ein entscheidender Mechanismus der Consizes-VM ungenutzt. In Consizes schreibt man Programme, indem man Wörter zum

globalen Wörterbuch hinzufügt und das Wort mit einer Quotierung – einem Mini-Programm, wenn man so möchte – assoziiert. Das ist ein sehr einfacher, aber auch sehr leistungsfähiger Abstraktionsmechanismus. So abstrahiert das Wort `-rot` die Quotierung `[ rot rot ]`. Man spricht auch von einer benannten Abstraktion: `-rot` ist der „Name“ für die Abstraktion `[ rot rot ]`. Sie erinnern sich an Kap. 2.7, S. 17:

```
> \ -rot source
[ rot rot ]
```

Mit dieser Abstraktionstechnik werden große und umfangreiche Programme überhaupt erst realisierbar. Wir Menschen müssen mehr oder minder große Programmeinheiten unter für uns sinngebenden Namen fassen können. Unsere intellektuellen Fähigkeiten stoßen beim Programmieren ansonsten rasch an ihre Grenzen.

Fortan wollen wir die Definition neuer Einträge im Wörterbuch wie folgt notieren: Ein Doppelpunkt `:` leitet die Definition ein. Nach dem Doppelpunkt folgt das Wort, dann optional der Stapeleffekt und anschließend die das Wort definierende Wortfolge. Ein Semikolon schließt die Definition ab. Die definierende Wortfolge ist die mit dem Wort assoziierte Quotierung, allerdings ohne die eckigen Klammern.

In Anlehnung an die in Kap. 2.2 formulierte Grammatik hält die folgende Regel den Aufbau einer Wort-Definition fest.

$$\langle \text{definition} \rangle = \text{'}' \langle \text{separator} \rangle \langle \text{word} \rangle [ \langle \text{separator} \rangle \langle \text{stackeffect} \rangle ] \langle \text{separator} \rangle \langle \text{program} \rangle \langle \text{separator} \rangle \text{'}'$$

Die Angabe von Stapeleffekten kennen Sie bereits aus Kap. 3. Alle Wörter der Consizes-VM sind dort mit ihren Stapeleffekten angegeben worden. Details zur Umsetzung von  $\langle \text{definition} \rangle$  finden sich in Kap. 5.4.2, S. 60.

Die Syntax zur Definition neuer Wörter kann nicht direkt verwendet werden, da die Consizes-VM sie nicht kennt. Es bedarf eines kleinen Consizes-Programms, das diese komfortable Art der Definition neuer Wörter zur Verfügung stellt. Dieses Consizes-Programm ist als sogenanntes „Bootimage“ in der Datei `bootimage.txt` abgelegt und muss zuvor geladen werden. Der Ladevorgang fährt die Consizes-VM in einen programmiertauglichen Zustand hoch, was man als „[Booten](#)“ bezeichnen kann.

```
>> \ bootimage.txt slurp uncomment tokenize
>>   emptystack \ continue push \ concat push \ rot push
>>   \ top push \ swap push \ pop push \ dup push \ swap push
>>   call/cc
```

Der Code ist nichts anderes als eine ausführliche Variante von:

```
\ bootimage.txt load call
```

Da die Wörter `load` und `call` zu Beginn der Abarbeitung der Prelude jedoch noch nicht bereit stehen, sind hier beide Wörter händisch aufgelöst worden. In Kap. 5.6.3 ist beschrieben, was in dem Bootimage steht und wie es erzeugt wird.

### 5.1.3 Grundverhalten von `read-word` und `read-mapping`

Der durch `stepcc` beschriebene Ausführungsmechanismus der Consizes-VM ist in engen Grenzen konfigurierbar. Das Verhalten der beiden Wörter

`read-word` und `read-mapping` kann vom Anwender bzw. von der Anwenderin frei definiert werden, siehe Kap. 3.11.

Die Prelude assoziiert beide Wörter mit „leeren“ Programmen. Das bedeutet: Wörter genauso wie Mappings werden einfach auf dem Datastack belassen.

```
>> : read-word    ( wrd -- wrd ) ;
>> : read-mapping ( map -- map ) ;
```

Durch Anpassung dieser Wörter sind beispielsweise Kodierungskonventionen für Wörter und Mappings einführbar, die eine Sonderbehandlung erfahren oder eine Vorverarbeitung erfordern sollen. Zu beachten ist, dass eine Neudefinition dieser Meta-Wörter kompatibel bleibt zum Standardverhalten. Sonst besteht das Risiko, existierenden Code in größerem Umfang zu brechen.

#### 5.1.4 Mehr davon: Stack Shuffler

In praktisch allen höheren Programmiersprachen können Variablen in Form von Zuweisungen (bei imperativen Sprachen) oder Substitutionen (bei deklarativen/funktionalen Sprachen) verwendet werden. Variablen erfüllen dabei hauptsächlich zwei Aufgaben: sie helfen Werte zwischenspeichern und sie neu zu arrangieren für den Aufruf von Funktionen, Prozeduren oder Methoden. Darüber hinaus bieten Variablennamen semantische Brücken für den Programmierer bzw. die Programmiererin an, sich den Inhalt oder den Zweck eines Wertes zu merken.

Da es in Consize keine Variablen gibt, muss das Zwischenspeichern von Werten über den Datastack (beispielsweise per `dup`) geschehen und das Rearrangieren von Werten über Stack-Shuffling gelöst werden. Es ist daher sehr hilfreich, weitere Wörter als Abstraktionen zum Stack-Shuffling zur Verfügung zu haben.

Da Variablennamen als semantische Brücken beim Programmieren fehlen, gibt es nur eine Möglichkeit, um Consize-Programme übersichtlich zu halten: Man muss die Definition von neuen Wörtern kurz halten und gegebenenfalls weitere Wörter einführen, um über geeignete Abstraktionen die Programme lesbar zu halten. Das ist auch der Grund, warum sich Wortdefinitionen oft nur über ein, zwei oder drei Zeilen erstrecken und praktisch nie über ein Dutzend Codezeilen hinausgehen. Und es unterstreicht auch die Bedeutung der Angabe von Stackeffekten. Stackeffekte beschreiben oft hinreichend genau, was ein Wort tut, so dass man sich das *Wie* der Manipulation der Werte auf dem Datastack nicht merken muss.

```
>> : 2drop ( x y -- ) drop drop ;
>> : 3drop ( x y z -- ) drop drop drop ;
>> : 2dup ( x y -- x y x y ) over over ;
>> : dupd ( x y -- x x y ) swap dup rot ;
>> : -rot ( x y z -- z x y ) rot rot ;
>> : rot4 ( x y z u -- y z u x ) [ rot ] dip swap ;
>> : -rot4 ( x y z u -- u x y z ) swap [ -rot ] dip ;
>> : 3dup ( x y z -- x y z x y z )
>>   rot dup -rot4 rot dup -rot4 rot dup -rot4 ;
>> : over ( x y -- x y x ) swap dup -rot ;
>> : 2over ( x y z -- x y z x y ) rot dup -rot4 rot dup -rot4 ;
>> : nip ( x y -- y ) swap drop ;
```

```
>> : 2nip ( x y z -- z ) nip nip ;
>> : pick ( x y z -- x y z x ) rot dup [ -rot ] dip ;
```

Es deutet sich bei den Definitionen einiger Stack-Shuffler wie z.B. `rot` und `-rot` an, dass es eine Alternative zum Stack-Shuffling gibt, die die Abwesenheit von Variablen elegant kompensiert: Kombinatoren, siehe Kap. 5.3.

Zum Beispiel kann `rot4` auch wie folgt definiert werden; das Stack-Shuffling ist im Kopf kaum mehr nachvollziehbar – die Kommentare mögen bei den gedanklichen Schritten helfen.

```
: rot4 ( x y z u -- y z u x )
  [ ] swap push swap push % x y [ z u ]
  rot swap                % y x [ z u ]
  dup top swap pop top    % y x z u
  rot ;                   % y z u x
```

Da die Stack-Shuffler der Consize-VM nicht über die ersten drei Elemente auf dem Datastack hinaus reichen, muss hier zu dem Trick gegriffen werden, Elemente vom Datastack in einen Stapel zu „packen“, um auf das vierte Element von oben, hier `x`, zugreifen zu können.

Mit Hilfe des `dip`-Kombinators wird der Code radikal kürzer und gleichzeitig auch leicht nachvollziehbar. Es werden zunächst die unteren drei Werte, `x y z`, auf dem Datastack per `rot` rotiert, dann die obersten zwei Werte mit `swap` getauscht.

Interessant ist auch, dass bei dem Einsatz von Kombinatoren die Reversibilität des Verhaltens von `-rot4` zu `rot4` klar zutage tritt: `-rot4` tauscht erst die beiden obersten Elemente und rotiert dann die untersten drei. Die alternative Definition von `-rot4` mittels

```
: -rot4 ( x y z u -- u x y z ) rot4 rot4 rot4 ;
```

macht nicht nur deutlich schwerer nachvollziehbar, was die mehrfache Anwendung eines `rot4` bewirkt, sondern sie hat auch die dreifachen Laufzeitkosten eines `rot4`.

Mehr zu `dip` und den anderen Kombinatoren steht in Kap. 5.3.

### 5.1.5 Freunde und Helferlein

Bei der Arbeit mit Consize werden Sie feststellen, dass Sie einige Wortkombinationen sehr häufig benötigen. Es macht Sinn, eigene Wörter dafür einzuführen, um den Consize-Code lesbarer zu machen.

Die folgenden Wörter sind hilfreiche Begleiter bei der Arbeit mit Stapeln.

```
>> : swapu ( itm stk -- stk' ) swap push ;
>> : unpush ( stk -- stk' itm ) dup pop swap top ;
>> : empty? ( stk -- t/f ) [ ] equal? ;
>> : size ( seq -- n ) dup empty? [ drop 0 ] [ pop size 1 + ] if ;
```

Die Anzahl der Elemente in einer Sequenz liefert `size` zurück.

```
> clear [ ] size [ x y z ] size
0 3
```

Der Datentyp „nil“ ist in Consize nur indirekt über ein `top` eines leeren Stapels definiert. Um `nil` als Typ zugreifbar zu haben, gibt es das Wort `nil`.



Das Wort `delete` entfernt ein Wort mit seiner assoziierten Quotierung aus dem Wörterbuch der Consize-VM. Das Wort `values` gibt – im Gegensatz zu `keys` – alle mit den Schlüsseln assoziierten Zielwerte als Sequenz zurück.

```
>> : nil ( -- nil ) [ ] top ;
>> : delete ( itm -- ) get-dict dissoc set-dict ;
>> : values ( dict -- seq ) dup keys swap [ _|_ get ] swapu map ;
```

## 5.2 Grundlegende Kombinatoren

### 5.2.1 Quotierungen aufrufen

Wenn Consize ein Programm abarbeitet, ist es zu einem guten Teil mit der Auflösung benannter Abstraktionen beschäftigt: Das oberste Wort auf dem Callstack wird durch den Inhalt der mit ihm assoziierten Quotierung ersetzt. Genauer: Das Wort wird vom Callstack entfernt und die assoziierte Quotierung mit dem Callstack konkateniert.

Für den Aufruf anonymer Abstraktionen, d.h. Quotierungen, die nicht im Wörterbuch mit einem Wort verknüpft sind, gilt im Grunde derselbe Ablauf. Das Wort `call` erwartet eine Quotierung auf dem Datastack, die es mit dem Callstack konkateniert. Die Notation des Stackeffekts soll genau das andeuten, was per `call/cc` umgesetzt wird.

```
>> : call ( ... [ quot ] -- ... | quot )
>> [ swap unpush rot concat continue ] call/cc ;
```

Das per `call/cc` initiierte Programm tauscht Data- und Callstack der aktuellen Continuation (`swap`), holt das oberste Element (die Quotierung) vom Datastack (`unpush`), bringt den Callstack wieder nach oben (`rot`) und konkateniert Quotierung und Callstack miteinander (`concat`). Anschließend übernimmt die so veränderte Continuation wieder die Ausführung (`continue`).

Die Technik, mit `call/cc` und `continue` ein Programm zu unterbrechen, es zu modifizieren und dann fortzusetzen, nennt man „[Metaprogrammierung](#)“. Das mit `call/cc` mitgegebene Programm modifiziert das aktuell in der Ausführung begriffene Programm. Diese zur Laufzeit des Programms stattfindende Änderung nennt man deshalb bei Bedarf auch genauer „Laufzeit-Metaprogrammierung“.

Mit `call` lässt sich die Ausführung eines Programms verzögern.

```
> clear [ 4 2 3 + ]
[ 4 2 3 + ]
> call
4 5
```

Das Wort `fcall` (für „*call via a function*“) arbeitet ähnlich wie `call`. Es erzeugt im Gegensatz zu `call` über `func` einen eigenen Ausführungskontext, bekommt einen leeren Stapel als Datastack, wendet die Funktion darauf an (`apply`) und liefert das Ausführungsergebnis als Stapel zurück und zwar mit dem „umgekehrten“ Ergebnisstapel (`reverse`).

```
>> : fcall ( quot -- seq ) get-dict func [ ] swap apply reverse ;

> clear [ 4 2 3 + ] fcall
[ 4 5 ]
```

### 5.2.2 Boolesche Werte und die binäre Wahl

Ein grundlegendes Feature einer turingvollständigen Programmiersprache ist es, binäre Entscheidungen treffen zu können: wähle – abhängig von einem Entscheidungswert – entweder dieses oder jenes. Im Idealfall ist der Entscheidungswert selber binär. Zu Ehren des Logikers [GEORGE BOOLE](#) heißen diese beiden Werte „boolesche Werte“ und werden mit „wahr“ (*true*) und „falsch“ (*false*) bezeichnet. In Consize repräsentieren die Wörter **t** und **f** auf dem Datastack diese beiden Optionen.

Die Consize-VM bietet von sich aus kein Wort an, das Entscheidungen realisiert; in vielen Programmiersprachen dient dazu das *if*. Trotzdem – und das ist das Spannende daran – ist ein Wort für Entscheidungen nachrüstbar. Der [Lambda-Kalkül](#) macht es übrigens genauso.

Das Wort **choose** ( **t/f this that -- this/that** ) bilde die Grundlage für die binäre Auswahl. **choose** erwartet auf dem Datastack einen booleschen Wert und zwei Wahlwerte, die wir im Stapeleffekt mit **this** und **that** bezeichnen. Abhängig vom booleschen Wert lässt **choose** entweder **this** oder **that** auf dem Datastack zurück.

Die Bedeutung von **t** und **f** erschließt sich nur im Kontext des Wortes **choose** – und so bilden die mit den booleschen Werten assoziierten Programme exakt das beschriebene Verhalten nach: **t** entfernt **that** vom Datastack, um **this** zu erhalten, **f** macht es genau anders herum. Ohne den Kontext von **choose** sind die booleschen Werte **t** und **f** bedeutungslos.

```
>> : t ( this that -- this ) drop ;
>> : f ( this that -- that ) swap drop ;
>> : true ( -- t ) \ t ;
>> : false ( -- f ) \ f ;
```

Die Wörter **true** und **false** dienen lediglich dem Lese- und Schreibkomfort; es ist rasch vergessen, das Escape-Wort einem **t** bzw. **f** voranzustellen.

Das Wort **choose** muss per **rot** den booleschen Wert lediglich oben auf den Datastack bringen, die Bedeutung (sprich: Semantik) von **t** bzw. **f** mit **lookup** nachschlagen, und die ermittelte Quotierung mit **call** aufrufen. Das Wort **choose** macht nicht viel mehr als das mit **t** bzw. **f** assoziierte Verhalten zu aktivieren.

```
: choose ( t/f this that -- this/that ) rot lookup call ;
```

Mit dieser Definition könnte man es bewenden lassen, doch es gibt einen guten Grund, die Auslegung boolescher Werte ein wenig zu erweitern.

Falls sich bei **choose** weder ein **t** noch ein **f** an dritter Stelle, sondern ein beliebiger anderer Wert auf dem Datastack befindet, dann hat der Programmierer bzw. die Programmiererin den im Stapeleffekt dokumentierten „Vertrag“ zum Gebrauch des Wortes **choose** gebrochen. Die Folgen aus einem fälschlichen Gebrauch des Wortes hat der Programmierer bzw. die Programmiererin zu verantworten. Es spielt keine Rolle, ob der Fehlgebrauch unabsichtlich erfolgt oder nicht. Dieses Prinzip ist so wichtig, dass es nicht nur „im richtigen Leben“ sondern auch in der Softwaretechnik zur Anwendung kommt: Wer einen Vertrag verletzt, darf die andere Partei (in dem Fall Consize) nicht für den entstehenden Schaden verantwortlich machen. Man muss beim Programmieren durchaus Vorsicht walten lassen.

Man könnte eine Vertragsverletzung durch eine Fehlermeldung abfangen, um auf das Problem aufmerksam zu machen und gegebenenfalls darauf zu reagieren. Allerdings verändert ein solches Vorgehen das Wort `choose` von einer binären zu einer ternären Wahl: wähle `this` im Fall von `t`, `that` im Fall von `f` und mache etwas gänzlich anderes, wenn ein anderes Datum an dritter Position im Datastack steht. Das passt nicht zu unseren anfänglichen Intentionen, mit `choose` lediglich eine binäre Auswahl treffen zu wollen.

Programmiersprachen mit [dynamischer Typisierung](#) – Consize gehört dazu – unterstützen zwar ein strikt binäres Verhalten, vermeiden aber Fehlermeldungen in der Regel durch eine simple Regelung: Jeder Wert, der sich von `f` unterscheidet wird so interpretiert als sei er ein logisches `true`. Wenn der dritte Wert auf dem Datastack nicht `f` ist, dann wählt `choose` `this`, ansonsten `that` aus. Die Definition von `choose` verändert sich entsprechend geringfügig. Der Stern `*` im Stapeleffekt steht für einen beliebigen Wert außer `f`.

```
>> : choose ( f/* this that -- that/this )
>>   swap rot false equal? lookup call ;
```

Sprachen mit [statischer Typisierung](#) schließen den Fehlerfall durch eine Typüberprüfung vor Programmausführung aus. Wäre Consize eine statisch typisierte Sprache, könnte die Verwendung eines booleschen Wertes sicher gestellt werden, und es würde die eingangs angegebene Definition von `choose` ausreichen.

Das Verhalten von `choose` demonstriert folgende Beispielergabe an der Konsole.

```
> clear false this that
f this that
> choose
that
> [ 1 2 3 ] this that choose
that this
```

Die logischen Operationen der [Booleschen Algebra](#) `and` (Konjunktion), `or` (Disjunktion) und `not` (Negation) sind dieser erweiterten Interpretation logischer Werte („alles was nicht `f` ist, ist logisch `t`“) angepasst.

```
>> : and ( f/* f/* -- t/f ) over choose ; % Factor
>> : or  ( f/* f/* -- t/f ) dupd choose ; % Factor
>> : not ( f/* -- t/f ) false true choose ;
```

Ein kurzes Beispiel zeigt den Gebrauch von `and`.

```
> clear true true and
t
> false true and
t f
```

### 5.2.3 Entscheiden, wo's lang geht

Das Wort `choose` realisiert zwar eine binäre Wahl, doch werden damit noch keine unterschiedlichen Verhaltenskonsequenzen umgesetzt. Das ist erst dann der Fall, wenn `this` und `that` Quotierungen sind, die nach einem `choose` per `call` aufgerufen werden. Dafür gibt es das Wort `if`; die Quotierungen werden im Stapeleffekt mit `then` und `else` bezeichnet.

Dass die Auswirkungen auf den Datastack von den beiden Quotierungen abhängen und nicht vorhersehbar sind, deuten die Punkte auf der rechten Seite des Stapeleffekts an.

```
>> : if ( f/* then else -- ... ) choose call ;
>> : if-not ( f/* then else -- ... ) swap if ;
>> : when ( f/* then -- ... ) [ ] if ;
>> : unless ( f/* else -- ... ) [ ] if-not ;
```

Die Wörter **if-not**, **when** und **unless** sind hilfreiche Abstraktionen, die alle Variationen des **if**-Themas abdecken: **if-not** vertauscht die Rolle der beiden Quotierungen **then** und **else** (dem gedanklich eine Negation des booleschen Wertes entspricht), **when** führt die Quotierung nur aus, wenn der vorangehende Wert als logisch *true* gilt, **unless**, wenn der vorangehende Wert *false* ist.

```
> clear 5 dup 3 < [ 1 + ] [ 1 - ] if
4
> clear 5 dup 3 < [ 1 + ] [ 1 - ] if-not
6
> clear 5 true [ 1 + ] when
6
> clear 5 false [ 1 - ] unless
4
```

Ein Bedingungs-*wert* außer **f** gilt im Kontext eines **if**, **when** oder **unless** als logisch *true*, selbst wenn der Bedingungs-*wert* nicht gleich **t** ist. Oft ist es so, dass man mit logisch wahren Bedingungs-*werten* genau dann weiter arbeiten möchte, wenn sie nicht **t** sind.

Die Wörter **if\***, **when\*** und **unless\*** sind Varianten, die den Bedingungs-*wert*, so er denn logisch *true* ist, vor der Ausführung eines gedachten **if**, **when** bzw. **unless** auf dem Datastack duplizieren, so dass der Bedingungs-*wert* für die weitere Verarbeitung erhalten bleibt. Ist der Bedingungs-*wert* **f**, so unterbleibt die Duplizierung und **if\***, **when\*** bzw. **unless\*** arbeiten wie ihre „sternlosen“ Vorbilder **if**, **when** und **unless**.

```
>> : if*
>>   ( ..a ? true: '( ..a ? -- ..b ' ) false: '( ..a -- ..b ' ) -- ..b )
>>   pick [ drop call ] [ 2nip call ] if ; % Factor
>> : when* ( ? t -- ) over [ call ] [ 2drop ] if ; % Factor
>> : unless* ( ? f -- ) over [ drop ] [ nip call ] if ; % Factor

> clear 6 [ 1 + ] [ 0 ] if*
7
> clear false [ 1 + ] [ 0 ] if*
0
> 6 [ 1 + ] when*
7
> clear 5 6 [ 1 - ] unless*
5 6
> clear 5 false [ 1 - ] unless*
4
```

Die mit dem Wort **case** umgesetzte *n*-äre Entscheidung verallgemeinert das Konzept der binären **if**-Entscheidung. Eine Handlungsalternative hängt bei **case** nicht ab von zwei Alternativen (**f** oder nicht **f**), sondern von einem Wert aus einer Menge von Werten. Die Grundfunktionalität von **case** ist bereits durch Mappings gegeben; nach einem **get** muss im

Grunde nur noch ein `call` folgen. Mit dem Auswahlwert `:else` besteht die Option, eine Reaktion zu initiieren, wenn kein sonstiger Auswahlwert in `case` zutrifft.

Das Wort `SYMBOL:` setzt mit dem nachfolgenden Wort, hier `:else`, eine Definition auf, die das Wort mit sich selbst als Datenwert definiert (hier `: :else \ :else ;`).

```
>> SYMBOL: :else
>> : case ( val { val' quot ... } -- | quot )
>> :else over [ _|_ ] get get call ;
```

Es ist zu beachten, dass die Zielwerte in einem „case-Mapping“ stets Quotierungen sind.

```
> clear 3 \ red
3 red
> { \ red [ 1 + ] \ blue [ 1 - ] :else [ ] } case
4
> blue { \ red [ 1 + ] \ blue [ 1 - ] :else [ ] } case
3
> black { \ red [ 1 + ] \ blue [ 1 - ] :else [ ] } case
3
```

Die Verschachtelung von `if`-Wörtern in den `else`-Quotierungen eines `if`-Worts erzeugt rasch unleserlichen Code. Als Alternative bietet sich das syntaktisch übersichtlichere `cond` an, das die `test`- und `then`-Quotierungen verschachtlungsfrei zu notieren erlaubt. Einer `test`-Quotierung folgt eine `then`-Quotierung; im `else`-Fall folgt die nächste `test`-Quotierung usw.; die Terminologie orientiert sich an der Beschreibung des Stapeleffekts für `if`. Eine optionale `else`-Quotierung dient für den Fall, wenn alle `test`-Quotierungen fehlschlagen.

```
>> : cond ( [ test1 then1 test2 then3 ... else ] -- | then/else )
>>   dup empty?           % anything left to test?
>>   [ drop ]             % no: quit
>>   [ unpush swap dup empty? % only one quotation left?
>>   [ drop call ]         % yes: call 'else'
>>   [ unpush swap         % otherwise:
>>   [ ] \ cond push swapu % prepare 'cond' recursion
>>   [ call ] 2dip if ]    % call 'testN' and apply 'if'
>>   if ]
>>   if ;
```

Per Konvention wird die Folge von Quotierungen vor einem `cond` in runden Klammern notiert, nicht zuletzt, um eine visuell leichtere Abgrenzung zu haben. Beachten Sie, dass in den `test`-Quotierungen in aller Regel ein `dup` nötig ist, nicht zuletzt, um den Test-Wert für nachfolgende Bedingungen zu erhalten.

```
> clear

> 7 ( [ dup 0 > ] [ 1 + ] [ dup 0 < ] [ 1 - ] [ ] ) cond
8
> -7 ( [ dup 0 > ] [ 1 + ] [ dup 0 < ] [ 1 - ] [ ] ) cond
8 -8
> 0 ( [ dup 0 > ] [ 1 + ] [ dup 0 < ] [ 1 - ] [ ] ) cond
8 -8 0
```

Ein `case` lässt sich immer über ein `cond` nachbilden, allerdings hat ein `case` durch das Mapping eine konstante Laufzeit, während die Laufzeit von `cond` mit jeder Verschachtelung linear anwächst. Umgekehrt kann nicht jedes `cond` ohne weiteres in ein äquivalentes `case` umgewandelt werden.

### 5.3 Kombiniere: Kombinatoren

Ein Wort heißt „Kombinator“, wenn es eine oder mehr Quotierungen auf dem Datastack erwartet und die Quotierung(en) zur Abarbeitung bringt. So ist z.B. das Wort `call` ein Kombinator, das Wort `concat` hingegen nicht.

Die Kapitel 5.3.1-5.3.5 behandeln verschiedene Klassen von Kombinatoren, die die Notwendigkeit des Stack-Shufflings minimieren und deshalb lesbarere Programme mit sich bringen. Die Definitionen der Kombinatoren orientieren sich in vielen Fällen an der konkatenativen Sprache Factor. In Kap. 5.3.6 geht es um Sequenzkombinatoren, die eine Anwendung eines Kombinator über eine Folge von Daten realisieren.

#### 5.3.1 „Abtauch“-Kombinatoren: `dip`

Die `dip`-Kombinatoren rufen wie ein `call` eine Quotierung auf dem Datastack auf. Im Gegensatz zu einem reinen `call` gehen die unmittelbar „vor“ der Quotierung stehenden Daten für die Dauer des Aufrufs gleichermaßen auf Tauchstation; das englische Wort *dip* ist hier im Sinne von „abtauchen“ zu verstehen. Nach dem Aufruf erscheinen die „abgetauchten“ Daten wieder auf dem Datastack. Das Wort `dip` verbirgt ein Element vor der aufzurufenden Quotierung, das Wort `2dip` verbirgt zwei Elemente, `3dip` drei und `4dip` vier.

```
>> : dip ( x quot -- x ) [ ] rot push \ \ push concat call ;
>> : 2dip ( x y quot -- x y ) swap [ dip ] dip ;
>> : 3dip ( x y z quot -- x y z ) swap [ 2dip ] dip ;
>> : 4dip ( w x y z quot -- w x y z ) swap [ 3dip ] dip ;

> clear [ ] 4 5 [ push ] dip
[ 4 ] 5
> clear [ ] 4 5 [ drop ] 2dip
4 5
```

Die Definition  $n$ -facher `dip`-Kombinatoren folgt einem einfachen Schema: Das einleitende `swap` und das beendende `dip` bleiben immer gleich; lediglich die Quotierung greift auf die vorhergehende `dip`-Definition zurück. In der Rückverfolgung des Bildungsgesetzes kann man auch fragen: Wie müsste demnach die Definition von `dip` lauten? In der Quotierung wäre ein `0dip` zu verwenden. Ein `dip`, das 0 Werte auf dem Datastack „abtauchen“ lässt, ist identisch mit `call`.

```
: dip ( x quot -- x ) swap [ call ] dip ;
```

Diese Definition nimmt auf sich selbst Bezug und liefert auch in ihrer Auflösung keine weitere Erkenntnisse. Einen Hinweis, wie `dip` implementiert werden kann, liefert sie dennoch: Es ist der Versuch, das im Stapel-effekt mit `x` bezeichnete Element nicht mehr vor der Quotierung, sondern es per `swap` hinter die Quotierung zu bekommen, so dass der `call` der Quotierung `quot` das Element `x` nicht mehr erfasst.

Man kann dieses Verhalten zum Beispiel durch die Manipulation der aktuellen Continuation nach einem `swap` erreichen; der Stapeleffekt deutet an, was hier gemacht wird.

```
: dip ( itm quot -- quot | call \ itm )
  swap [ swap unpush rot swap push \ \ push \ call push continue ]
  call/cc
```

Grundsätzlich sollte man den Einsatz von Continuations vermeiden wann immer möglich; das hat formale Gründe, auf die in Kap. 5.4.3 näher eingegangen wird. Alternativ kann man den vor der Quotierung stehenden Wert ans Ende der Quotierung anhängen und mit einem Escape-Wort sicherstellen, dass der Wert nicht als zu interpretierendes Wort behandelt wird. Genau das tut die in der Prelude verwendete Definition, siehe oben. Alternative Definitionen sind:

```
: dip ( x quot -- x ) swap [ ] swap push \ \ push concat call ;
: dip ( x quot -- x ) reverse \ \ push swap push reverse call ;
```

Die alternativen Definitionen sind ebenso lesbar und einleuchtend wie die in der Prelude verwendete. Eine Messung der Laufzeiten könnte als Kriterium herangezogen werden, um die schnellste Lösung zu wählen. Da `dip` durchaus laufzeitkritisch ist – es spielt bei den nachfolgenden Kombinatoren eine entscheidende Rolle –, hat z.B. die konkatenative Sprache Factor `dip` im Kern seiner VM aufgenommen. Angenommen, `dip` sei als primitives Wort gegeben, dann sind `call` und `rot` definierbar als:

```
: call ( quot -- ... ) dup dip drop ;
: rot ( x y z -- y z x ) [ swap ] dip swap ;
```

Es hängt sehr davon ab, welche Wörter als primitiv angesehen und in einer VM implementiert werden, und welche Wörter dann in Folge „abgeleitete“ Wörter sind.

### 5.3.2 Erhaltungskombinatoren: `keep`

Die `keep`-Kombinatoren rufen die Quotierung auf dem Datastack wie ein `call` auf, sie bewahren (*keep*) jedoch nach dem Aufruf eine Reihe von Daten, die sich „vor“ der Quotierung und vor dem Aufruf auf dem Datastack befanden. Das Wort `keep` erhält ein Datum, `2keep` zwei Datenwerte und `3keep` drei Datenwerte.

```
>> : keep ( x quot -- x ) [ dup ] dip dip ;
>> : 2keep ( x y quot -- x y ) [ 2dup ] dip 2dip ;
>> : 3keep ( x y z quot -- x y z ) [ 3dup ] dip 3dip ;
```

Die Ausdrucksmittel für die Beschreibung der Stapeleffekte reichen nicht aus, um die Unterschiede zu den `dip`-Kombinatoren hervorzuheben. Die tatsächlichen Auswirkungen auf den Datastack hängen von dem Aufruf der Quotierung ab.

```
> clear 2 3 [ + ] 2keep
5 2 3
```

Beachten Sie, dass die `keep`-Kombinatoren wie die `dip`-Kombinatoren einem regulären Aufbau folgen – diesmal ohne jegliche Brüche. Eine alternative Implementierung ist:

```
: keep ( x quot -- x ) over [ call ] dip ; % see Factor
: 2keep ( x y quot -- x y ) 2over [ call ] 2dip ;
: 3keep ( x y z quot -- x y z ) 3over [ call ] 3dip ;
```

Beachten Sie, dass `3over` in der Prelude nicht definiert ist.

### 5.3.3 Cleave-Kombinatoren: `bi`, `tri` und `cleave`

Das Wort *cleave* heißt hier soviel wie „bewahren“, „festhalten“, „teilen“.

Die `bi`- und `tri`-Kombinatoren wenden zwei bzw. drei Quotierungen nacheinander auf den Datastack an und restaurieren ein, zwei oder drei Werte auf dem Datastack vor dem Aufruf der nächsten Quotierung. Im Stapeleffekt sind die Quotierungen mit `p`, `q` und `r` bezeichnet, die restaurierten Werte mit `x`, `y` und `z`.

```
>> : bi ( x p q -- ) [ keep ] dip call ;
>> : 2bi ( x y p q -- ) [ 2keep ] dip call ;
>> : 3bi ( x y z p q -- ) [ 3keep ] dip call ;

>> : tri ( x p q r -- ) [ [ keep ] dip keep ] dip call ;
>> : 2tri ( x y p q r -- ) [ [ 2keep ] dip 2keep ] dip call ;
>> : 3tri ( x y z p q r -- ) [ [ 3keep ] dip 3keep ] dip call ;
```

Ein paar wenige Beispiele mögen die Arbeitsweise von `bi`- bzw. `tri`-Kombinatoren veranschaulichen.

```
> clear 2 [ 1 + ] [ dup * ] bi
3 4
> [ + ] [ * ] 2bi
7 12
> clear 2 [ 1 + ] [ dup * ] [ 1 - ] tri
3 4 1
```

Der `cleave`-Kombinator verallgemeinert die `bi`- bzw. `tri`-Kombinatoren. Der `cleave`-Kombinator nimmt beliebige viele Quotierungen als Sequenz entgegen und wendet die Quotierungen nacheinander auf einen (`cleave`), auf zwei (`2cleave`) bzw. drei (`3cleave`) Werte auf dem Datastack an; vor jedem Aufruf werden die Werte restauriert. Das Wort `each` ist in Kap. 5.3.6 definiert.

```
>> : cleave ( x [ p q ... ] -- ) [ keep ] each drop ;
>> : 2cleave ( x y [ p q ... ] -- ) [ 2keep ] each 2drop ;
>> : 3cleave ( x y z [ p q ... ] -- ) [ 3keep ] each 3drop ;
```

Das folgende Beispiel ist identisch mit dem vorstehenden `tri`-Beispiel.

```
> clear 2 ( [ 1 + ] [ dup * ] [ 1 - ] ) cleave
3 4 1
```

### 5.3.4 Spread-Kombinatoren: `bi*` und `tri*`

Der `bi*`-Kombinator erwartet zwei Quotierungen (`p` und `q`), der `tri*`-Kombinator drei Quotierungen (`p`, `q` und `r`). Die Quotierungen verarbeiten im Fall von `bi*` und `tri*` jeweils nur einen Wert, bei `2bi*` und `2tri*` jeweils zwei Werte. Die Quotierungen werden auf die Verarbeitung der Stapelwerte verteilt – *spread* bedeutet soviel wie „verteilen“, „spreizen“.

Im Fall von `bi*` arbeitet `p` auf `x` und `q` auf `y`. Und im Fall von `2bi*` verarbeitet `p` die Werte `w` und `x` und `q` die Werte `y` und `z`.



```
>> : bi* ( x y p q -- ) [ dip ] dip call ;
>> : 2bi* ( w x y z p q -- ) [ 2dip ] dip call ;

> clear 2 3 [ 1 + ] [ dup * ] bi*
3 9
> clear 1 2 3 4 [ + ] [ * ] 2bi*
3 12
```

Die Kombinatoren `tri*` und `2tri*` arbeiten entsprechend.

```
>> : tri* ( x y z p q r -- ) [ 2dip ] 2dip bi* ;
>> : 2tri* ( u v w x y z p q r -- ) [ 4dip ] 2dip 2bi* ;

> clear 4 3 2 [ 1 + ] [ dup * ] [ 1 - ] tri*
5 9 1
> clear 6 5 4 3 2 1 [ + ] [ * ] [ - ] 2tri*
11 12 1
```

Die Verallgemeinerung der `bi*`- und `tri*`-Kombinatoren in Form eines `spread`- bzw. `2spread`-Wortes (analog zu `cleave` bzw. `2cleave`) wird derzeit nicht unterstützt.

### 5.3.5 Apply-Kombinatoren: `bi@` und `tri@`

Die Apply-Kombinatoren sind „Anwendungskombinatoren“ (*apply* heißt „anwenden“), die wie Spread-Kombinatoren arbeiten, im Gegensatz dazu jedoch nur eine Quotierung erwarten, die entsprechend dupliziert wird. Die Definitionen sind selbsterklärend.

```
>> : bi@ ( x y quot -- ) dup bi* ;
>> : 2bi@ ( w x y z quot -- ) dup 2bi* ;
>> : tri@ ( x y z quot -- ) dup dup tri* ;
>> : 2tri@ ( u v w x y z quot -- ) dup dup 2tri* ;

> clear 3 4 [ dup * ] bi@
9 16
> clear 6 5 4 3 2 1 [ * ] 2tri@
30 12 2
```

### 5.3.6 Sequenzkombinatoren: `each` und `Co`.

Sequenzkombinatoren wenden eine Quotierung `quot` auf jedes Element einer Sequenz `seq` an. Damit stehen Abstraktionen zur Verfügung, die dasselbe erreichen, wofür in anderen Programmiersprachen Schleifenkonstrukte wie „`for`“ und sogenannte Iteratoren zur Verfügung stehen. In einer funktionalen Sprache wie Consize geht man die Elemente einer Sequenz nicht per Index, sondern einfach der Reihe nach durch.

Der `each`-Kombinator ist der elementarste der Sequenzkombinatoren, er legt die Ergebnisse der Anwendung der Quotierung auf die einzelnen Elemente schlicht auf dem Datastack ab.

```
>> : each ( seq quot -- ... )
>>   swap dup empty?
>>   [ 2drop ]
>>   [ unpush -rot over [ call ] 2dip each ]
>>   if ;

> clear ( 1 2 3 4 ) [ dup * ] each
1 4 9 16
```

Das Wort **map** fasst die Ergebnisse wiederum in einer Sequenz zusammen.

```
>> : map ( seq quot -- seq' )
>>   [ push ] concat [ ] -rot each reverse ;
```

Die Definition von **map** erweitert das durch die Quotierung dargestellte Programm um ein **push**, das die einzelnen Ergebnis auf einen per **[ ]** und **-rot** bereit gestellten Stapel ablegt. Am Schluss stellt **reverse** die korrekte Reihenfolge her.

```
> clear ( 1 2 3 4 ) [ dup * ] map
[ 1 4 9 16 ]
```

Das Wort **reduce** aggregiert die Werte einer Sequenz zu einem Einzelwert bezüglich einer Operation, die durch eine Quotierung repräsentiert wird. Die Annahme ist zum einen, dass die Quotierung eine zweiwertige Operation ist, d.h. dass sie zwei Werte auf dem Datastack erwartet. Zum anderen ist **identity** das neutrale Element dieser Operation.

```
>> : reduce ( seq identity quot -- res ) [ swap ] dip each ;
```

Ein paar Beispiele: Das neutrale Element der Addition ist 0, das der Multiplikation 1 und das der Konkatenation **[ ]**.

```
> clear ( 1 4 9 16 ) 0 [ + ] reduce
30
> clear ( ) 0 [ + ] reduce
0
> clear ( 2 3 4 ) 1 [ * ] reduce
24
> clear ( [ 1 ] [ 2 ] [ 3 4 ] ) [ ] [ concat ] reduce
[ 1 2 3 4 ]
```

Die Wörter **map** und **reduce** sind ein besonderes Paar, da sie ein gewichtiges Prinzip verkörpern, das als Idee z.B. die Konzeption der Sprache **MapReduce** geprägt hat: Mit **map** kann eine Operation in Form einer Quotierung im Grunde parallel auf den einzelnen Daten einer Sequenz arbeiten, mit **reduce** werden die Einzelergebnisse zusammengetragen und ausgewertet. Nach diesem Prinzip verarbeitet Google mit MapReduce die riesigen Datenmengen, die bei der Erfassung von Webseiten und anderen Datenquellen anfallen. Die Berechnung verteilt Google auf Rechencluster von mehreren tausend Rechnern.

Das Wort **filter** nutzt die Quotierung als Prädikat, um nur die Elemente aus der Sequenz herauszufiltern, die den Prädikatstest bestehen. Eine Quotierung heißt Prädikat, wenn sie als Ergebnis ihrer Ausführung entweder ein *true* oder *false* in Form von **t** bzw. **f** auf dem Datastack ablegt.

```
>> : filter ( seq quot -- seq' ) % quot is a predicate
>>   [ dup ] swap concat
>>   [ [ push ] [ drop ] if ] concat
>>   [ ] -rot each reverse ;
```

Auch das Wort **filter** erweitert ähnlich wie **map** das Programm der Quotierung: ein vorangestelltes **dup** dupliziert die Werte zur Prüfung durch das Prädikat. Abhängig von dem Ergebnis der Prüfung wird der untersuchte Werte entweder auf dem Stapel abgelegt oder verworfen. Auch hier ist ein abschließendes **reverse** notwendig.

```
> clear ( 1 3 -4 5 0 7 2 ) [ 0 > ] filter
[ 1 3 5 7 2 ]
```

Das Wort `remove` macht das Gegenteil von `filter`: Es fasst die von `filter` verworfenen Elemente zusammen. Die Definition macht genau das, indem es das Prädikatsergebnis mit `not` negiert.

```
>> : remove ( seq quot -- seq' ) [ not ] concat filter ;

> clear ( 1 3 -4 5 0 7 2 ) [ 0 > ] remove
[ -4 0 ]
```

## 5.4 Erweiterung der Consizes-Grammatik

Consize hat eine derart einfache Grammatik, die jeden Programmtext erfolgreich in eine Folge von Wörtern zerlegen lässt. Leerräume im Programmtext grenzen die Wörter voneinander ab, siehe auch Kap. 3.9.

Die Großzügigkeit der Grammatik hat eine Schattenseite: Es fehlt die Möglichkeit, den Code syntaktisch zu strukturieren und Literale für Stapel und Mappings zu verwenden.

Als Literale bezeichnet man syntaktische Vereinbarungen, die eine Notation zur direkten Angabe eines Datentypen vorsehen. Zum Beispiel kodiert die Schreibweise `[ x y ]` einen Stapel, was einem die wenig leserliche Form der Stapelkonstruktion erspart:

```
emptystack \ y push \ x push
```

Kap. 5.4.1 führt Literale für zusammengesetzte Datentypen, sprich Stapel und Mappings ein.

Eine syntaktische Struktur bietet Ihnen etwa die aus diesem Kapitel bekannte Form, neue Wörter zu definieren. Nach dem einleitenden Doppelpunkt folgt das Wort, der optionale Stapeleffekt und die definierenden Wörter bis zum Semikolon. Damit befasst sich Kap. 5.4.2. In Kap. 5.4.3 wird eine weitere, kleine syntaktische Struktur zur Definition von Wörtern eingeführt, die als Symbole fungieren.

### 5.4.1 Literale für zusammengesetzte Datentypen

Eine Literal-Notation für zusammengesetzte Datentypen ist leicht in Consize integriert, wenn die Klammern eigenständige Wörter sind und nicht mit dem Wort verschmelzen. Die verschmelzende Schreibweise `[x y]` hat das Problem, die zwei Wörter `[x` und `y]` jeweils darauf hin untersuchen zu müssen, ob sie mit einem öffnenden Klammerzeichen beginnen oder einem schließenden Klammerzeichen enden. Dieser Analyse muss prinzipiell jedes Wort unterzogen werden. Ein unnötiger Aufwand, da es auch einfacher geht.

Die nicht-verschmelzende Notation `[ x y ]` lässt die Klammern eigenständige Wörter sein. Jetzt dient das Wort `[` als Auslöser, um die Suche nach dem schließenden Klammerwort zu starten und aus den Elementen zwischen den Klammer-Wörtern den Stapel zu konstruieren. Sobald eine öffnende Klammer die Suche triggert, wird das Klammer-Wort auf dem Datastack abgelegt und die schließende Klammer in der Continuation gesucht.

```
>> : [ ( -- quot ) \ [ [ scan4] continue ] call/cc ;
>> : ( ( -- seq ) \ ( [ scan4] continue ] call/cc ;
```

```
>> : { ( -- map ) \ { [ scan4] continue ] call/cc ;
```

Das Wort `scan4`] sucht nach `]`, `}` bzw. `)` in der aktuellen Continuation und liefert im Fall von eckigen und runden Klammern einen Stapel und bei geschweiften Klammern ein Mapping zurück. Worin unterscheiden sich die Notationen im Detail?

Die eckigen Klammern repräsentieren eine Quotierung und übernehmen alle Wörter zwischen den Klammern so wie sie sind. Runde Klammern erlauben die Ausführung von Wörtern zwischen der öffnenden und der schließenden runden Klammer.

```
> clear [ 1 dup 1 + dup 1 + ]
[ 1 dup 1 + dup 1 + ]
> clear ( 1 dup 1 + dup 1 + )
[ 1 2 3 ]
```

Die geschweiften Klammern erzeugen ein Mapping, die – wie runde Klammern – die Wörter zwischen den Klammern ausführen und dann erst das Mapping bilden.

```
> clear { 1 dup 1 + 3 4 }
{ 1 2 3 4 }
```

Natürlich werden Verschachtlungen von Klammern vom Wort `scan4`] berücksichtigt. Ebenso beachtet `scan4`] die Wirkung des Wortes `\` als Escape-Wort. Erreicht `scan4`] das Ende des Callstacks, ohne dass die Klammern aufgelöst werden konnten, fehlt irgendwo zu einer öffnenden Klammer ihr schließendes Pendant, was eine Fehlermeldung nach sich zieht.

```
>> : scan4] ( ds cs -- ds' cs' )
>>   unpush dup
>>   { \ ]   [ drop [ ] rot scan4] ]
>>     \ }   over
>>     \ )   over
>>     \ [   [ rot swapu swap scan4] scan4] ]
>>     \ {   over
>>     \ (   over
>>     \ \   [ drop unpush rot swapu \ \ push swap scan4] ]
>>     :else [ rot swapu swap scan4] ]
>>     nil  [ \ syntax-error [ unbalanced brackets ] _|_ ]
>>   } case ;
```

Das Wort `over` realisiert innerhalb der geschweiften Klammern zum `case`-Wort einen kleinen [Hack](#). Code-Dopplungen werden per `over` einfach in die nächste Zeile übertragen.

Ist eine schließende Klammer gefunden, wird von hinten her das Feld „aufgeräumt“: Die Elemente werden durch `scan4`] vom Ende her per `push` solange in einen Stapel befördert, bis die öffnende Klammer gefunden ist. Die öffnende Klammer bestimmt, was mit den aufgesammelten Elementen passiert: Werden die Elemente unverändert gelassen (Quotierung mit `[]`), werden sie innerhalb der Klammern in einem eigenen Kontext mit `fcall` aufgerufen (so der Fall bei `(` und `{`), wird aus dem Ergebnis ein Mapping gemacht (was bei `{` der Fall ist)?

```
>> : scan4] ( cs' stk ds' -- ds'' cs'' )
>>   unpush dup
```

```
>> { \ [ [ drop swap                push swap ]
>>   \ { [ drop swap fcall mapping push swap ]
>>   \ ( [ drop swap fcall          push swap ]
>>   \ \ [ drop unpush rot swapu \ \ push swap scan4[ ]
>>   :else [ rot swapu swap scan4[ ]
>>   nil   [ \ syntax-error [ unbalanced brackets ] _|_ ]
>> } case ;
```

Consize verlangt immer eine ausgewogene Anzahl an öffnenden und schließenden Klammern, nicht jedoch, dass die schließende Klammer zur öffnenden passt. Auch wenn das höchst verwirrend zu lesen ist (und daher niemals so geschrieben werden sollte), das folgende Beispiel liefert ein Mapping, obwohl die schließende Klammer nicht passt.

```
> clear { 1 dup 2 3 }
{ 1 1 2 3 }
```

Sollen die Elemente für ein Mapping nicht ausgeführt werden, so ist mit einer Quotierung und `mapping` zu arbeiten:

```
> clear { 1 dup 2 3 }
{ 1 1 2 3 }
> clear [ 1 dup 2 3 ] mapping
{ 1 dup 2 3 }
```

Mit `parse-quot` werden die innerhalb einer Quotierung verwendeten Literal-Notationen in die entsprechenden Datentypen umgewandelt.

```
>> : parse-quot ( quot -- quot' )
>> \ [ push reverse \ ] push reverse call ;
```

Die Folge der Wörter `[`, `1`, `2`, `3` und `]` sieht zwar aus wie die Notation eines Stapels, es sind aber tatsächlich nur fünf Wörter, wie sich durch `top` im Beispiel zeigt.

```
> clear ( \ [ 1 2 3 \ ] )
[ [ 1 2 3 ] ]
> top
[
```

Nach der Anwendung von `parse-quot` ist die Wortfolge in einen Stapel verwandelt worden. Die Intention der Notation als Literal muss per `parse-quot` ausdrücklich eingefordert werden.

```
> clear ( \ [ 1 2 3 \ ] )
[ [ 1 2 3 ] ]
> parse-quot
[ [ 1 2 3 ] ]
> top
[ 1 2 3 ]
```

#### 5.4.2 Neue Wörter mit `:` bis `;` definieren

Ohne eine spezielle Syntax zur Definition neuer Wörter bedarf es mindestens des Wortes `def`, um ein neues Wort mit einem beliebigen Datum im Wörterbuch eintragen zu können.

```
>> : def ( wrd itm -- ) swap get-dict assoc set-dict ;
```

Für die Ablage aller Stapeleffekte gibt es das Wort `*effect*`. Das zu `*effect*` abgelegte Mapping speichert Wörter und die mit ihnen verbundenen Stapeleffekte.

```
>> : *effect* ( -- ) { } ;
```

Das Wort `def+` ist ein um die Speicherung des Stapeleffekts erweitertes `def`.

```
>> : def+ ( wrd [ effect ] [ body ] -- )
>>   rot dup rot def *effect* assoc [ ] swapu \ *effect* swap def ;
```

Das Wort `:` leitet die Definition eines neuen Wortes unter einer übersichtlicheren Syntax ein. In der aktuellen Continuation wird mit `scan4`; nach dem Wort `;` gesucht, das das Ende der Definition markiert. Die Definition wird in ihre Anteile zerlegt und zur Verarbeitung durch `def+` aufbereitet.

```
>> : : ( | ... ' ; -- quot )
>>   [ [ ] swap scan4; destruct-definition def+ continue ] call/cc ;
```

Das Wort `scan4`; arbeitet wie ein `scan4]` oder `scan4[`. Kleine Unterschiede im Stapeleffekt erzeugen leichte Variationen im Aufbau des Programmtextes.

```
>> : scan4; ( ds [ ] cs -- ds cs' quot )
>>   unpush dup
>>   { \ ;   [ drop swap reverse ]
>>     \ \   [ drop unpush rot \ \ push swapu swap scan4; ]
>>     :else [ rot swapu swap scan4; ]
>>     nil   [ \ syntax-error [ incomplete definition ] _|_ ]
>>   } case ;
```

Das Wort `destruct-definition` zerlegt die zwischen `:` und `;` gesammelten Wörter in das zu definierende Wort, den Stapeleffekt und den „Rest“, den eigentlichen Rumpf der Wortdefinition. Per `parse-quot` werden Literal-Notationen in die entsprechenden Datentypen umgewandelt. Die Umwandlung erfolgt damit zur Definitionszeit eines Worts und nicht zur Laufzeit – was in einer frühen Version von Consize der Fall war und beträchtliche Laufzeiteinbußen mit sich brachte.

```
>> : destruct-definition ( quot -- wrd stackeffect body )
>>   dup top swap pop      % extract word
>>   [ ] swap              % prepare extraction of stack effect
>>   dup top \ ( equal?    % extract stack effect
>>     [ pop look4) ] when % if given
>>   parse-quot ;          % and parse quotation
```

Auch das Wort `look4)` unterscheidet sich in seiner Arbeitsweise im Grunde nicht von `scan4]`, `scan4[` und `scan4`;

```
>> : look4) ( [ ... ] quot -- [ ... ]' quot' )
>>   unpush dup
>>   { \ )   [ drop swap reverse swap ]
>>     \ \   [ drop unpush rot swapu swap look4) ]
>>     :else [ rot swapu swap look4) ]
>>     nil   [ \ syntax-error [ incomplete stack effect ] _|_ ]
>>   } case ;
```

### 5.4.3 Symbole

In Consize erwartet jedes Wort, die zu verarbeitenden Daten auf dem Datastack vorzufinden. Bis auf die Wörter `call/cc` und `get-dict` gilt das ausnahmslos für alle Wörter der Consize-VM und für die Wörter der Prelude. Das drückt sich auch in der Postfix-Notation aus; die auf den Daten operierenden Wörter kommen immer *nach* den Daten.

In der Prelude gibt es wenige Ausnahmen von dieser Regel – und dazu gehören all die Wörter dieses Kapitels, die die Grammatik von Consize erweitern: `[`, `(`, `{`, `:` und das noch zu besprechende `SYMBOL:`. Wir klassifizieren diese Wörter als „syntaktische Wörter“. Syntaktische Wörter benötigen Daten, die nicht auf dem Datastack, sondern auf dem Callstack zu finden sind. Das ist auf den ersten Blick nicht verträglich mit dem konkatenativen Paradigma.

Die Ordnung wird dadurch wiederhergestellt, weil diese Wörter mit Hilfe von `call/cc` nur einen definierten Teil der Daten vom Callstack abgreifen, die prinzipiell hätten auch auf dem Datastack stehen können, wozu es dann eines entsprechenden „normalen“ Wortes bedürfte. Ganz deutlich wird das am Beispiel des Wortes `:`, das eine Wortdefinition einleitet. Die Daten vom Callstack werden aufbereitet zur Verarbeitung durch `def+`. Das syntaktische Wort `:` hat in `def+` seinen „normalen“, den Datastack verarbeitenden Gegenpart. Bei `[`, `(` und `{` ist es sogar noch einfacher. Diese Wörter repräsentieren zusammen mit ihren schließenden Gegenstücken ein einziges Datum auf dem Datastack.

Syntaktische Wörter haben also einzig die Funktion, aus der „normalen“ Postfix-Notation auszubrechen, um eine syntaktische Struktur anzubieten, von der man sich Vorteile in der Wahrnehmung durch den Programmierer bzw. die Programmiererin verspricht. Auf gut deutsch: Der Programmtext soll lesbarer werden. Lesbarkeit und syntaktische Spieleereien stehen nicht im Widerspruch zum konkatenativen Paradigma, sie erfordern nur die Auflösung der syntaktischen Form in „normale“ konkatenative Strukturen.

Ganz in diesem Sinne arbeitet das syntaktische Wort `SYMBOL:`. Es erwartet genau ein Wort oben auf dem Callstack und baut den Code für die folgende Definitionsstruktur auf. Aus

```
SYMBOL: <word>
```

```
wird
```

```
: <word> \ <word> ;
```

Die Definition (ohne Stapeleffekt) sorgt dafür, dass das Wort sich selbst als Datum auf dem Datastack ablegt. Und genau das versteht Consize unter einem Symbol. Symbole sind Wörter, die sich selbst als Datum repräsentieren.

```
>> : SYMBOL: ( | itm -- )
>>   [ unpush dup ( \ ; ) swapu \ \ push swapu \ : push
>>     swap concat continue ] call/cc ;
```

Jeder Gebrauch von `call/cc`, der nicht zur Umsetzung eines syntaktischen Wortes dient, muss ernsthaft die Verträglichkeit mit dem konkatenativen Paradigma hinterfragen. Das Wort `call/cc` macht die Tür zur Manipulation der aktuellen Continuation weiter auf als es für das konkatenative Paradigma notwendig ist. Tatsächlich stehen Continuations

in jüngerer Zeit in der Kritik, bei „unsachgemäßem“ Gebrauch unverträglich mit funktionalen Sprachen zu sein (Consize ist eine funktionale Sprache), was weitreichende und ungewollte Konsequenzen haben kann.<sup>1</sup> Als Lösung gelten begrenzte Continuations (*delimited continuations*, die auch unter dem Namen *partial continuations* firmieren), die einen unsachgemäßen Gebrauch erst gar nicht ermöglichen.

## 5.5 Die Interaktion mit Consize

### 5.5.1 Repräsentation von Daten und ihre Ausgabe

Die Wörter `<space>` und `<newline>` sind nützliche Helferlein. Beide Wörter erleichtern ein wenig die Konstruktion von Ausgaben über die Konsole.

```
>> : <space> ( -- space ) \space char ;
>> : <newline> ( -- newline ) \newline char ;
```

Das Wort `println` steht für *print line*; eine Abstraktion, die die Ausgabe auf der Konsole mit einem Zeilenumbruch enden lässt.

```
>> : println ( -- ) print <newline> print flush ;
```

Die Consize-VM hat keinen Zugriff darauf, wie Daten innerhalb der VM gespeichert und kodiert sind. Es ist nicht einmal festgelegt, wie Stapel, Mappings, Funktionen und „Nichts“ ausgegeben werden sollen. Nur für Wörter ist durch `print` in der Consize-VM eine Repräsentation gegeben, die dem Wortnamen entspricht.

Für die Repräsentation der verschiedensten Datentypen zeichnet sich das Wort `repr` verantwortlich. Hier wird festgelegt, wie die Werte der Datentypen dargestellt (repräsentiert) werden sollen. Aus naheliegenden Gründen werden die Literal-Notationen für Stapel und Mappings verwendet wie Sie sie bereits aus Kap. 2 kennen. Nicht zuletzt erlaubt die Symmetrie von Repräsentation und Literal-Notation, dass eine Repräsentation eines Wertes grundsätzlich auch wieder von Consize eingelesen werden kann. Nur für Funktionen gibt es keine sinnvolle Repräsentation, weshalb einheitlich die Darstellung `<fct>` gewählt wird.

Da Consize nur für Wörter eine Repräsentation kennt, müssen die Repräsentationen für die zusammengesetzten Datentypen aus Einzelwörtern mit `word` zu einem Repräsentationswort zusammengesetzt werden. Selbstverständlich ist der Prozess rekursiv.

```
>> : repr ( itm -- wrd )
>>   dup type
>>   { \ wrd [ ]
>>     \ stk [ [ ] \ [ push <space> push swap
>>               [ repr push <space> push ] each
>>               \ ] push reverse word ]
>>     \ map [ unmap [ ] \ { push <space> push swap
>>               [ repr push <space> push ] each
>>               \ } push reverse word ]
>>     \ fct [ drop \ <fct> ]
>>     \ nil [ drop \ nil ]
>>     :else [ \ repr-error [ unknown type ] _|_ ]
>>   } case ;
```

<sup>1</sup> <http://lambda-the-ultimate.org/node/4586> bietet einen guten Einstieg in die Diskussion.



Die Repräsentation eines Stapels hat immer eine öffnende und eine schließende Klammer. Sollen die äußeren eckigen Klammern samt der Leerräume entfernt werden, so hilft `unbracket-stk-repr`. Ein leerer Stapel wird nach Entfernung der Klammern auf ein Leerzeichen reduziert.

```
>> : unbracket-stk-repr ( wrd -- wrd' ) % '[ ... ]' => '...'
>> unword
>> pop pop reverse pop pop reverse
>> dup empty? [ <space> push ] when
>> word ;
```

In der Einführung zu Consize, genauer in Kap. 2.6, haben wir für die Darstellung einer Continuation, also der gleichzeitigen Darstellung von Data- und Callstack eine Notation vereinbart, in der ein senkrechter Strich „|“ beide Stapel voneinander trennt. Das Wort `printcc` gibt eine Continuation genau so auf der Konsole aus.

```
>> : printcc ( ds cs -- ds cs )
>> swap dup reverse repr unbracket-stk-repr print
>> <space> print \ | print <space> print
>> swap dup repr unbracket-stk-repr println ;

> clear [ 2 1 ] [ swap drop ] printcc
1 2 | swap drop
[ 2 1 ] [ swap drop ]
```

Beachten Sie: Bei der Stapelnotation mit eckigen Klammern liegt das offene Stapelende immer links. Bei der Ausgabe der Continuation ist der Datastack mit seinem offenen Stapelende zum senkrechten Strich hin orientiert. Die Umkehrung in der Darstellung des Datastacks erledigt ein `reverse` in `printcc`.

### 5.5.2 Die REPL: Die Interaktion über die Konsole

[Lisp](#) gilt als die zweitälteste Programmiersprache nach Fortran. Mit Lisp kam die [Read-Evaluate-Print-Loop](#) in die Welt. Die REPL beschreibt die Interaktion des Programmierers bzw. der Programmiererin mit Lisp über die Konsole: Die Eingabe wird eingelesen (*read*), der eingegebene Ausdruck ausgewertet (*evaluate*) und das Ergebnis der Auswertung auf der Konsole ausgegeben (*print*). Dieses Schema wiederholt sich wieder und wieder – das ist die Schleife (*loop*) der Interaktion.

Die REPL beschreibt das universelle Ablaufschema aller interaktiven Sprachen, zu denen Python, Ruby, Perl und eben auch Consize gehören.

Der `reader` macht über das „>“-Zeichen (samt Leerzeichen) auf sich aufmerksam, nimmt per `read-line` eine Eingabe entgegen, entfernt aus der Eingabe per `uncomment` die Kommentare und aktiviert den Tokenizer mit `tokenize`. Das Resultat ist eine Quotierung auf dem Datastack. Damit `uncomment` funktioniert, muss ein `\newline char` zur eingelesenen Eingabezeile hinzugefügt werden.

```
>> : reader ( -- quot )
>> \ > print <space> print flush read-line
>> [ ] \newline char push swapu word
>> uncomment tokenize ;
```

Der `evaluator` ist gleichzusetzen mit `call` in Consize.

```
>> : evaluator call ;
```

Der `printer` gibt das Ergebnis der Evaluation, den Datastack, auf der Konsole aus.

```
>> : printer ( -- )
>>   get-ds reverse repr unbracket-stk-repr println ;
```

Die `repl` ist die sich wiederholende Abfolge von `reader`, `evaluator` und `printer`.

```
>> : repl reader evaluator printer repl ;
```

So einfach ist es, eine Programmiersprache mit einer REPL zu versehen! Allerdings muss eine interaktive Programmiersprache eine Voraussetzung erfüllen: Kurze syntaktische Einheiten müssen gültige Programme darstellen und inkrementell den Programmzustand verändern können. Beide Belange erfüllt Consize. Traditionelle Compilersprachen wie Java, C, C++ und C# sind allein schon von ihrer Grammatik nicht darauf ausgelegt, „Kleinstprogramme“ zuzulassen.

### 5.5.3 Dateien lesen und starten

Das Laden eines Consize-Programms erfordert nach dem Einlesen (`slurp`) die Entfernung aller Kommentare (`uncomment`) und das auf das Tokenizing reduzierte Parsen des Quelltextes. Das Wort `run` ruft das geladene Programm auf.

Mit `lload` und `lrun` gibt es die entsprechenden Wörter, wenn das Consize-Programm als „literarisches Programm“ (*literate program*, siehe Kap. 3.9), vorliegt.

```
>> : load ( wrd -- quot ) slurp uncomment tokenize ;
>> : lload ( wrd -- quot ) slurp undocument uncomment tokenize ;
>> : run ( wrd -- ... ) load call ;
>> : lrun ( wrd -- ... ) lload call ;
```

Die folgenden drei Wörter sind nicht viel mehr als Abkürzungen, die bei der (Weiter-)Entwicklung der Prelude von Nutzen sind.

```
>> : prelude ( -- ... ) \ prelude.txt run ;
>> : test-prelude ( -- ... ) \ prelude.tests.txt run ;
>> : pprelude ( -- ... ) \ prelude-plain.txt run ;
```

### 5.5.4 Reflexion und Debugging

Consize ist ein [reflexives Programmiersystem](#), d.h. es kann auf seinen aktuellen Programmzustand zugreifen und ihn verändern; manchmal spricht man auch von [Introspektion](#). Die Fähigkeit dazu ist in der Consize-VM in dem Wort `stepcc` angelegt. Der Programmzustand ist gegeben durch das Wörterbuch, den Data- und den Callstack.

Die folgenden Wörter vereinfachen den Zugriff auf den aktuellen Programmzustand: `lookup` schlägt die mit einem Wort im Wörterbuch assoziierte Quotierung nach, `source` gibt sie auf der Konsole aus. Das Wort `get-ds` legt den Datastack auf dem Datastack ab (was seltsam klingen mag, aber funktioniert), `set-ds` verändert den Datastack und mit `clear` kann der Datastack „gelöscht“ werden.

```
>> : lookup ( word -- item ) get-dict \ word-not-found get ;
>> : source ( word -- ) lookup repr println ;
```

```
>> : get-ds ( -- stk ) [ swap dup push swap continue ] call/cc ;
>> : set-ds ( stk -- ) [ swap top swap continue ] call/cc ;
>> : clear ( -- ) [ ] set-ds ;
```

Das Wort **abort** unterbricht die Programmausführung und bietet die REPL zur Interaktion an. Das Wort **exit** beendet die Arbeit der Consize-VM. Sobald der Callstack gänzlich abgearbeitet, d.h. leer ist, hat die Consize-VM ihre Aufgabe erfüllt.

```
>> : abort ( -- ) [ drop [ printer repl ] continue ] call/cc ;
>> : exit ( -- ) [ drop [ ] continue ] call/cc ;
```

In den [Urzeiten der Computerei](#) bestanden die Rechner teils aus elektromechanischen [Relais](#) und teils aus rein elektronischen Bauteilen wie [Röhren](#). Sie dürfen sich diese Bauteile in den Größendimensionen eines Daumens und größer vorstellen. Die damaligen Rechner beanspruchten mit ihren mehreren Tausend Bauteilen nicht nur den Platz ganzer Schränke, sondern den ganzen Räume. So soll einst ein Käfer, zu Englisch *bug*, eine Fehlfunktion ausgelöst haben, als er in den Wirrungen des elektronischen Allerleis herumkrabbelte, einen Kurzschluss auslöste und sein Leben ließ. Seit diesem historisch dokumentierten Ereignis werden auch die [Programmfehler](#) in der Software als „Bugs“ bezeichnet.

Seinerzeit gestaltete sich die Suche nach einem verkohlten Käfer in den unendlichen Weiten der Hardware als Herausforderung. Mit der Erfindung des [Integrierten Schaltkreises](#) (*integrated circuit*, IC) und der darauf folgenden drastischen Miniaturisierung der Hardware, gehört diese Art der Fehlersuche der Vergangenheit an. Doch nicht minder schwer und aufwendig ist bisweilen die Suche nach den meist unfreiwillig eingebauten „Bugs“ in der Software.

Die Suche nach dem Softwarefehler nennt man „Debugging“ und das dazu verwendete Hilfsmittel als „[Debugger](#)“. Eine zentrale Funktion eines Debuggers ist die Einzelschritt-Ausführung eines Programm ab einem definierten [Haltepunkt](#), dem *breakpoint*. So kann man sozusagen in Zeitlupe, Schritt für Schritt verfolgen, was das Programm tut und zwischen den Einzelschritten den aktuellen Programmzustand inspizieren. All das soll helfen, einen Softwarefehler systematisch einzukreisen und zu entdecken.

Das Wort **break** definiert einen Haltepunkt. Das Wort legt die aktuelle Continuation auf dem Datastack ab und bietet dazu den gewohnten interaktiven Dialog an.

```
>> : break ( -- ds cs ) [ printer repl ] call/cc ;
```

Das Wort **step** führt einen Rechenschritt mit der auf dem Datastack befindlichen Continuation aus.

```
>> : step ( ds cs -- ds' cs' )
>>   dup empty? [ get-dict -rot stepcc rot drop ] unless ;
```

Um das oberste Wort auf dem Callstack der Continuation komplett auszuführen, dazu dient **bstep** (*big step*). Das Wort funktioniert nur, wenn die weiteren Wörter auf dem Callstack keine Rolle spielen – was der Normalfall ist.

```
>> : bstep ( ds cs -- ds' cs' )
>>   unpush dup
>>   { \ \ [ drop unpush [ ] swapu \ \ push ]
>>     :else [ [ ] swapu ]
```

```
>> nil [ drop [ ] ]
>> } case get-dict func swap [ apply ] dip ;
```

Ersetzt man im Beispiel `bstep` durch `step`, tritt der Unterschied in der Auflösung der Programmausführung deutlich zutage.

```
> clear [ 2 3 ] [ \ 4 -rot + ]
[ 2 3 ] [ \ 4 -rot + ]
> bstep
[ 4 2 3 ] [ -rot + ]
> bstep
[ 2 3 4 ] [ + ]
> bstep
[ 5 4 ] [ ]
> bstep
[ 5 4 ] [ ]
```

### 5.5.5 Unit-Testing

Der [Softwaretest](#) ist eine qualitätssichernde Maßnahme und gerade auf der Modulebene so wichtig, dass eine Unterstützung zur Formulierung und Ausführung von [Modultests](#) (*unit tests*) nicht fehlen soll. Für praktisch jede Programmiersprache wird für diesen Zweck eine [Unittest-Software](#) bereitgestellt, die das Testen vereinfacht. In Consize ist so etwas in zwei Zeilen Code programmiert, wenn man auf die Ausgabe auf der Konsole verzichtet. Mit Konsolen-Ausgaben braucht es ein paar Zeilen mehr.

```
>> : unit-test ( result quot -- )
>> [ \ test print [ <space> print repr print ] bi@ ] 2keep
>> [ fcall equal? ] 2keep
>> rot
>> [ <space> print \ passed println 2drop ]
>> [ <space> print \ failed println \ with print <space> print
>> nip fcall repr println abort ]
>> if ;
```

Ein Unittest erwartet den zu antizipierten Inhalt des Datastacks (`result`) nach Abarbeitung des in `quot` abgelegten Programms. Produziert der Aufruf von `quot` nicht das mit `result` vorgegebene Resultat, bricht die Ausführung ab und das erwartete Ergebnis wird zusammen mit der Abbruchmeldung ausgegeben. Die Angabe des erwarteten Ergebnisses hilft, den Fehlschlag des Tests besser zu verstehen.

```
> ( 5 ) [ 2 3 + ] unit-test
test [ 5 ] [ 2 3 + ] passed

> ( 7 ) [ 2 3 + ] unit-test
test [ 7 ] [ 2 3 + ] failed
with [ 5 ]
```

## 5.6 Serialisierung, Consize-Dumps und Bootstrapping

### 5.6.1 Die Serialisierung von Daten

Mit der [Serialisierung](#) eines Datums ist eine Beschreibungsform gemeint, die zur sequentiellen Ablage auf einem Datenträger zwecks Speicherung geeignet ist und eine vollständige Rekonstruktion des Datums erlaubt.

Consize serialisiert Daten so, wie Sie es in der Einleitung zu Consize in Kap. 2.5 (S.14ff.) kennengelernt haben. Das Wort `serialize` automatisiert den Vorgang, ein Datum einzig mit den Wörtern der Consize-VM in eine Wortfolge zur Erzeugung des Datums zu übersetzen.

```
>> : serialize ( quot -- quot' )
>>   get-ds [ clear ] dip
>>   dup top swap pop
>>   [ -serialize- get-ds ] dip
>>   swap reverse push set-ds ;

>> : -serialize- ( item -- stream-of-items )
>>   dup type
>>   { \ wrd [ \ \ swap ]
>>     \ stk [ \ emptystack swap reverse [ -serialize- \ push ] each ]
>>     \ map [ unmap -serialize- \ mapping ]
>>     \ nil [ drop \ emptystack \ top ]
>>     :else [ \ serialization-error [ invalid type ] _|_ ]
>>   } case ;
```

Mit `call` können serialisierte Daten wieder rekonstruiert werden.

```
> clear [ 1 hello 2 ] serialize
[ emptystack \ 2 push \ hello push \ 1 push ]
> call
[ 1 hello 2 ]
```

Die Wörter `remove-fct-assocs` und `keep-fct-assocs` sind zwei Hilfsörter: Sie entfernen (*remove*) oder behalten (*keep*) die Einträge in einem Wörterbuch, die mit einer Funktion assoziiert sind. Mit diesen Wörtern wird das Erzeugen eines Dumps vereinfacht!

```
>> : remove-fct-assocs ( dict -- dict' )
>>   dup keys
>>   [ 2dup swap \ _|_ get
>>     type \ fct equal?
>>     [ swap dissoc ]
>>     [ drop ]
>>   ] if ] each ;

>> : keep-fct-assocs ( dict -- dict' )
>>   dup keys
>>   [ 2dup swap \ _|_ get
>>     type \ fct equal?
>>     [ drop ]
>>     [ swap dissoc ]
>>   ] if ] each ;
```

### 5.6.2 Der Dump: Ein Schnappschuss des Wörterbuchs

Der `dump` erzeugt eine speicherfähige Kopie des Gedächtnisses von Consize einschließlich der übergebenen Quotierung. Das zentrale Gedächtnis von Consize ist das Wörterbuch.

Zur Erzeugung des Dumps müssen alle atomaren Wörter der Consize-VM aus dem Wörterbuch entfernt werden, da die mit ihnen assoziierten Funktionen nicht serialisierbar sind – das ist auch gar nicht nötig, da die grundlegenden Wörter der Consize-VM und ihre Funktionen ohnehin zur

Verfügung stehen. Die mit `dump` übergebene Quotierung dient als Start-routine, so dass ein Dump, der z.B. von einem Speichermedium eingelesen wird, sich selbst aktivieren und starten kann.

```
>> : dump ( quot -- quot' )
>> [ ] get-dict remove-fct-assocs push serialize
>> swap serialize concat
>> [ continue ] concat ;
```

Mit `create-dump` wird ein Dump unter dem angegebenen Namen im Dateisystem abgelegt, der von der Consize-VM gestartet werden kann und sich nach dem Start mit `say-hi` und `repl` über die Konsole meldet. Seien Sie übrigens ein wenig geduldig, das Erzeugen eines Dumps nimmt etwas Zeit in Anspruch.

```
>> : create-dump ( filename -- )
>> [ get-dict swap merge set-dict say-hi repl ] dump
>> repr unbracket-stk-repr swap spit ;
```

Wozu ist ein Dump von Consize überhaupt von Nutzen?

Die Version der Prelude, die Sie aktuell lesen, ist die Prelude im Klartext. Diese Fassung der Prelude verwendet die in Kap. 5.4 vorgestellten Erweiterungen für eine programmierfreundliche Darstellung des Quelltextes. Mit dieser Darstellung geht nur ein kleines Problem einher: Das Einlesen der Prelude dauert ein wenig, da das inkrementelle Definieren von Wörtern und das Auflösen syntaktischer Kodierungen etwas Zeit benötigt. Viel effizienter ist es, einen mit `create-dump` erzeugten Dump der Prelude einzulesen. Darum startet die Consize-VM in aller Regel mit einem Dump der Prelude.

Sie können also Consize sowohl mit dem Dump der Prelude oder mit der Klartext-Prelude starten. Da Consize beim Start generell die Datei `prelude.txt` erwartet, ist entweder die Dump- oder die Klartext-Datei entsprechend umzubenennen. In der Regel ist die Datei `prelude.txt` die Dump-Version und `prelude-plain.txt` die Prelude im Klartext. Die Datei `prelude.txt` kann man auch mit dem Wort `prelude` laden und starten. Analog lädt und startet `pprelude` die Datei `prelude-plain.txt`, siehe Kap. 5.5.3.

### 5.6.3 Bootstrapping Consize

Um Consize mit einem Dump der Prelude starten zu können, muss der Dump erst einmal erzeugt werden. Dazu ist es notwendig, Consize mit der Prelude im Klartext zu starten. Die Klartext-Fassung der Prelude kann von der Consize-VM nicht so ohne weiteres verarbeitet werden. Schließlich verwendet die Prelude Erweiterungen (siehe Kap. 5.4), die Consize zu Beginn nicht kennt.

Das ist der Grund, warum die Prelude mit dem Laden des sogenannten Bootimage beginnt (siehe Kap. 5.1.2). Das Bootimage ist ein Programm, das all die Wörter definiert, die notwendig sind, um die Verarbeitung der Prelude mit ihren syntaktischen Erweiterungen vorzubereiten. Das Bootimage muss dazu freilich selber auf diese Erweiterungen verzichten, weshalb es nicht sonderlich leserlich daherkommt.

Ursprünglich habe ich das Bootimage per Hand geschrieben, und anfangs gab es auch keine gesonderte Datei mit dem Bootimage, die notwendigen Definitionen waren direkter Bestandteil der Prelude. Inzwischen nimmt

mir das Wort `bootimage` die Handarbeit ab. Es produziert automatisch den Code für die einleitenden Definitionen und speichert sie in der Datei `bootimage.txt` ab.

```
>> : bootimage ( -- )
>>   bootstrap repr unbracket-stk-repr \ bootimage.txt spit ;
```

Dieser Automatismus verschleiert ein wenig, wie das Bootimage historisch entstand, welche „Handarbeit“ nötig war, um die Verarbeitung der Prelude mit den Literalen für Stapel und Mappings und mit der Notation für Definitionen zu ermöglichen. Um den Weg nachvollziehbar zu machen, sei hier der Ausgangspunkt des Bootimages, die Entstehung des Wortes `def` ausführlich beschrieben.

Das Wort `def` ist essentiell. Es steht im Brennpunkt aller weiterer Aktivitäten, die immer darauf abzielen, neue Wörter dem Wörterbuch hinzuzufügen. Aus Kap. 5.4.2 kennen Sie die Definition:

```
: def ( wrd itm -- ) swap get-dict assoc set-dict ;
```

Zu Beginn steht uns diese Schreibweise in Consize nicht zur Verfügung. Beim Start von Consize sind ausschließlich die Wörter der Consize-VM verfügbar. Wir müssen uns den gewünschten Programmierkomfort Schritt für Schritt „erarbeiten“.

Angenommen, das Wort `def` wäre in Consize bereits definiert, dann könnten wir `def` immerhin mit sich selbst definieren.

```
\ def [ swap get-dict assoc set-dict ] def
```

Aber wie kann ein Wort sich selbst definieren ohne bereits definiert zu sein? Natürlich geht das nicht. Aber der Zirkelschluss, dass sich ein Wort selbst definiert, lässt sich händisch auflösen. Informatiker nennen diese Technik „[Bootstrapping](#)“. Wir ersetzen einfach `def` mit seiner eigenen Definition!

```
\ def [ swap get-dict assoc set-dict ] swap get-dict assoc set-dict
```

Noch steht uns allerdings eines im Weg. Am Anfang der Prelude können wir nicht die Notation mit den eckigen Klammern nutzen. Wir müssen die Quotierung ebenfalls händisch aufbauen.

```
\ def
emptystack \ set-dict push \ assoc push \ get-dict push \ swap push
swap get-dict assoc set-dict
```

Das ist exakt der Code, den Consize in den ersten zwei Zeilen des Wortes `bootstrap` erzeugt, um ein Bootimage zu erstellen.

```
>> : bootstrap ( -- quot )
>>   [ \ def ] \ def lookup serialize concat
>>     \ def lookup concat
>>   [ swapu *effect* def+ unpush -rot over
>>     SYMBOL: case when if choose call t f false fcall lookup
>>     scan4] scan4[ parse-quot destruct-definition
>>     : scan4; look4)
>>     read-word read-mapping ]
>>   ( \ [ \ ( \ { ) concat
>>   [ bootstrap-word concat ] each ;
```

Probieren Sie es an der Konsole aus! Ich habe mir hier aus Platzgründen lediglich erlaubt, die Ausgabe an der Konsole durch Zeilenumbrüche ein wenig lesbarer zu machen.

```
> clear

> [ \ def ] \ def lookup serialize concat
[ \ def emptystack \ set-dict push \ assoc push
    \ get-dict push \ swap push ]
> \ def lookup concat
[ \ def emptystack \ set-dict push \ assoc push
    \ get-dict push \ swap push
swap get-dict assoc set-dict ]
```

Jetzt, da `def` definiert ist, kann man damit alle weiteren Wörter aus Kap. 5.4.2 definieren wie z.B. `*effect*`, `def+` und zuvor `swapu`, da `swapu` in der Definition von `def+` verwendet wird. Das Wort `bootstrap-word` automatisiert die Arbeit, die ich einst händisch erledigen musste.

```
>> : bootstrap-word ( wrd -- quot )
>>   [ [ ] swapu \ \ push ] keep
>>   lookup serialize concat
>>   [ def ] concat ;
```

Den Code zur Definition von `swapu` generiert `bootstrap-word` wie folgt; per Hand habe ich seinerzeit (ohne die umgebenden eckigen Klammern) exakt diesen Code geschrieben.

```
> clear \ swapu bootstrap-word
[ \ swapu emptystack \ push push \ swap push def ]
```

Die Auswahl der zum Bootstrapping benötigten Wörter, ihre Abhängigkeiten und die Reihenfolge der Definitionen dokumentiert der Definitionsrumpf des Wortes `bootstrap`.

Ist dieser Prozess des händischen Schreibens von vorbereitenden Wortdefinitionen einmal geschehen, dann kann die Prelude geladen, ein Dump produziert und der händische Vorbereitungsteil verworfen werden. Fortan können über den Dump Änderungen an der Klartext-Prelude geladen und zu einem neuen Dump aufbereitet werden. Plötzlich ist nicht mehr erkenntlich, was zuerst da war: der Dump oder die Prelude im Klartext.

Das ist schon kurios und erinnert sehr an das [Henne/Ei-Problem](#). Was war zuerst da: die Henne oder das Ei? Ohne Huhn kein Ei und ohne Ei kein Huhn. Oder am Beispiel von `def` noch extremer: Ohne `def` kein `def`. Wir Informatiker halten uns nicht lange mit solchen philosophisch anmutenden Fragestellungen auf und durchbrechen per Bootstrapping die Selbstbezüglichkeit. Das Ergebnis: Sie haben den Kreislauf von Huhn und Ei in Gang gesetzt. Auf eben diese Weise haben wir das Wort `def` aus der Taufe gehoben, um die restlichen Wörter zu definieren. Sie können nun auch `def` mit sich selber definieren. Und ist das einmal geschehen, können Sie kess die Frage nach dem „Was war zuerst?“ stellen.<sup>2</sup>

Um den Bootstrapping-Prozess nicht vollkommen zu vernebeln, ist das Wort `bootstrap` in die Prelude aufgenommen worden, die den historischen Vorgang des händischen Bootstraps zwar nicht exakt nachbildet,

<sup>2</sup> Sie dürfen davon ausgehen, dass Hühner nicht per Bootstrapping entstanden sind ;-)



aber von seiner Idee her dokumentiert. Prinzipiell könnte auch ein Mini-Dump mit den Wörtern aus `bootstrap` zum Start der Prelude herangezogen werden, damit wäre aber die Idee des Bootstrappings nicht mehr nachvollziehbar.

Die Prelude definiert auch noch einmal all die Wörter, die durch das Bootimage bereits definiert sind. Das geschieht vor allem zu Dokumentationszwecken. Die Wortdefinitionen im Bootimage kennen beispielsweise keine Angabe von Stapeleffekten. Das wird mit der Prelude nachgeholt. Die Prelude überschreibt sozusagen die Definition der Wörter aus dem Bootimage.

## 5.7 Here we go!

### 5.7.1 Consize meldet sich zur Arbeit

Die Prelude begrüßt mit `say-hi` den Anwender bzw. die Anwenderin und startet die REPL. Nun ist Consize zur Interaktion bereit!

```
>> : say-hi ( -- )
>> [ ]
>> [ This is Consize -- A Concatenative Programming Language ]
>> [ push <space> push ] each
>> pop reverse word println ;

>> say-hi repl
```

### 5.7.2 Von der Dokumentation zum Code

Die Dokumentation zur Prelude, die Sie gerade lesen, enthält den vollständigen Code zur Prelude; das sind all die Textstellen, die mit „>>“ ausgewiesen sind. Dieses Dokument ist also selbst ein Beispiel für ein „literarisches Programm“. Sie können den Quellcode der Prelude aus der Dokumentation mit dem folgenden Programm aus dem [L<sup>A</sup>T<sub>E</sub>X](#)-File extrahieren und unter einem neuen Dateinamen abspeichern.

```
> \ Consize.Prelude.tex slurp undocument \ <filename> spit
```

Achten Sie darauf, dass Sie sich nicht ungewollt die Datei mit der aktuellen Prelude überschreiben.

---

## Anhang A Mathematische Grundlagen

---

Consize ist eine funktionale Programmiersprache, die nicht – wie meist üblich – auf dem Lambda-Kalkül, sondern auf einem Homomorphismus beruht, der Programme mit Funktionen und die Konkatenation von Programmen mit Funktionskomposition in Beziehung setzt

### A.1 Der konkatenative Formalismus in denotationeller Semantik

Gegeben sei ein Vokabular  $V = \{w_1, w_2, \dots\}$  mit einer Menge von Wörtern. Die Menge aller nur endlichen Sequenzen, die mit den Wörtern des Vokabulars  $V$  gebildet werden können – das beinhaltet sowohl die leere Sequenz als auch beliebige Verschachtelungen – sei mit  $S^V$  bezeichnet und werde mit Hilfe der Kleeneschen Hülle definiert;  $S$  sei aus  $V$  abgeleitet zu  $S = \{[w_1], [w_2], \dots\}$ :

$$S^V = S^* \cup (S^*)^* \cup ((S^*)^*)^* \cup \dots$$

Der Operator zur Konkatenation  $\oplus : S^V \times S^V \rightarrow S^V$  konkateniere zwei Sequenzen  $[s_1, \dots, s_n] \oplus [s'_1, \dots, s'_m] = [s_1, \dots, s_n, s'_1, \dots, s'_m]$ .  $(S^V, \oplus, [])$  bildet einen Monoid: Die Operation der Konkatenation ist in sich geschlossen, die leere Sequenz ist das neutrale Element der Konkatenation, und das Gesetz der Assoziativität gilt.

Gegeben sei weiterhin die Menge der Funktionen  $F = \{f_1, f_2, \dots\}$ , wobei für alle Funktionen  $f \in F$  gilt:  $f : S^V_{\perp} \rightarrow S^V_{\perp}$ . Das Symbol  $\perp$  markiert in der denotationellen Semantik den Fehlerfall,  $S^V_{\perp} = S^V \cup \{\perp\}$ . Es gilt:  $f(\perp) = \perp$ .

Der Operator zur Komposition zweier Funktionen  $;\ : (S^V_{\perp} \rightarrow S^V_{\perp}) \times (S^V_{\perp} \rightarrow S^V_{\perp}) \rightarrow (S^V_{\perp} \rightarrow S^V_{\perp})$  definiere die Komposition zweier Funktionen  $f : S^V_{\perp} \rightarrow S^V_{\perp}$  und  $g : S^V_{\perp} \rightarrow S^V_{\perp}$  als  $f;g : S^V_{\perp} \rightarrow S^V_{\perp}$ , wobei gilt  $(f;g)(s) = g(f(s))$  für alle  $s \in S^V_{\perp}$ . Das neutrale Element der Funktionskomposition ist die Identitätsfunktion  $id : S^V_{\perp} \rightarrow S^V_{\perp}$  mit  $id(s) = s$  für alle  $s \in S^V_{\perp}$ .  $(S^V_{\perp}, ;, id)$  bildet ebenfalls einen Monoid.

Zu den beiden Monoiden, der Konkatenation von Sequenzen und der Komposition von Funktionen, gesellen sich zwei weitere Funktionen, um die Bedeutung (Denotation) einer Sequenz als „Programm“ zu definieren. Wir nennen eine solche Sequenz auch „Quotierung“.

Das Wörterbuch werde durch eine Funktion  $D : V \rightarrow F$  gegeben: Jedes Wort ist eindeutig mit einer Funktion assoziiert. Die Funktion  $self : S^V \rightarrow (S^V \rightarrow S^V)$  sei definiert als  $self(s)(s') = [s] \oplus s'$ ,  $s, s' \in S^V$ .

Die Denotation  $\llbracket s \rrbracket$  einer Sequenz  $s \in S^V$  liefert immer eine Sequenz verarbeitende Funktion zurück und ist definiert über sämtliche Spielarten, die für die Sequenz  $s$  denkbar sind: (1) wenn sie leer ist, (2) wenn sie ein einziges Wort enthält, (3) wenn sie eine einzige Sequenz enthält, und (4) wenn die Sequenz aus den vorigen Möglichkeiten zusammengesetzt ist.

$$(1) \llbracket [] \rrbracket = id$$

- (2)  $\llbracket w \rrbracket = f$  für  $w \in V$  und  $f = D(w)$
- (3)  $\llbracket s \rrbracket = \text{self}(s)$  für  $s \in S^V$ .
- (4)  $\llbracket s_1 \oplus s_2 \rrbracket = \llbracket s_1 \rrbracket; \llbracket s_2 \rrbracket$  für  $s_1, s_2 \in S^V$ .

Die vier Gleichungen betonen den **Homomorphismus**: Die Konkatenation von Sequenzen findet ihr Abbild in der Komposition von Funktionen. Für **Turing-Vollständigkeit** genügt ein Homomorphismus allein nicht. Es fehlt etwas, das die **Selbstbezüglichkeit** herstellt – das entscheidende Merkmal turingvollständiger Systeme

Den notwendigen Selbstbezug stellt die zu  $\text{self}$  inverse Funktion  $\text{self}^{-1}$  her. Die Funktion sei definiert über den Zusammenhang

$$(\text{self}; \text{self}^{-1})(s) = \llbracket s \rrbracket$$

Das mit der Funktion  $\text{self}^{-1}$  assoziierte Wort heiße **call**. Statt diese Assoziation als Bestandteil des Wörterbuchs einzufordern, gelte mit **call**  $\notin V$  die fünfte „Regel“:

$$(5) \llbracket \text{call} \rrbracket = \text{self}^{-1} \Rightarrow \llbracket s \rrbracket \oplus \llbracket \text{call} \rrbracket = \llbracket s \rrbracket$$

Elementarer kann man eine Programmiersprache kaum mehr formalisieren. Es ist das Minimum dessen, was in Anlehnung an die **Kategorientheorie** den Formalismus zu einer **kartesisch abgeschlossenen Kategorie** (*cartesian closed category*) und damit turingvollständig macht.

Die Implikationen sind beachtlich: Der konkatenative Formalismus entledigt sich einer „Bürde“ des Lambda-Kalküls: Variablen. Variablen sind in diesem System vollkommen überflüssig, was Begründungen über Consizes-Programme erheblich vereinfacht und das gedankliche Mitführen von Umgebungsvariablen (wie in der Veranschaulichung des **Lambda-Kalküls** üblich) unnötig macht.

Das notwendige Arrangieren von Argumenten auf dem Stapel übernehmen so genannte Stack-Shuffler, die ihrerseits Funktionen sind und sich somit vollkommen einfügen in das Schema der Funktionskomposition.

Der konkatenative Formalismus ist – ähnlich dem Lambda-Kalkül – in dieser extremen Reduktion kaum einsatztauglich für praktische Programmierzwecke. Entscheidend ist die Einführung benannter Abstraktionen. Namen sind wichtige „Krücken“ in der Begrenzung des menschlichen Intellekts sich anonyme Abstraktionen praktisch kaum merken zu können.

## A.2 Der Bezug zur operationalen Semantik, der Consizes-VM

Der konkatenative Formalismus und die **operationale Semantik** scheinen sich auf den ersten Blick fremd zu sein. Tatsächlich ist die operationale Semantik dem konkatenativen Formalismus sehr treu.

Wörter, Funktionen und Sequenzen (in Consizes durch Stapel implementiert) sind leicht in Deckung gebracht. Die Funktion von Mappings (in anderen Sprachen auch als Hashmaps, assoziative Arrays oder Dictionaries bezeichnet) kann durch Stapel emuliert werden. Mappings sind für ihren Einsatzzweck jedoch deutlich effizienter als es Stapel sind.

Die folgenden Betrachtungen beziehen sich auf die operationale Semantik, so wie sie in Kap. 3.11 durch das Wort **stepcc** definiert ist. Dabei gehen

wir ausführlich auf die drei Fallunterscheidungen bei der Beschreibung von `stepcc` ein (S. 32 ff.).

`itm` ist ein Wort

**Zum ersten Punkt in der operationalen Semantik** Die scheinbare Erweiterung, das globale Wörterbuch nicht nur auf die Assoziation mit Funktionen zu beschränken und auf Quotierungen auszudehnen, begründet sich in den Gleichungen (2) und (4). Statt neue Funktionen über die Funktionskomposition zu bilden und im Wörterbuch mit Wörtern zu assoziieren, stellt Gleichung (4) die Option in den Raum, eine Quotierung im Wörterbuch einzutragen. Semantisch ändert sich dadurch nichts, solange jede Quotierung so weit auf Einzelwörter „dekonkateniert“ und durch assoziierte Quotierungen aufgelöst wird, bis eine Funktion im Wörterbuch die Ausführung der Funktion laut Gleichung (2) einfordert. `stepcc` setzt diese Option konsequent um.

Diese Option hat mehrere, entscheidende Vorteile: Erstens sind auf diese Weise zwanglos benannte Abstraktionen eingeführt. Unbenannte Quotierungen werden als „anonyme Abstraktionen“ bezeichnet, benannte, sprich über das Wörterbuch assoziierte Quotierungen als „benannte Abstraktionen“.

Zweitens fehlt Funktionen eine sie identifizierende Repräsentation; alle Funktionen werden durch `<fct>` repräsentiert. Würde Consize ausschließlich Wörter mit Funktionen assoziieren, so wären die Repräsentationen der Funktionsabstraktionen aussageelos. Dagegen sprechen die in Quotierungen enthaltenen Wörter Klartext; sie sind identisch mit dem Programmtext bei der Programmierung. Damit ist die Reflektion von Abstraktionen in Consize sehr einfach.

Das hat drittens zur Folge, dass man den Callstack bzw. Quotierungen nicht nur einfach reflektieren, sondern ebenso einfach manipulieren kann. Davon macht `call/cc` Gebrauch, indem es die auf dem Datastack liegende Quotierung zum Programm macht und die „übrige“ Continuation frei gibt zur beliebigen Manipulation. Streng genommen kann mit `call/cc` der konkatenative Formalismus ausgehebelt werden, da eine uneingeschränkte Manipulation der rechnerischen Zukunft eines Programms möglich ist. Übt man ein wenig Programmierdisziplin und nutzt `call/cc` im Sinne einer *delimited continuation*, so ist das unkritisch und wieder im Einklang mit dem konkatenativen Formalismus. So ist denn auch die Realisierung von `call` (Gleichung (5)) mittels `call/cc` unproblematisch wie auch das Parsen von Klammern zur Laufzeit.

Anbei bemerkt zeigt die Diskussion einen interessanten Zusammenhang auf: Delimited Continuations sind das Laufzeitäquivalent einer Vorverarbeitung einer Quotierung vor ihrem Aufruf mit `call`.

Viertens kann ein konkatenatives Programmiersystem auch gänzlich anders, nämlich über ein Rewriting System mit Pattern Matching und Pattern Instantiation implementiert werden, was insbesondere das Stack-Shuffling so gut wie überflüssig macht.

Das Meta-Wort `read-word` ist ein pragmatisches Feature, um auch unbekannte, nicht im Wörterbuch aufgeführte Wörter in Anwendung von Gleichung (2) grundsätzlich mit einer Funktion zu assoziieren.

`itm` ist ein Mapping

**Zum zweiten Punkt** Ähnlich zu `read-word` ist `read-mapping` ein Feature, um Mappings – die ja auch durch Stapel umgesetzt werden könnten – gemäß Gleichung (3) zu behandeln oder ihnen, sofern ein Mapping etwas anderes darstellen soll, eine Sonderbehandlung zukommen zu lassen.

`itm` ist weder Wort noch Mapping

**Zum dritten Punkt** Wenn `itm` weder ein Wort noch ein Mapping ist, dann muss es ein Stapel, eine Funktion oder „nil“ sein. Gleichung (3) beschreibt die Behandlung eines Stapels. Funktionen sind entsprechend der dargelegten Argumentation wie Quotierungen, d.h. ebenfalls wie Stapel zu behandeln. Da „nil“ in Consize nur im Kontext von Stapeln eine Funktion hat, ist auch hier Gleichung (3) schlüssig angewendet.

Die Consize-Implementierung ist ein zu Lehrzwecken geeignetes Beispiel, welche Design-Optionen ein Formalismus erlaubt, um ihn als Virtuelle Maschine zu realisieren. Die Funktion `stepcc` beansprucht keine 20 Zeilen Code in ihrer Implementierung.