# Konkatenative Programmierung mit Consize

Dominikus Herzberg*
Technische Hochschule Mittelhessen

Viele Leser und Leserinnen, meist meine Studierenden, haben dabei geholfen, kleinere und größere Mängel im Text zu beseitigen. Vielen herzlichen Dank dafür!

# Inhaltsverzeichnis

# 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, objektorientierten 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 hervorzaubern, 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 Musikkenner 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.[1]

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 drechseln. 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

[1] 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. Die dazu notwendige Java-Laufzeitumgebung, das *Java Runtime Environment* (JRE), sollten Sie zuvor auf Ihrem Rechner installiert haben. Die Consize-Implementierung `consize.clj` samt der unabdingbar notwendigen Dateien `prelude.txt` und `bootimage.txt` werden benötigt. Am einfachsten ist es, die Dateien in das Clojure-Verzeichnis zu kopieren.

Starten Sie mit `cmd` eine Kommandozeile unter Windows, wechseln Sie mit `cd` in das Verzeichnis, in dem sich die Clojure-Installation mit den besagten Dateien befindet, und tippen Sie Folgendes ein:

```
java -cp clojure-1.5.1.jar clojure.main consize.clj
"\ prelude.txt run say-hi"
```

Aus Gründen der besseren Lesbarkeit habe ich die Zeile umgebrochen; Sie müssen alles in einer Zeile eingeben. Java wird gestartet, der Klassenpfad auf Clojure gesetzt, Clojure wird gestartet, Consize geladen und der in Anführungszeichen gesetzte Text als Startprogramm an Consize übergeben. Nach einer Weile sollte sich Consize auf der Konsole melden:

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

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

Sollten Sie Consize mit einem `exit` beenden, kommen Sie zurück zur Ebene der Kommandozeile von Windows.

Unter Windows kann eine Batchdatei sehr hilfreich sein, um sich Tipparbeit beim Starten von Clojure zu sparen.[2]

[2] Siehe z.B. "'Clojure per Kommandozeile in Windows: CLASSPATH-Setting"' auf dem denkspuren-Blog.

## 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 Einträger 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 leicht 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 ⟨*whitespace*⟩ (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 ⟨*separator*⟩ repräsentiert ein Trennzeichen und ist definiert als ein Leerraum ⟨*whitespace*⟩ gefolgt von beliebig vielen weiteren ⟨*whitespace*⟩s. Mit anderen Worten: Ein ⟨*separator*⟩ besteht aus mindestens einem ⟨*whitespace*⟩.

Als ⟨*symbol*⟩ 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 ⟨*symbol*⟩ sogar noch weiter zu fassen ist: Als ⟨*symbol*⟩ kommt jedes Terminalsymbol in Frage, das nicht als ⟨*whitespace*⟩ gilt.

Ein Wort ⟨*word*⟩ in Consize besteht aus mindestens einem ⟨*symbol*⟩. Und ein Programm ⟨*program*⟩ beginnt optional mit einem ⟨*word*⟩, es schließt

optional mit ⟨*separator*⟩ und erlaubt dazwischen beliebig viele Wiederholungen aus ⟨*separator*⟩ und ⟨*word*⟩.

Wenn Sie sich die Regel für ein Programm ⟨*program*⟩ 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 (⟨*separator*⟩) in eine Folge von Wörtern (⟨*word*⟩). 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 („schlürft“, *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 Consize 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.[1] 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 `wrd` bekannt. Es handelt sich also tatsächlich um ein Wort!

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

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

Hätte der Programmtext nicht den Grammatikregeln entsprochen, so hätte Consize den Text als syntaktisch ungültig zurückgewiesen. Wie Sie aber wissen, ist die Grammatik von Consize derart primitiv und grundlegend, dass es keine syntaktisch ungültigen Programmtexte geben kann. Consize 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 Consize so einfach, dass sie schon wieder problematisch ist: Ohne strukturbildende Ausdrucksmittel wie z.B.

[1] Ich verwende Consize zusammen mit Microsoft Windows.

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

Das mag wie ein „Hack“ wirken, mit dem sich fehlende grammatische Strukturen simulieren lassen. Im gewissen Sinne stimmt das sogar. Aber Sie lernen auf diese Weise den Umgang mit sogenannten Continuations. Mit Hilfe einer Continuation können Sie die Zukunft eines Programms verändern – ein Feature, das nur wenige Sprachen unterstützen.

## 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.

⟨*sequence*⟩ = (‘`[`’|‘`(`’) { ⟨*separator*⟩ ⟨*item*⟩ } ⟨*separator*⟩ (‘`]`’|‘`)`’) ;

⟨*mapping*⟩ = ‘`{`’ { ⟨*separator*⟩ ⟨*item*⟩ ⟨*separator*⟩ ⟨*item*⟩ } ⟨*separator*⟩ ‘`}`’ ;

⟨*item*⟩ = ⟨*word*⟩ | ⟨*sequence*⟩ | ⟨*mapping*⟩ ;

⟨*program*⟩ = [ ⟨*item*⟩ ] { ⟨*separator*⟩ ⟨*item*⟩ } [ ⟨*separator*⟩ ] ;

In der von Consize simulierten Grammatik gibt es Folgen (⟨*sequence*⟩) von Elementen (⟨*item*⟩), 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 (⟨*separator*⟩) 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 ⟨*sequence*⟩ auf ⟨*item*⟩, ⟨*item*⟩ wiederum kann eine ⟨*sequence*⟩

sein. Auf diese Weise beschreibt die Grammatik Verschachtelungen. 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 könnten, 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 `push`en, 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 mittels `\` das Wort `push` als reines Datum ausgewiesen und mit einem anschließenden `push` auf einem 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.

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 Wörterbuchs.

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 Kopfenden 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 und interpretiert, solange bis 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 ist mit einer Funktion assoziiert, die einen leeren Stapel oben auf dem Datastack ablegt. 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 mit eckigen Klammern notierter Stapel sein „offenes“, „oberes“ Ende *immer* auf der linken Seite hat.

```
[ 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 Konsole 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 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. Diese, im Wörterbuch eingetragenen Quotierungen zu einem Wort, stellen „benamte Abstraktionen“ dar. Das Wort `-rot` abstrahiert die Wortfolge `[ rot rot ]`. Anders gesagt, `-rot` ist der Name für die Abstraktion `[ rot rot ]`.

Wenn Sie ein Wort nachschlagen, das nicht im Wörterbuch verzeichnet ist, meldet Ihnen Consize das: Es kann nichts (`nil`) im Wörterbuch gefunden werden.

```
> \ x source
nil
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 bislang 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 in Konkatenation 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 Consize-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, welcher Wert 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 aus welchem Grund 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 `pop`s 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

In diesem Kapitel geht es um den Kern von Consize, um die Virtuelle Maschine (VM), die aus dem eigentlichen Interpreter und einer Laufzeitumgebung besteht.

Die Consize-VM besteht aus einem Mapping mit rund 50 atomaren Wörtern. Da die Schlüsselwerte des Mappings allesamt Wörter sind, nennen wir das Mapping „Wörterbuch“ (*dictionary*). Jedes dieser Wörter ist mit einer Funktion assoziiert. Einzig in vier Ausnahmefällen sind die Funktionen in Stapeln eingepackt.

Grundsätzlich nimmt jede Funktion einen Stapel als Input entgegen und liefert einen Stapel als Ergebnis zurück. Darum bezeichnet man konkatenative Sprachen auch gerne als „stapelverarbeitende Sprachen“. Streng genommen muss eine konkatenative Sprache nicht unbedingt aus stapelverarbeitenden Funktionen aufgebaut sein.

Der eigentliche Interpreter wird durch die mit dem Wort `stepcc` abgebildete Funktion realisiert. Das Wort ist der Dreh- und Angelpunkt der Virtuellen Maschine und ist zentral für das Verständnis des Interpreters. Dazu kommen noch die vier erwähnten Sonderfälle, die in Stapeln verpackten Funktionen zu den Wörtern `call/cc`, `continue`, `get-dict`, `set-dict`. Diese vier Funktionen ergänzen die Fähigkeit zur Metaprogrammierung.

Die übrigen Wörter der VM samt ihrer Funktionen bilden die Laufzeitumgebung ab, die nötig ist für den Umgang mit den fünf von Consize unterstützten Datentypen, für Ein- und Ausgabeoperationen, für arithmetische Operationen und für das Lesen und Zerlegen eines Consize-Programms. Wenn man wollte, könnte die Consize-VM mit deutlich weniger atomaren Wörtern und Funktionen auskommen. Darunter würde allerdings die Anschaulichkeit leiden, teils auch die Effizienz in der Ausführung. Die hier vorgestellte VM versucht eine goldene Mitte zu finden. Einerseits ist es sehr komfortabel, Stapel und Mappings als Datenstrukturen fertig zur Verfügung zu haben. Andererseits fehlt damit der Einblick, wie solche Datenstrukturen intern aufgebaut sind und funktionieren.

Eines ist in dem Zusammenhang wichtig zu erwähnen: Alle Datentypen in Consize sind immutabel (unveränderlich). Kap. 4 geht darauf näher ein.

Die Spezifikation der Consize-VM gibt pro Wort in der Regel ein Beispiel zur Verwendung des Wortes an. Die Beispiele setzen die geladene Prelude voraus, mit der Consize in aller Regel aufgerufen wird.

## 3.1 Kurzdokumentation mittels Stapeleffekten

Die mit den Wörtern der Virtuellen Maschine assoziierten Funktionen verarbeiten Stapel. Jede Funktion erwartet einen Stapel als Argument und liefert einen Stapel als Ergebnis zurück. Die Funktionen unterscheiden sich darin, wie viele Elemente sie auf dem Eingangsstapel mindestens

erwarten und von welchem Typ die Elemente sein müssen. Vom Ergebnis her interessiert, was sich im Vergleich zum Eingangsstapel auf dem Ausgangsstapel verändert hat.

Mit Hilfe des Stapeleffektes (*stack effect*) beschreibt man genau dieses Verhältnis von Erwartungen an den Eingangsstapel und den Auswirkungen auf den Ausgangsstapel. Der Stapeleffekt wird in runden Klammern notiert. Links vom Doppelstrich „`--`“ steht, was auf dem Eingangsstapel erwartet wird, rechts vom Doppelstrich steht, was der Ausgangsstapel liefert, sofern die Erwartungen an den Eingangsstapel erfüllt sind. Die Angaben für die Stapel sind von rechts nach links zu lesen, sprich, rechts ist das jeweilige obere Ende des Eingangs- bzw. Ausgangsstapels.

Ein Beispiel: Für das atomare Wort `swap` gibt `( x y -- y x )` den Stapeleffekt an. Damit ist gemeint: Auf dem Eingangsstapel müssen sich mindestens zwei Elemente befinden, wobei wir das oberste Element mit `y` und das darauf folgende mit `x` bezeichnen. Auf dem Ausgangsstapel bleiben all die nicht weiter benamten Elemente erhalten. Sie werden ergänzt um das mit `y` und – ganz zuoberst auf dem Stapel – um das mit `x` bezeichnete Element aus dem Eingangsstapel. Kurzum: `swap` tauscht die obersten beiden Elemente auf dem Stapel.

Es ist gar nicht so selten, dass sich auf dem Stapel ein oder mehrere Stapel befinden. Wollen wir uns bei der Angabe des Stapeleffekt auf den Inhalt eines Stapels beziehen, so notieren wir den Stapel in bekannter Notation mit eckigen Klammern und arbeiten ebenso mit Bezeichnern auf den Stapelpositionen. Vergessen Sie nicht, dass bei eckigen Klammern das obere Stapelende *immer* links ist. Um die übrigen Elemente eines Stapels zu fassen, arbeiten wir mit dem `&`-Zeichen als Rest-Erkenner. Das heißt: `[ x y ]` meint einen Stapel mit genau zwei Elementen, dessen obersten Wert wir mit `x` und den folgenden Wert mit `y` bezeichnen. `[ x & r ]` bezieht sich auf einen Stapel der mindestens ein Element hat. Den obersten Wert bezeichnen wir mit `x`, alle restlichen Elemente des Stapels bezeichnen wir mit `r`. `[ x y & r ]` erwartet einen Stapel mit mindestens zwei Elementen. `[ & r ]` spezifiziert einen Stapel mit beliebig vielen Elementen, die allesamt durch `r` erfasst sind. Wir machen von solchen Beschreibungen zu Stapeleffekten z.B. in Kap. 3.4 und Kap. 3.12 Gebrauch.

Konventionen:

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

- 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 oder geben ihm einen generischen Namen wie z.B. `x`, `y` oder `z`.

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 Eingangsstapel wird als *stack shuffling* bezeichnet. Ganze vier Wörter dienen dazu, jedes gewünschte Arrangement der obersten drei Elemente auf dem Datastack herzustellen.

`dup ( x -- x x )` : dupliziert das oberste Element auf dem Stapel.

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

`swap ( x y -- y x )` : vertauscht die obersten Elemente auf dem Stapel.

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

`drop ( x -- )` : entfernt das oberste Element auf dem Stapel.

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

`rot ( x y z -- y z x )` : rotiert die obersten drei Elemente auf dem Stapel, wobei das drittoberste Element nach ganz oben gebracht wird.

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

Die Stack Shuffler zum Rearrangieren, Duplizieren und Entfernen von Elementen des Eingangsstapels sind unverzichtbar, da Consize nicht das Konzept der Variable kennt. Kombinatoren sind ein anderer, eleganter Weg, wie man ohne Variablen auszukommen vermag, siehe Kap. 5.3.

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

Consize kennt insgesamt fünf immutable Datentypen. Ein Datentyp repräsentiert eine Menge von Datenwerten. So steht der Datentyp `wrd` für die Menge aller Wörter, `fct` für die Menge aller Funktionen, `stk` für die Menge aller Stapel und `map` 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“.

`type ( itm -- wrd )` : ermittelt den Datentypen des obersten Elements auf dem Stack. 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 beiden Stapelwerte; 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 Stapelwerte auf Identität. Legt `t` oder `f` auf dem Ergebnisstapel ab.

Wörter, die als Ergebnis auf dem Rückgabestapel entweder ein `t` (für *true*) oder `f` (für *false*) zurücklassen, sind meist als Prädikate (ein Begriff aus der Prädikatenlogik) zu verstehen und 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 Stapel entweder `t` oder `f`.

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

Der Stapel ist *die* allgegenwärtige Datenstruktur in Consize. Nicht nur bilden die Funktionen aller primitiven Wörter einen Eingangsstapel auf einen Ausgangsstapel ab. Man kann auch Stapel mit den Konstruktoren `emptystack` und `push` erzeugen und auf dem Ausgangsstapel ablegen. Die Zerlegung eines Stapels auf dem Eingangsstapel ist mit den Destruktoren `top` und `pop` möglich.

Wenn ein Stapel eine Folge von Daten enthält, dann sprechen wir auch immer wieder von einer Sequenz (*sequence*). Repräsentiert der Inhalt eines Stapels ein Programm, dann sprechen wir von einer „Quotierung“ (*quotation*).

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

```
> clear emptystack
[ ]
```

`push ( stk itm -- [ itm & stk ] )` : erzeugt einen neuen Stapel, 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 ( [ itm & stk ] -- itm )` : legt das oberste Element `itm` vom Stapel `stk` auf dem Ergebnisstapel 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 ( [ itm & stk ] -- stk )` : legt den um das oberste Element reduzierten Stapel auf dem Ergebnisstapel 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` weiter arbeiten 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` sind ebenfalls Konstruktoren und 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`

Mappings dienen dazu, Schlüssel- mit Zielwerten zu assoziieren, weshalb diese Datenstruktur auch assoziatives Datenfeld genannt wird. Wenn die Schlüsselwerte ausschließlich Wörter sind, dann nennen wir ein Mapping in Consize auch „Wörterbuch“ (*dictionary*). Mappings sind ein vielseitig verwendbarer Datentyp.

`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 Stapel, wobei die Assoziationen eine Folge von Schlüssel- und Zielwerten bilden. Die Reihenfolge der Schlüssel- und Zielwertpaare ist unbestimmt. Ein leeres Mapping führt zu einem leeren Stapel.

```
> 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 dem Mapping `map` die Assoziation aus Schlüsselwert `key` und Zielwert `val` hinzu und liefert das neue Mapping `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 Ergebnisstapel 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` (Stapel) 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`

Ein Wort ist in Consize ein eigener Datentyp, der eine Folge von beliebigen Einzelzeichen (*characters*) repräsentiert. Normalerweise verstehen wir in Consize unter einem Wort eine etwas striktere Auslegung: Ein Wort besteht aus mindestens einem oder mehr *sichtbaren* Einzelzeichen. Diese strikte Interpretation liegt der Arbeitsweise von `tokenize` zugrunde (Kap. 3.9). Tatsächlich kann ein Wort auch Leerzeichen und andere Steuerzeichen enthalten oder sogar leer sein. Das ist nützlich z.B. für die Erzeugung von Ausgaben auf der Konsole mittels `print` (Kap. 3.7).

`unword ( wrd -- seq )` : zerlegt ein Wort in seine Einzelzeichen 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 Ergebnisstapel 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. Der vorrangige Nutzen von `char` besteht in der Erzeugung von Sonderzeichen über die Unicode-Kodierung.

```
> 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 erscheinen Ausgaben nicht direkt auf dem Bildschirm, sondern wandern zunächst in einen Zwischenspeicher, einen Puffer.

`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 Ergebnisstapel 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 Ergebnisstapel. Tatsächlich ist `Hello you` ein einziges 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 ( source -- wrd )` : interpretiert das Wort `source` als Datenquelle, liest die Daten von dort ein und legt die Daten als Wort `wrd` auf dem Ergebnisstapel ab. Nur Datenquellen mit Textdaten können sinnvoll verarbeitet werden. Bei einfachen Wortnamen für `source` liest Consize die Daten von einer namensgleichen Datei ein, die sich im Aufrufverzeichnis von Consize befindet. Die Konventionen für Dateipfade folgen java.io. Es 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 zu unterziehen.

`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. Als Leerraum (*whitespace character* gilt eine nicht leere Folge bzw. Kombination der folgenden Zeichen: Leerzeichen, horizontaler Tabulator, vertikaler Tabulator, Zeilenvorschub, Seitenvorschub und 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 extrahierten Anteile werden per Wagenrücklauf und Zeilenvorschub miteinander verknüpft und als neues Wort `wrd'` zurück gegeben.

Mit `uncomment` unterstützt Consize eine schwache Form des *Literate Programming*, wie sie auch die funktionale Programmiersprache Haskell anbietet. Das Literate Programming zielt darauf ab, nicht den Code zu dokumentieren, sondern die Dokumentation mit Code anzureichern. Die Idee des Literate Programming stammt von Donald E. Knuth, dem Schöpfer von TEX.

So haben Sie zwei Möglichkeiten: Sie schreiben entweder Programmcode und reichern ihn mit Kommentaren an, die durch ein Prozentzeichen ausgewiesen sind; dann entfernt `uncomment` die Kommentare aus dem Consize-Code. Oder Sie schreiben eine Dokumentation und reichern diese mit Programmcode an, der Code ist durch „`>>` “ bzw. „`%>>` “ auszuweisen; dann entfernt `undocument` die Dokumentation und lässt den Consize-Code übrig, den man in einem weiteren Schritt von Kommentaren per `uncomment` befreien kann.

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: `apply`, `func`, `compose`

Die Bedeutung atomarer Wörter ist über die mit ihnen assoziierten Funktionen definiert. Diese Funktionen sind fest vorgegeben und heißen auch *primitive Funktionen*. Mit `apply` wird eine Funktion auf einen Stapel angewendet, und mit `func` können eigene Funktionen definiert werden. Das Wort `compose` erlaubt die Komposition zweier Funktionen zu einer neuen Funktion.

`apply ( stk fct -- stk' ) :` wendet die Funktion `fct` auf den Stapel `stk` an. Das Ergebnis der Funktionsanwendung ist `stk'`.

Im Beispiel wird die mit `rot` im Wörterbuch assoziierte primitive Funktion auf den Stapel `[ 1 2 3 ]` angewendet.

```
> clear [ 1 2 3 ] \ rot get-dict nil get
[ 1 2 3 ] <fct>
> apply
[ 3 1 2 ]
```

`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 Semantik der Anwendung dieser Funktion auf einen Stapel (z.B. per `apply`) ist wie folgt definiert: Der Stapel sei der initiale Datastack, die Quotierung `quot` der initiale Callstack und `dict` das initiale Wörterbuch. `stepcc` wird solange wiederholt mit dem sich verändernden Triple aus Callstack, Datastack und Wörterbuch aufgerufen, bis der Callstack leer ist. Als Ergebnis der Anwendung der per `func` erzeugten Funktion wird lediglich der Datastack als „normaler“ Stapel (wie in `apply`) zurück gegeben.

```
> clear [ 1 2 3 ] [ rot ] get-dict func
[ 1 2 3 ] <fct>
> apply
[ 3 1 2 ]
> clear [ 1 2 3 ] [ rot swap ] get-dict func apply
[ 1 3 2 ]
```

Wenn ein Fehler bei der Anwendung einer mit `func` definierten Funktion auftritt, dann wird das Wort `error` auf dem Callstack abgelegt. Die Bedeutung von `error` ist frei definierbar und nicht vorgegeben.

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

Die Bedeutung der per `compose` erzeugten Funktion ist indirekt definiert: Die Anwendung der Funktion `fct3` auf einen Stapel `stk` liefert das gleiche Resultat wie die Anwendung von `fct1` auf `stk` mit folgender Anwendung von `fct2` auf dieses Ergebnis. Anders ausgedrückt: Die Programme `compose apply` und `[ apply ] bi@` sind ergebnisgleich. Der Apply-Kombinator `bi@` ist in der Prelude definiert, siehe Kap. 5.3.5.

```
> clear [ 1 2 3 ] \ rot get-dict nil get \ swap get-dict nil get
[ 1 2 3 ] <fct> <fct>
> compose
[ 1 2 3 ] <fct>
> apply
[ 1 3 2 ]
```

## 3.11 Der Interpreter: `stepcc`

Die Definition des Interpreters kommt mit drei Konzepten aus: der Konkatenation, der Funktionsanwendung und dem Nachschlagen eines Wortes im Wörterbuch. Die zentrale Stellung der Konkatenation ist der Grund für die Charakterisierung von Consize als konkatenative Sprache. Eher nebensächlich ist, ob es bei der Konkatenation um Stapel, Arrays, Vektoren, Ströme, generell um Folgen irgendwelcher Art geht. Da Consize stapelbasiert ist, kann die Konkatenation in manchen Fällen durch ein `push` eines Elements auf den Stapel ersetzt werden. Aber das ist eher ein Detail der Implementierung, weniger ein konzeptuelles.

`stepcc ( dict ds cs -- dict' ds' cs' )` : erwartet auf dem Eingangsstapel zwei Stapel `cs` und `ds`, die wir Callstack und Datastack nennen, und ein Wörterbuch `dict`. Der Callstack `cs` muss mindestens ein Element enthalten! Das Wort `stepcc` definiert einen Rechenschritt in Consize als Veränderungen bezüglich des Callstacks, des Datastacks und des Wörterbuchs. Zum leichteren Verständnis vereinbaren wir folgende Substitutionen (Ersetzungen):[1]

- `cs` = `[ itm ] rcs concat`. Der nicht-leere Callstack kann verstanden werden als Konkatenation einer Sequenz mit einem Element `itm` und den „restlichen“ Elementen `rcs`; `itm` repräsentiert das oberste Element auf dem Callstack.
- `res` = `itm dict nil get`. Der mit dem obersten Element des Callstacks assoziierte Wert im Wörterbuch sei durch `res` erfaßt; ist kein Eintrag zu `itm` im Wörterbuch zu finden, so ist `res` = `nil`.

Die Veränderungen auf dem Ergebnisstapel seien ebenfalls in Form von Substitutionen für `cs'`, `ds'` und `dict'` notiert. Sofern in den folgenden Fallunterscheidungen nichts anderes angegeben ist, gilt grundsätzlich `cs'` = `rcs`, `ds'` = `ds`, `dict'` = `dict`.

Wort auf Callstack nachschlagen:

1. Ist `itm` ein Wort, dann schlage das Wort im Wörterbuch nach und betrachte das Resultat `res`.

– Stapel: Programmaufruf

   (a) Ist `res` ein Stapel (Quotierung/Programm):
   `cs'` = `res rcs concat`

– Funktion: Anwendung auf Datastack

   (b) Ist `res` eine Funktion:
   `ds'` = `ds res apply`

[1] Bitte verwechseln Sie das Gleichheitszeichen auf keinen Fall mit einer Zuweisung.

– sonst: Wort inspizieren

(c) weder/noch:
`ds' = [ itm ] ds concat`,
`cs' = [ read-word ] rcs concat`

Mapping auf Callstack:
– Mapping inspizieren

2. Ist `itm` ein Mapping:
`ds' = [ itm ] ds concat`,
`cs' = [ read-mapping ] rcs concat`

Funktion auf Callstack:
– Anwendung auf alles

3. Ist `itm` eine Funktion, so wird die Funktion angewendet auf den Eingangsstapel mit dem kleinen Unterschied, dass `rcs` statt `cs` verwendet wird. Der Ergebnisstapel ist das Resultat von:
`[ rcs ds dict ] r concat itm apply`
Mit `r` seien die „restlichen" Elemente des Eingangsstapels ohne die führenden drei Elemente des Stapeleffekts (`cs`, `ds` und `dict`) erfasst.

Stapel/`nil` auf Callstack:
– auf Datastack ablegen

4. Ist `itm` entweder ein Stapel oder `nil`:
`ds' = [ itm ] ds concat`

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 dem Callstack ein Wort und findet sich zu diesem Wort im Wörterbuch eine Quotierung, dann ersetzt der Inhalt der Quotierung das Wort auf dem Callstack. Die Ersetzung geschieht durch Konkatenation der Quotierung mit dem Rest des Callstacks.

```
> 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 den Datastack angewendet. Mit `get-dict` und `get` schlagen wir die zu `rot` hinterlegte Funktion im Wörterbuch nach.

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

Findet sich zu dem Wort im Wörterbuch kein Eintrag oder ist es im Wörterbuch weder mit einer Quotierung noch mit einer Funktion assoziiert, dann gilt das Wort als unbekannt. Das Wort wird auf dem Datastack abgelegt und auf dem Callstack durch `read-word` ersetzt.

```
> 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.

Ähnlich ist der Umgang mit Mappings. Ist das oberste Element auf dem Callstack kein Wort, sondern ein Mapping, dann wird das Mapping auf dem Datastack abgelegt und auf dem Callstack durch `read-mapping` ersetzt. Das Wort `read-mapping` ist ein Meta-Wort, mit dem man das Verhalten im Umgang mit Mappings frei definieren kann.

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

Ist das oberste Element auf dem Callstack weder ein Wort noch ein Mapping, sondern eine Funktion, dann wende die Funktion auf den gesamten Eingangsstapel an, mit dem `stepcc` aufgerufen wurde. Das Beispiel ist zwar wenig sinnhaft, zeigt aber dennoch, wie `rot` nicht auf den durch `[ 1 2 3 ]` repräsentierten Datastack der Continuation arbeitet, sondern auf dem Eingangsstapel.

```
> clear { } [ 1 2 3 ] [ ] \ rot get-dict nil get push
{ } [ 1 2 3 ] [ <fct> ]
> stepcc
[ 1 2 3 ] [ ] { }
```

Ist das oberste Elemente auf dem Callstack weder ein Wort noch ein Mapping noch eine Funktion, dann muss es sich um einen Stapel oder `nil` handeln; mehr als diese fünf Datentypen gibt es nicht. Ein Stapel bzw. `nil` wandert vom Callstack auf den Datastack.[2]

```
> clear { } [ z y x ] ( [ 1 2 3 ] nil ) stepcc
{ } [ [ 1 2 3 ] z y x ] [ nil ]
> stepcc
{ } [ nil [ 1 2 3 ] z y x ] [ ]
```

Die Wörter `read-word` und `read-mapping` definieren ein einfaches Metaprotokoll. Mit ihnen kann das Verhalten bei einem unbekannten Wort oder bei einem Mapping auf dem Callstack frei definiert werden. Ohne eine Definition für `read-word` wird bei Interpretation eines unbekannten Wortes eine Endlosschleife losgetreten: Wenn `read-word` unbekannt ist, wird per `read-word` nach der Bedeutung von `read-word` gefahndet.

## 3.12 Metaprogrammierung mit `call/cc`, `continue`, `get-dict`, `set-dict`, `\`

In dem durch `stepcc` definierten Interpreter sind auch ohne Fall 3 (Funktion auf Callstack) die entscheidenden Mechanismen angelegt, um ein turingmächtiges Rechensystem zu realisieren. Fall 3 erweitert den Interpreter um die Möglichkeit der Metaprogrammierung, einer Funktion wird der gesamte Zustand des Interpreters zur freien Manipulation übergeben. Damit ist all das möglich, was in anderen Programmiersprachen als Reflexion (*reflection*) oder Introspektion (*introspection*) bezeichnet wird.

Das Wort `call/cc` (*call with current continuation*) dient als Einstieg zur Manipulation von Call- und Datastack. Call- und Datastack werden in Kombination auch als „Continuation“ bezeichnet. Mit `continue` wird der Meta-Modus beendet und die meist über `call/cc` unterbrochene und manipulierte Continuation fortgesetzt (darum *continue*). Mit `set-dict` kann der aktuelle Programmkontext, das Wörterbuch gesetzt und mit `get-dict` gelesen werden.

Die Stapeleffekte zu den Wörtern sind, wie gehabt, die Effekte der Funktionen auf Eingangs- und Ausgangsstapel. Dennoch gibt es einen wichtigen Unterschied: Die Wörter sind im Wörterbuch nicht direkt mit den Funktionen assoziiert, die Funktionen sind in einem Stapel eingepackt. Das hat eine entscheidende Konsequenz bei der Interpretation des Wortes durch `stepcc`: Der Stapel wird im ersten Schritt mit dem Callstack

[2] Ich verwende die runden Klammern, damit `nil` als Datenwert *nil* und nicht als Wort `nil` in dem Stapel liegt.

konkateniert; damit befindet sich die Funktion als oberstes Element auf dem Callstack. Im zweiten Schritt greift dann der 3. Fall in `stepcc`, und die Funktion wird angewendet auf den aktuellen, gesamten Zustand aus Callstack, Datastack und Wörterbuch des Interpreters.

`call/cc ( [ quot & ds ] cs -- [ cs ds ] quot )` : erwartet zwei Stapel, die Continuation aus Callstack `cs` und Datastack `ds`, wobei sich auf dem Datastack als oberstes Element eine Quotierung `quot` befindet. Die Quotierung `quot` übernimmt als „neuer“ Callstack das Ruder der Programmausführung, die Continuation aus `cs` und `ds` bildet den Inhalt des „neuen“ Datastacks. Das durch `quot` repräsentierte Programm kann nun die auf dem Datastack abgelegte Continuation beliebig manipulieren.

`continue ( [ cs ds & r ] quot -- ds cs )` : ist das Gegenstück zu `call/cc`. Das Wort erwartet zwei Stapel, die Continuation aus der Quotierung `quot` (die durch `call/cc` in den Rang des ausführenden, aktuellen Callstacks erhoben wurde) und dem Datastack auf dem sich die übernehmende Continuation befindet `[ cs ds & r ]`. Weitere Elemente auf dem Datastack werden ignoriert, d.h. `r` spielt keine Rolle. Mit `cs` und `ds` wird die „neue“ Continuation gesetzt.

Als Beispiel für `call/cc` und `continue` diene die Implementierung des *backslash*-Wortes „\“. Der Backslash ist Teil der Consize-VM.

`\ ( ds [ wrd & cs ] -- [ wrd & ds ] cs )` : legt das dem Wort `\` folgende Element auf dem Callstack direkt 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:

```
: \ ( ds [ \ wrd & cs ] -- [ wrd & ds ] cs )
  [ dup top rot swap push swap pop continue ] call/cc ;
```

Im Beispiel wird das dem Quote folgende `swap` auf dem Datastack abgelegt, während das zweite, unquotierte `swap` seinen „normalen“ Dienst verrichtet.

```
> clear 1 2 \ swap swap
1 swap 2
```

Die Interpretation des Wortes `\` triggert die Ablage der aktuellen Continuation auf dem Datastack. Die Quotierung zu `call/cc` sorgt dann für den Transfer des Wortes vom Call- auf den Datastack. Sie können das leicht nachvollziehen, wenn Sie `\` durch ein `break` im Beispiel ersetzen.

```
> clear 1 2 break swap swap
[ 2 1 ] [ swap swap printer repl ]
```

Nun sehen Sie die Inhalte der im Moment von `break` gültigen Continuation und können händisch die Manipulation durchführen. Per `continue` wird anschließend die Programmausführung an die veränderte Continuation übergeben.

```
> dup top rot swap push swap pop
[ swap 2 1 ] [ swap printer repl ]
> continue
1 swap 2
```

`get-dict ( dict ds cs -- dict [ dict & ds ] cs )` : erwartet die Continuation aus `cs` und `ds` samt Wörterbuch `dict` und legt ein Duplikat des Wörterbuchs als oberstes Element auf dem Datastack ab.

Wenn Sie `get-dict` an der Konsole eingeben, dauert die Aufbereitung der Ausgabe eine Weile. Haben Sie ein wenig Geduld. Zwar ist die Ausgabe umfangreicher Wörterbücher oder Mappings wenig sinnvoll, aber Sie bekommen einen Einblick, was so alles im Wörterbuch steht.

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

`set-dict ( dict [ dict' & ds ] cs -- dict' ds cs )` : erwartet die Continuation aus `cs` und `ds` samt Wörterbuch `dict`, wobei sich auf dem Datastack als oberstes Element ein Wörterbuch `dict'` befindet. Das Wörterbuch `dict'` ersetzt das Wörterbuch `dict`.

Im Beispiel fügen wir dem Wörterbuch das Wort `square` mit einer Quotierung hinzu.

```
> clear [ dup * ] \ square get-dict assoc set-dict

> 4 square
16
```

## 3.13 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 $x \cdot y$ 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 $+32.767$.

## 3.14 Zum Start: `load`, `call`, `run`

Mit dem Aufruf von Consize ist über die Kommandozeile als Argument ein Consize-Programm als Zeichenkette zu übergeben. Angenommen, diese Zeichenkette werde durch das Wort `<args>` repräsentiert, so entspricht die Semantik des Programmstarts der Wortfolge

```
[ ] <args> uncomment tokenize get-dict func apply
```

Mit der Beendigung von Consize wird der aktuelle Datastack auf dem Bildschirm ausgegeben. Die Ausgabe und die Notation ist abhängig von der gewählten Implementierungssprache von Consize.

Zum Programmstart stehen alle bisher erwähnten Wörter der Consize-VM im Wörterbuch bereit inklusive der folgenden Wörter, die es erleichtern, ein Programm wie z.B. die Prelude aufzurufen und zu starten.

`load ( source -- [ & itms ] )` : liest ein Consize-Programm aus der gegebenen Quelle, typischerweise aus einer Datei, entfernt Kommentare und zerlegt den Input in Wörter und gibt eine Sequenz aus Wörtern zurück. `load` ist in der Consize-VM wie folgt definiert:

```
: load ( source -- [ & itms ] ) slurp uncomment tokenize ;
```

`call ( [ quot & ds ] cs -- ds quot cs concat )` : nimmt eine Quotierung vom Datastack und konkateniert sie mit dem Callstack.

```
: call ( [ quot & ds ] cs -- ds quot cs concat )
  [ swap dup pop swap top rot concat continue ] call/cc ;
```

Der Aufruf einer über das Wörterbuch assoziierten Quotierung, einer sogenannten „benamten“ Quotierung, ist über den Mechanismus von `stepcc` geregelt. Der Aufruf einer nicht-benamten, anonymen Quotierung ist mit `call` möglich.

`run ( source -- ... )` : liest ein Consize-Programm aus der gegebenen Quelle und führt das Programm aus.

```
: run ( source -- ... ) load call ;
```

**!** → Beim Start von Consize sind die Meta-Wörter `read-word`, `read-mapping` und `error` nicht definiert! Damit fehlt Consize das Wissen, wie es mit unbekannten Wörtern, Mappings und Fehlersituationen umgehen soll. Die Bedeutung dieser Wörter ist von der Programmiererin bzw. dem Programmierer festzulegen.

## 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?`.

Optionale Wörter der Consize-VM müssen, sofern sie nicht Teil einer Consize-Implementierung sind, in einer angepassten Prelude bereitgestellt werden.

# 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 `push`en 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 `push`en und `top`en 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 ge`dup`tes 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 `push`en, wird der Stack niemals verändert, sondern ein neuer Stack erzeugt, der alle Element des bisherigen Stacks inklusive des ge`push`ten Elements enthält. Der Stack *vor* dem `push` kann niemals mit dem Stack *nach* dem `push` identisch sein.[1] 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.

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

Der Clou an Consize 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 Consize 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 Consize, um den Quelltext aus der Dokumentation zu filtern.

Auch wenn der Einstieg in die Prelude gleich ein unvermittelter Einstieg in die Programmierung mit Consize ist: Sie werden sehen, Consize ist nicht wirklich schwer zu verstehen. In Consize 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 Consize: Ihnen stehen die Consize-Erweiterungen direkt zur Verfügung. Sie arbeiten mit einer geladenen Prelude, um die Prelude zu verstehen. Das ist einfacher und umkomplizierter als sich das anhört. Sie können ein paar Hilfsmittel, wie z.B. den Debugger nutzen, um sich die Arbeitsweise von Consize zu veranschaulichen.

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

### 5.1.1 Consize-Lizenz

Die Prelude beginnt mit einer Präambel. Die Prelude ist Open-Source-Software (OSS). Consize soll als Bildungsgut allen Interessierten frei zur Verfügung stehen.

```
>> %%% A Prelude for Consize in Consize
>> %%% Copyright (c) 2017, Dominikus Herzberg, https://www.thm.de
>> %%% New BSD License: http://opensource.org/licenses/BSD-3-Clause
```

### 5.1.2 Booting zur Verarbeitung der Prelude

In Consize 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 benamten Abstraktion: `-rot` ist der „Name“ für die Abstraktion `[ rot rot ]`.

```
> \ -rot get-dict nil get
[ 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. Ansonsten stoßen unsere intellektuellen Fähigkeiten beim Programmieren 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.

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

⟨*definition*⟩ = ':' ⟨*separator*⟩ ⟨*word*⟩ [ ⟨*separator*⟩ ⟨*stackeffect*⟩ ] ⟨*separator*⟩
⟨*program*⟩ ⟨*separator*⟩ ';'

Die Angabe von Stapeleffekten kennen Sie bereits aus Kap. 3. Alle Wörter der Consize-VM sind dort mit ihren Stapeleffekten angegeben worden. Details zur Umsetzung von ⟨*definition*⟩ finden sich in Kap. 5.7.2, S. 67.

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

```
>> \ bootimage.txt run
```

In Kap. 5.9.3 ist beschrieben, was in dem Bootimage steht und wie es erzeugt wird.

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

Der durch `stepcc` beschriebene Ausführungsmechanismus der Consize-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.

Die Prelude assoziiert beide Wörter mit „leeren“ Programmen. Das bedeutet: Unbekannte Wörter werden genauso wie Mappings 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 alternative Definition dieser Meta-Wörter das Risiko birgt, existierenden Code in größerem Umfang zu brechen. Man muss sehr sorgsam mit diesen Wörtern umgehen.

### 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 zwischenzuspeichern 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 die Erzeugung von Duplikaten auf dem Datastack möglich sein (deshalb gibt es `dup` in der Consize-VM) und das Rearrangieren von Werten über Stack-Shuffling gelöst werden. Es ist daher sehr hilfreich, weitere Wörter als Abstraktionen zu den Stack-Shufflern der Consize-VM 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 Stapeleffekten. Stapeleffekte beschreiben oft hinreichend genau, was ein Wort tut, so dass man sich das *Wie* der Manipulation der Werte auf dem Eingangsstapel nicht merken muss. Stapeleffekte erfüllen die Funktion einer Schnittstellenbeschreibung (*interface description*).

```
>> : 2drop ( x y -- ) drop drop ;
>> : 3drop ( x y z -- ) drop drop drop ;
>> : 2dup ( x y -- x y x y ) over over ;
>> : 3dup ( x y z -- x y z x y z ) pick pick pick ;
>> : dupd ( x y -- x x y ) swap dup rot ;    % dup deep
>> : swapd ( x y z -- y x z ) swap rot rot ; % swap deep
>> : -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 ;
>> : pick ( x y z -- x y z x ) rot dup [ -rot ] dip ;
>> : over ( x y -- x y x ) swap dup -rot ;
>> : 2over ( x y z -- x y z x y ) pick pick ;
>> : nip ( x y -- y ) swap drop ;
>> : 2nip ( x y z -- z ) nip nip ;
```

Es deutet sich bei den Definitionen einiger Stack-Shuffler wie z.B. `rot4` und `-rot4` 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' ) cons ; % deprecated
>> : cons ( itm stk -- [ itm & stk ] ) swap push ;
>> : uncons ( [ itm & stk ] -- itm stk ) dup top swap pop ;
>> : unpush ( [ itm & stk ] -- stk itm ) dup pop swap top ;
>> : empty? ( stk -- t/f ) ( ) equal? ;
>> : size ( seq -- n ) dup empty? [ drop 0 ] [ pop size 1 + ] if ;
>> : time ( quot -- ... msecs )
>>   current-time-millis swap dip current-time-millis swap - ;
```

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

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

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

Das Wort `lookup` schlägt den mit einem Wort assoziierten Wert im Wörterbuch nach, `delete` entfernt das Wort auf seinen Zielwert aus dem Wörterbuch. Das Wort `values` gibt – im Gegensatz zu `keys` – alle mit den Schlüsseln assoziierten Zielwerte als Sequenz zurück.

```
>> : nil ( -- nil ) ( ) top ;
>> : lookup ( word -- item ) get-dict nil get ;
>> : delete ( itm -- ) get-dict dissoc set-dict ;
>> : values ( dict -- seq ) dup keys swap [ nil get ] cons map ;
```

### 5.1.6 Kombinatoren: `call`, `fcall`

Ein „Kombinator“ ist ein Wort, das einen Stapel zur Ausführung bringt, d.h. das den Stapel als Programm interpretiert. Einen solchen Stapel nennen wir „Quotierung“. Der Stapel dient nur als Mittel zum Zweck, um die Ausführung des darin enthaltenen Programms zurückzustellen. Der Kombinator aktiviert die Quotierung.

Wenn Consize ein Programm abarbeitet, ist es zu einem guten Teil mit der Auflösung benamter Abstraktionen beschäftigt: Das oberste Wort auf dem Callstack wird durch den Inhalt der mit ihm assoziierten Quotierung ersetzt. Technisch ausgedrückt: 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 der gleiche Ablauf – realisiert durch den Kombinator `call`. Das Wort erwartet eine Quotierung auf dem Eingangsstapel, die es mit dem Callstack konkateniert. Implementiert werden kann dieses Verhalten mit Zugriff auf die aktuelle Continuation. Der Stapeleffekt soll die Umsetzung per `call/cc` abbilden.

```
>> : call ( [ quot & ds ] cs -- ds quot cs concat )
>>   [ swap unpush rot concat continue ] call/cc ;
```

Das mit `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`).[1]

Im Beispiel wird das per Quotierung zurückgestellte Programm, die Addition, erst durch den Aufruf von `call` ausgeführt.

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

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 auch „Laufzeit-Metaprogrammierung“.

Mit Hilfe des `call`-Kombinators können direkt oder indirekt alle anderen Kombinatoren abgeleitet werden. Einzig `fcall` (für „*call via a function*“) ist eine Ausnahme, aber nur deshalb, weil es eine andere Implementierungsstrategie über Funktionen verfolgt. Es erzeugt im Gegensatz zu `call`

[1] `call` ist bereits in der Consize-VM definiert, siehe Kap. 3.14. Da `call` nicht zwingend Teil der VM sein muss, wird die Definition in der Prelude wiederholt.

über `func` einen eigenen Ausführungskontext, bekommt einen leeren Stapel als Eingangsstapel, 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 ]
```

Das Wort `fcall` nimmt den Callstack der Implementierungssprache von Consize in Anspruch, `call` nutzt den Callstack der aktuellen Continuation.

Die Namen der Kombinatoren in den folgenden Kapiteln orientieren sich in vielen Fällen an der konkatenativen Sprache Factor. Bisweilen sind auch die Definition der Kombinatoren von Factor übernommen.

## 5.2 Entscheidungskombinatoren

Ein grundlegendes Feature einer turingvollständigen Programmiersprache ist es, Entscheidungen treffen zu können: wähle – abhängig von einem Entscheidungswert – entweder dieses oder jenes. Im einfachstenfall ist der Entscheidungswert zweiwertig, 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 Werte `t` und `f` diese beiden Optionen; in einem Programm schreibt es sich lesbarer mit `true` und `false`.

### 5.2.1 Boolesche Werte und die binäre Wahl mit `choose`

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 )` bildet 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 ;
```

Die Wörter `true` und `false` dienen lediglich dem Lese- und Schreibkomfort beim Programmieren; es ist rasch vergessen, das Escape-Wort einem `t` bzw. `f` voranzustellen, denn schließlich sollen die Programme für `t` und `f` nicht sofort ausgeführt werden.

```
>> : true  ( -- t ) \ t ;
>> : false ( -- f ) \ f ;
```

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 eines nicht booleschen Wertes 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.

Wenn die booleschen Werte nur im Kontext von `choose` sinnvoll interpretiert werden können, stellt sich die Frage, ob die Bedeutung von `t` und `f` überhaupt außerhalb von `choose` bekannt sein muss. Anders gefragt: Kommen wir ohne Einträge für `t` und `f` im Wörterbuch aus? In der Tat lässt sich ein „lokales“ Wörterbuch im Rumpf der Definition von `choose` verwenden. Und den Vergleich per `equal?` bekommen wir bei einer Wörterbuchabfrage per `get` gleichermaßen „geschenkt“.

```
>> SYMBOL: t
>> SYMBOL: f
>>
>> : choose ( f/* this that -- that/this )
>>   rot { \ f [ swap drop ] } [ drop ] get call ;
```

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

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

Die logischen Operationen der Booleschen Algebra `and` (Konjunktion), `or` (Disjunktion), `xor` (ausschließende 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
>> : xor ( f/* f/* -- t/f ) [ f swap choose ] when* ; % 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
```

### 5.2.2 Binäre Entscheidungen: `if`, `when`, `unless` & Co.

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 er *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
```

Jeder Wert außer `f` gilt im Kontext eines `if`, `if-not`, `when` oder `unless` als logisch *true*, selbst wenn der Bedingungswert nicht gleich `t` ist. Da macht es bisweilen Sinn, den logisch wahren Bedingungswert auf dem Stapel durch ein implizites `dup` zu erhalten, um mit ihm weiter rechnen zu können. Genau dafür gibt es die „Stern-Varianten“ `if*`, `when*` und `unless*`. Ist der Bedingungswert `f`, so unterbleibt die Duplizierung und `if*`, `when*` bzw. `unless*` arbeiten wie ihre „sternlosen“ Vorbilder `if`, `when` und `unless`.

```
>> : if* ( f/* then else -- ... )
>>   pick [ drop call ] [ 2nip call ] if ; % Factor
>> : when* ( f/* then -- ... ) over [ call ] [ 2drop ] if ; % Factor
>> : unless* ( f/* else -- ... ) 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
```

### 5.2.3 *n*-äre Entscheidungen: `case` und `cond`

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 ... } -- ... )
>>   :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 Verschachtlung 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 ] -- ... )
>>   dup empty?                 % anything left to test?
>>     [ drop ]                 % no: quit
>>     [ uncons dup empty?      % only one quotation left?
>>       [ drop call ]          % yes: call 'else'
>>       [ uncons               % otherwise:
>>         [ ] \ cond push cons % 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 ein besseres Laufzeitverhalten, während die Laufzeit von `cond` mit jeder Verschachtlung linear anwächst. Umgekehrt kann nicht jedes `cond` in ein äquivalentes `case` umgewandelt werden.

## 5.3 Aufruf-Kombinatoren

Dieses Kapitel beschäftigt sich mit Kombinatoren, die Varianten von `call` sind und unter dem Oberbegriff der „Aufruf-Kombinatoren“ (`call`-Kombinatoren) laufen. Die Abtauch-Kombinatoren (`dip`-Kombinatoren) lassen die oberen Stapelwerte vor dem `call` der Quotierung abtauchen, die Erhaltungskombinatoren (`keep`-Kombinatoren) restaurieren die oberen Werte nach dem `call` wieder. Wieder andere Kombinatoren rufen zwei oder mehr Quotierungen nach verschiedenen Mustern auf (Cleave-, Spread- und Apply-Kombinatoren).

Generell reduzieren diese Kombinatoren die Notwendigkeit des Stack-Shufflings und bringen deshalb lesbarere Programme mit sich.

### 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 von `dip` im Definitionsrumpf durch die Definition von `dip` keine weitere Erkenntnisse. Einen Hinweis, wie `dip` implementiert werden kann, liefert sie dennoch: Es ist der Versuch, das im Stapeleffekt 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 cons \ \ 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. 5.7.3 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 [ ] cons \ \ push concat call ;
```

```
: dip ( x quot -- x ) reverse \ \ push cons 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.

Interessant sind in Consize die wechselseitigen Bezüge, die Wörter zueinander haben. Das offenbart innere Strukturen, die andere Programmiersprachen weniger deutlich erkennen lassen. Zum Beispiel könnten `dup`, `2dup` und `3dup` auch über Erhaltungskombinatoren definiert sein.

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

Andererseits ist `keep` mit `dup` definiert worden, `2keep` mit `2dup` etc. Es ist alles eine Frage, aus welchen Wörtern die Consize-VM besteht. Daraus sind die nicht atomaren Wörter abzuleiten.

Eine alternative Implementierung für die Erhaltungskombinatoren 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`, `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. 5.4 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*`, `tri*`, `spread`

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 Spread-Kombinatoren `bi*`- und `tri*` bietet das Wort `spread`; es erwartet $n$ Elemente und entsprechend $n$ Quotierungen in einem Stapel. Die $n$-te Quotierung wird auf das $n$-te Element angewendet.

Die Umsetzung des Wortes `spread` geschieht via `SPREAD`, das den notwendigen Code verschachtelter `dip`-Aufrufe für das gewünschte „Spreading“ erzeugt. Das Wort `reduce` ist ein Sequenzkombinator, siehe Kap. 5.4.

```
>> : SPREAD ( [ quot1 ... quotn ] -- ... ) % def inspired by Factor
>>   ( ) [ swap dup empty?
>>           [ drop ]
>>           [ [ dip ] rot concat cons ]
>>         if ]
>>   reduce ;
```

Greifen wir das obige Beispiel für `tri*` auf. `SPREAD` erzeugt den benötigten Code, d.h. `SPREAD` ist ein Code-Generator. Ein anschließendes `call` bringt den generierten Code zur Ausführung. Das Ergebnis ist mit dem `tri*`-Beispiel identisch.

```
> clear 4 3 2 ( [ 1 + ] [ dup * ] [ 1 - ] ) SPREAD
4 3 2 [ [ [ 1 + ] dip dup * ] dip 1 - ]
> call
5 9 1
```

Damit erklärt sich auch die Umsetzung von `spread`:

```
>> : spread ( itm1 ... itmn [ quot1 ... quotn ] -- ... ) SPREAD call ;
```

### 5.3.5 Apply-Kombinatoren: `bi@`, `tri@`, `both?`, `either?`

Die Apply-Kombinatoren sind „Anwendungskombinatoren“ (*apply* heißt „anwenden“), die wie Spread-Kombinatoren arbeiten, im Gegensatz dazu jedoch nur eine Quotierung erwartet, die entsprechend `dup`liziert 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
```

Zwei Beispiele für die Anwendung des `bi@`-Kombinators sind `both?` und `either?`.

```
>> : both? ( x y pred -- t/f ) bi@ and ;
>> : either? ( x y pred -- t/f ) bi@ or ;

> clear 2 -3 [ 0 > ] both?
f
> 2 -3 [ 0 > ] either?
f t
```

## 5.4 Sequenzkombinatoren

Sequenzkombinatoren wenden eine Quotierung auf jedes Element einer Sequenz 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.

### 5.4.1 Elemente bearbeiten: `each`, `map` und `reduce`

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. Der rekursive Aufruf ist in *tail position*, d.h. er ist das letzte Worte am Ende (*tail*) der Quotierung, die für die Rekursion verantwortlich ist. Man spricht auch von *tail recursion*, im Deutschen als „Endrekursion“ bezeichnet. Sie zeichnet sich dadurch aus, dass die Rekursion nicht zum Anwachsen des Callstacks führt – das Merkmal von Schleifenkonstrukten in imperativen Programmiersprachen.

```
>> : 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
```

Ein einfaches Beispiel für die Verwendung von `each` ist das Entpacken eines Stapel, hier definiert in Form des Wortes `unstack`.

```
>> : unstack ( stk -- ... ) ( ) each ;

> clear [ x [ y ] z ] unstack
x [ y ] z
```

Die Varianten `2each` und `3each` erwarten zwei bzw. drei Stapel, greifen dort jeweils das oberste Element ab und rufen damit die Quotierung auf.

```
>> : 2each ( stk1 stk2 quot -- ... )
>>   \ unstack push [ zip ] dip each ;
>> : 3each ( stk1 stk2 stk3 quot -- ... )
>>   \ unstack push [ 3zip ] dip each ;
```

Jeweils ein Beispiel illustriere den Gebrauch der Wörter.

```
> clear ( 1 2 3 ) ( 4 5 6 ) [ + ] 2each
5 7 9
> clear ( 1 2 ) ( 3 4 ) ( 5 6 ) [ + * ] 3each
8 20
```

Das Wort `map` fasst im Gegensatz zu `each` die Ergebnisse 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 bezüglicher 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 ) swapd 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 Beispiele motivieren drei nützlichen Abstraktionen: `sum` zur Summenbildung, `prod` zur Produktbildung und `cat` zur Verschmelzung mehrerer Sequenzen.

```
>> : sum ( [ x ... z ] -- sum ) 0 [ + ] reduce ;
>> : prod ( [ x ... z ] -- prod ) 1 [ * ] reduce ;
>> : cat ( [ seq1 ... seq2 ] -- seq ) ( ) [ concat ] reduce ;
```

Eine alternative, endrekursive (*tail recursive*) Definition für `size` (siehe Kap. 5.1.5) ist:

```
: size ( seq -- n ) 0 [ drop 1 + ] reduce ;
```

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.

Auch wenn sich Programme sehr kompakt mit `map` und `reduce` darstellen lassen, nicht immer sind diese Wörter die ideale Wahl. Die Definitionen

von `any?` und `all?` haben einen entscheidenden Nachteil: Sie laufen Gefahr zuviel des Guten zu tun. Bei `any?` ist der Abbruch sinnvoll, sobald die Anwendung des Prädikats auf ein Element erfolgreich ist – Folgewerte müssen per `map` nicht mehr untersucht werden. Ebenso kann `all?` abbrechen, sobald ein Prädikatstest fehl schlägt.

```
>> : any? ( seq pred -- t/f ) map f [ or ] reduce ;
>> : all? ( seq pred -- t/f ) map t [ and ] reduce ;

> clear ( 1 3 -4 5 0 7 2 ) [ 0 <= ] any?
t
> clear ( 1 3 -4 5 0 7 2 ) [ 0 >= ] all?
f
```

Natürlich kann man `any?` und `all?` entsprechend anders rekursiv programmieren. Doch die konzeptuelle Kürze mit `map` und `reduce` besticht! Es gibt funktionale Sprachen, wie z.B. Haskell, die auf eine andere Strategie zur Auswertung von Programmausdrücken zurückgreifen. Mit einer verzögerten Auswertung (*lazy evaluation*) werden überflüssige Rechenschritte vermieden, die konzeptuelle Kürze aber beibehalten.[2]

Zur Arbeit mit zwei oder drei Sequenzen stehen die folgenden Varianten von `map` und `reduce` bereit:

```
>> : 2map ( seq1 seq2 quot -- seq ) [ zip ] dip \ unstack push map ;
>> : 3map ( seq1 seq2 seq3 quot -- seq ) [ 3zip ] dip \ unstack push map ;
>> : 2reduce ( seq1 seq2 identity quot -- res )
>>   [ zip ] 2dip \ unstack push reduce ;
>> : 3reduce ( seq1 seq2 seq3 identity quot -- res )
>>   [ 3zip ] 2dip \ unstack push reduce ;
```

### 5.4.2 Sequenzverschnitte: `zip`

Oft ist der Wunsch, die Elemente von zwei oder mehr Stapeln zusammen bearbeiten möchte. Eine Lösung dazu bietet `zip`, das im Reißverschlußverfahren die jeweils obersten Elemente zweier Stapel zusammenfasst und aus den Paaren einen neuen Stapel erstellt.

```
>> : zip ( stk1 stk2 -- stk )
>>   2dup [ empty? ] bi@ or
>>     [ 2drop ( ) ]
>>     [ unpush ( ) cons  rot
>>       unpush rot cons -rot swap zip cons ]
>>    if ;
```

Am einfachsten ist `zip` am Beispiel zu verstehen.

```
> clear ( 1 2 3 ) ( 4 5 6 ) zip
[ [ 1 4 ] [ 2 5 ] [ 3 6 ] ]
```

Sind die beiden Stapel nicht gleich in der Anzahl ihrer Elemente, endet der Reißverschluß mit dem letzten Element des kürzeren Stapels.

```
> clear ( 1 2 3 4 ) ( 5 6 ) zip
[ [ 1 5 ] [ 2 6 ] ]
```

Die Wörter `3zip` und `4zip` bringen die Elemente von drei bzw. vier Stapeln zusammen.

[2] Trivial sind solche Überlegungen nicht, wie der Beitrag „Reducers in Clojure 1.5“ von Stefan Kamphausen auf heise-Developer aufzeigt.

```
>> : 3zip ( stk1 stk2 stk3 -- stk ) zip zip [ unstack cons ] map ;
>> : 4zip ( stk1 stk2 stk3 stk4 -- stk ) 3zip zip [ unstack cons ] map ;

> clear ( 1 2 ) ( 3 4 ) ( 5 6 ) 3zip
[ [ 1 3 5 ] [ 2 4 6 ] ]
> clear ( 1 2 ) ( 3 4 ) ( 5 6 ) ( 7 8 ) 4zip
[ [ 1 3 5 7 ] [ 2 4 6 8 ] ]
```

### 5.4.3 Aussortieren: `filter`, `remove`

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 pred -- seq' ) % pred is a quotation
>>   ( ) -rot [ keep and [ push ] when* ] cons each reverse ;

> 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 ]
```

## 5.5 Wiederholungskombinatoren

In einer funktionalen Sprache gibt es einzig die Rekursion als Mittel, um die Idee der Wiederholungen eines Vorgangs auszudrücken. Sequenz- wie auch Wiederholungskombinatoren abstrahieren gängige Muster der Wiederholung für verschiedene Zwecke und machen den Programmcode leichter lesbar.

### 5.5.1 Abbruch via Prädikat: `loop`, `do`, `while`, `until`

Die Wörter `while` und `until` abstrahieren ein gängiges Rekursionsschema: Wiederhole den Aufruf der Quotierung solange, wie das Prädikat ein *true* (`while`) bzw. ein *false* (`until`) zurück liefert. Beide Abstraktionen lassen sich auf das Wort `loop` zurückführen.

```
>> : loop ( pred -- ... ) [ call ] keep [ loop ] curry when ;
>> : do ( pred quot -- pred quot ) dup 2dip ;
>> : while ( pred quot -- ... ) swap do concat [ loop ] curry when ;
>> : until ( pred quot -- ... ) [ [ not ] concat ] dip while ;
```

Die Beispiele zeigen den Gebrauch der Wiederholungskombinatoren am Beispiel der Berechnung der Fakultät von 4.

```
> clear 1 4 [ [ * ] keep 1 - dup 0 > ] loop drop
24
> clear 4 1 [ over 0 > ] [ over * [ 1 - ] dip ] while nip
24
> clear 4 1 [ over 0 == ] [ over * [ 1 - ] dip ] until nip
24
```

Mit `do while` kann der Aufruf der Quotierung einmal vor der Abarbeitung durch `while` erzwungen werden.

### 5.5.2 Abbruch als Fixpunkt: `X`, `Y`

In der Regel wird die Rekursion dadurch hergestellt, dass ein Wort sich selbst im Definitionrumpf erwähnt und aufruft. Das ist die benamte Rekursion. Aber wie ist es um die anonyme, unbenamte Rekursion bestellt? Wie kann man ohne den Selbstaufruf eines Wortes Rekursion erzeugen?

Den Schlüssel zur Antwort liefert der `X`-Kombinator. Eine Quotierung dupliziert sich vor ihrem Aufruf, womit eine Programmkopie für einen Wiederholungsaufruf auf dem Stapel bereit liegt.

```
>> : X ( quot -- ... ) dup call ;
```

Der Lambda-Kalkül, die Basis vieler funktionaler Programmiersprachen, setzt die anonyme Rekursion mit dem Fixpunkt-Kombinator um, auch Y-Kombinator bezeichnet. Der Fixpunkt-Kombinator wiederholt die Anwendung einer Funktion auf einen Funktionswert solange, bis Eingangs- und Ausgangswert identisch sind – der Fixpunkt ist erreicht.

In Consize drückt sich der Y-Kombinator wie folgt aus; der X-Kombinator sorgt dafür, dass die Rekursion anonym bleibt.

```
>> : Y ( val quot -- res )
>>   [ [ [ call ] 2keep -rot dupd equal? ] dip
>>     swap [ drop nip ] [ swapd X ] if ] X ;
```

Die Fakultät von `4` lässt sich mit dem Y-Kombinator ohne jegliche namentliche Wort-Rekursion berechnen.

```
> clear

> 4 1 [ swap dup 0 equal? [ drop 1 ] when [ * ] keep 1 - swap ] Y nip
24
```

Man könnte also durchaus Programme in Consize schreiben, selbst wenn das Wörterbuch fixiert wäre und – da es dann kein `set-dict` gäbe – und es unmöglich wäre, neue Wörter im Wörterbuch mit ihren Quotierungen zu hinterlegen. Man müsste alle nicht primitiven Wörter selber händisch auflösen. Das ist mehr als unpraktisch, zeigt aber eines: Ein Rechenformalismus braucht keine benamten Abstraktionen, auch sind benamte Abstraktionen keine Notwendigkeit, um Programmieren zu können. Wenn jemand benamte Abstraktionen braucht, so sind es wir Menschen. Für uns ist eine Programmiersprache ohne benamte Abstraktionen schlicht unbrauchbar, unsere intellektuellen Fähigkeiten sind zu begrenzt. Was andererseits betont, wie wichtig die Wahl eines guten Namens für eine Abstraktion ist. Da geht es um nichts anderes als die Kommunikation von Mensch zu Mensch.

## 5.6 Kompositionskombinatoren: `curry`

Zwei Kombinatoren zur Komposition von Programmen bzw. Funktionen sind bereits Bestandteil der Consize-VM. Mit `concat` lassen sich zwei Quotierungen zu einer zusammenfassen. Die Komposition zweier Funktionen erfolgt mit `compose`.

Eine weitere Wortgruppe, die die Prelude zur Komposition von Programmen bereitstellt, sind `curry`, `2curry` und `3curry`. Das sogenannte „Currying“ ist ein Begriff aus der Welt der funktionalen Programmierung, der Name ist zu Ehren des Mathematikers Haskell Brooks Curry gewählt worden. In einer konkatenativen Sprache ist das Currying trivial: die gwünschte Anzahl an Argumenten wird in die Quotierung „gezogen“; das verkürzt die noch benötigten Argumente auf dem Stapel, wenn die Quotierung aufgerufen wird.

```
>> : curry ( itm quot -- quot ) cons \ \ push ;
>> : 2curry ( itm1 itm1 quot -- quot ) curry curry ;
>> : 3curry ( itm1 itm2 itm3 quot -- quot ) curry curry curry ;

> 1 [ + ] curry
[ \ 1 + ]
```

## 5.7 Erweiterung der Consize-Grammatik

Consize hat eine äußerst einfache Grammatik, die jeden Programmtext erfolgreich in eine Folge von Wörtern zerlegen lässt. Leerräume im Programmtext grenzen Wörter voneinander ab, siehe auch Kap. 3.9. Jedes Consize-Programm beginnt als eine Folge von Wörtern.

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

Ein Beispiel für eine syntaktische Struktur ist die Definition neuer Wörter. Nach dem einleitenden Doppelpunkt folgt das Wort, der optionale Stapeleffekt und die definierenden Wörter bis zum Semikolon. Damit befasst sich Kap. 5.7.2.

In Kap. 5.7.3 wird eine weitere, kleine syntatische Struktur zur Definition von Wörtern eingeführt, die als Symbole fungieren.

### 5.7.1 Literale: `[ ... ]`, `( ... )`, `{ ... }`

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 Klammern 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 cons swap scan4] scan4] ]
>>     \ {   over
>>     \ (   over
>>     \ \   [ drop unpush rot cons \ \ push swap scan4] ]
>>     :else [ rot cons 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 cons \ \ push swap scan4[ ]
>>     :else [ rot cons 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 ]
```

### 5.7.2 Wörter definieren: von : 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 ;
```

Das Wort `def+` ist ein um die Berücksichtigung des Stapeleffekts erweitertes `def`; tatsächlich wird der Stapeleffekt ignoriert und verworfen.

```
>> : def+ ( wrd [ effect ] [ body ] -- ) swap drop def ;
```

Das Wort `:` leitet die Definition eines neues 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 der Wortdefinition.

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

Das Wort `destruct-definition` zerlegt die zwischen `:` und `;` aufgesammelten 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 die wiederholte, Zeitintensive Interpretation der Literale zur Laufzeit vermeidet.

```
>> : destruct-definition ( quot -- wrd stackeffect body )
>>   uncons                % 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 cons swap look4) ]
>>     :else [ rot cons swap look4) ]
>>     nil   [ \ syntax-error [ incomplete stack effect ] _|_ ]
>>   } case ;
```

### 5.7.3 Datenwort definieren: `SYMBOL:`

In Consize erwartet jedes Wort, die zu verarbeitenden Daten auf dem Datastack vorzufinden. Es gibt 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, indem 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 Spielereien stehen nicht im Widerspruch zum konkatenativen Paradigma, sie erfordern nur die Auflösung der syntaktischen Form in „normale“ konkaktenative 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 ( \ ; ) cons \ \ push cons \ : 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.[3] Als Lösung gelten begrenzte Continuations (*delimited continuations*, die auch unter dem Namen *partial continuations* firmieren), die einen unsachgemäßen Gebrauch erst gar nicht ermöglichen.

## 5.8 Die Interaktion mit Consize

Die Interaktion mit Consize beschränkt sich auf die Übergabe eines Programms beim Aufruf der Consize-VM. Damit lässt sich nicht wirklich arbeiten. Also gilt es, Consize die Interaktion per Konsole „beizubringen“.

### 5.8.1 Datenrepräsentation und -Ausgabe: `repr`, `println`

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, so 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 ]
```

[3] http://lambda-the-ultimate.org/node/4586 bietet einen guten Einstieg in die Diskussion.

```
>>     \ fct [ drop \ <fct> ]
>>     \ nil [ drop \ nil ]
>>     :else [ \ repr-error [ unknown type ] _|_ ]
>>   } case ;
```

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 ;
```

### 5.8.2 Die Interaktion über die Konsole, die `repl`

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*). Dieser Ablauf 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>` zur eingelesenen Eingabezeile hinzugefügt werden.

```
>> : reader ( -- quot )
>>   \ > print <space> print flush read-line
>>   ( ) <newline> push cons 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.

### 5.8.3 Dateien lesen und starten: `(l)load`, `(l)run`

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-test.txt run ;
```

### 5.8.4 Abbruch und Reflexion: `exit`, `abort`, `source`, `clear`

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: `source` gibt den mit einem Wort assoziierten Wert aus dem Wörterbuch 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.

```
>> : 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 ;
```

### 5.8.5 Debugging: `break`, `step`

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 ganzer 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 ;
```

Ähnlich ist auch das Wort `error` definiert, so dass ein Verarbeitungsfehler in Consize kontrolliert aufgefangen werden kann:

```
>> : error ( -- ) [ \ error 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 ;
```

### 5.8.6 Unit-Testing: `unit-test`

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 Umgebung für Unit-Tests 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 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 ]
```

## 5.9 Serialisierung, Consize-Dumps und Bootstrapping

In einer funktionalen Programmiersprache mit referentieller Transparenz existiert das Konzept der Referenz nicht, was die Serialisierung von Daten sehr einfach macht. Insofern ist es nicht schwer, das Wörterbuch oder einen Teil davon in einer Datei abzuspeichern.

### 5.9.1 Die Serialisierung von Daten: `serialize`

Mit der Serialisierung eines Datums ist eine Beschreibungsform gemeint, die z.B. 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 uncons
>>   [ -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 ]
>>     \ fct [ drop \ \ \ <non-serializeable-fct> ]
>>     :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 ]
```

### 5.9.2 Ein Schnappschuss des Wörterbuchs: `dump`

Das Wort `dump` serialisiert ein übergebenes Wörterbuch und speichert es im angegebenen Ziel. Das serialisierte Format wird um Code ergänzt, so dass beim Einlesen das Wörterbuch nicht nur rekonstruiert, sondern anschließend mit dem aktuellen Wörterbuch ge`merge`d wird.

```
>> : dump ( dict filename -- )
>>   swap serialize [ get-dict merge set-dict ] concat
>>   repr unbracket-stk-repr swap spit ;
```

Man nennt das Erfassen und das Speichern des aktuellen Programmzustands eines Programmsystems auch einen „Dump erzeugen“. Seien Sie übrigens ein wenig geduldig, das Erzeugen eines Dumps nimmt etwas Zeit in Anspruch.

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. 5.7 vorgestellten syntaktischen 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 `dump` erzeugten Dump der Prelude einzulesen. Dann startet die Consize-VM die Prelude wesentlich schneller.

Wenn Sie mit `get-dict \ prelude-dump.txt dump` einen Dump der Prelude erzeugen, kann das Argument beim Starten der Consize-VM entsprechend angepasst werden:

```
"\ prelude-dump.txt run say-hi"
```

Per Dump ist die Prelude auf meinem Rechner um den Faktor 15-20 mal schneller geladen, als wenn die Prelude als Programmcode im „Klartext“ prozessiert werden muss. Im ersten Fall wird nur das Wörterbuch rekonstruiert, im letzten Fall muss das Wörterbuch erst inkrementell konstruiert werden.

### 5.9.3 Bootstrapping Consize: `bootimage`

Will man Consize mit dem Programmcode der Prelude starten, so kann das nicht gehen: In der Prelude werden syntaktische Wörter benutzt (siehe Kap. 5.7), 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 Dump eines minimalen Wörterbuchs, das all die Wörter enthält, die notwendig sind, um die Verarbeitung der Prelude mit ihren syntaktischen Erweiterungen vorzubereiten.

```
>> : bootstrapping-dict ( -- dict )
>>   [ def def+
>>     cons uncons unpush -rot over
>>     SYMBOL: case when if choose call fcall
>>     scan4] scan4[ parse-quot destruct-definition
>>     : scan4; look4)
>>     read-word read-mapping ]
>>   (  \ [  \ (  \ {  ) concat
```

```
>>   dup [ lookup ] map zip cat mapping ;
```

Mit dem Wörterbuch `bootstrapping-dict` ist leicht mittels `bootimage` ein initialer Dump produziert.

```
>> : bootimage ( -- )
>>   bootstrapping-dict \ bootimage.txt dump ;
```

Ursprünglich habe ich das Bootimage per Hand geschrieben. 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 Arbeit zu großen Teilen ab. Ich muss lediglich darauf achten, dass alle syntaktischen Wörter und die in den Wortdefinitionen verwendeten Wörter in dem Wörterbuch des Bootimages enthalten sind. Die Serialisierung und Speicherung als Datei habe ich schon zu einem sehr frühen Zeitpunkt der Entstehungsgeschichte der Prelude automatisiert.

Dieser Automatismus verschleiert, wie das Bootimage historisch entstanden ist. Ursprünglich war einiges an Handarbeit nötig, 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. 5.7.2 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 gibt es ausschließlich die Wörter der Consize-VM. 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, sprich serialisieren.

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

Das ist exakt der Code, mit dem die Prelude einst begann. Schrittweise kamen andere Wörter hinzu: `def+`, `cons`, `choose`, `if` usw. Die Reihenfolge der Wörter war getrieben von der Notwendigkeit, sich rasch nützliche Hilfsmittel zu schaffen, so dass das Programmieren in Consize zunehmend

angenehmer wurde. Die mit einem Wort assoziierten Quotierungen habe ich per Hand serialisieren müssen.

Ist dieser mühsame Prozess einer initialen Prelude zur Verarbeitung syntaktischer Wörter einmal bewältigt, dann kann die „normale“ Prelude geladen, ein Bootimage erzeugt und der manuelle Vorbereitungsteil verworfen werden. Fortan können nach dem Start des Bootimages Änderungen an der Prelude geladen und eventuell ein neues, aktualisiertes Bootimage generiert werden. Plötzlich ist nicht mehr erkenntlich, was zuerst da war: der Dump oder die Prelude. Man braucht ein Bootimage, um die Prelude zu laden – und eine Prelude, um ein Bootimage zu erzeugen.

Das ist 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` ist es gar der Selbstbezug: ohne `def` kein `def`. Wir Informatiker halten uns nicht lange mit solchen philosophisch anmutenden Fragestellungen auf und durchbrechen per Bootstrapping die Selbstbezüglichkeit. Das Ergebnis: Der Kreislauf von Huhn und Ei ist in Gang gesetzt. Auf eben diese Weise haben wir das Wort `def` aus der Taufe gehoben, um die restlichen Wörter zu definieren. Auch ist `def` nun mit sich selber definierbar. Ist das einmal geschehen, können Sie kess die Frage nach dem „Was war zuerst da?“ stellen, und die Menschheit in Debatten über den Anfang der Dinge verstricken.[4]

## 5.10 Zum Schluß

### 5.10.1 Begrüßung: `say-hi`

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 ] reduce
>>   pop reverse word println
>>   repl ;
```

### 5.10.2 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 „`>>` “ (inkl. Leerzeichen) ausgewiesen sind. Dieses Dokument ist also selbst ein Beispiel für ein „literarisches Programm“. Sie können den Quellcode der Prelude aus der Dokumentation mit dem folgendem Programm aus dem LaTeX-File extrahieren und unter einem neuen Dateinamen abspeichern.

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

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

[4] Sie dürfen davon ausgehen, dass Hühner nicht per Bootstrapping entstanden sind ;-)

# 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 erdenklichen Sequenzen, die mit den Wörtern des Vokabulars $V$ gebildet werden können – das beinhaltet sowohl die leere Sequenz als auch beliebige Verschachtlungen – 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 $\oplus : S^V \times S^V \to S^V$ 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: $f : S^V_\perp \to S^V_\perp$. 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 $; : (S^V_\perp \to S^V_\perp) \times (S^V_\perp \to S^V_\perp) \to (S^V_\perp \to S^V_\perp)$ definiere die Komposition zweier Funktionen $f : S^V_\perp \to S^V_\perp$ und $g : S^V_\perp \to S^V_\perp$ als $f; g : S^V_\perp \to S^V_\perp$, wobei gilt $(f; g)(s) = g(f(s))$ für alle $s \in S^V_\perp$. Das neutrale Element der Funktionskomposition ist die Identitätsfunktion $id : S^V_\perp \to S^V_\perp$ 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 $D : V \to F$ gegeben: Jedes Wort ist eindeutig mit einer Funktion assoziiert. Die Funktion $self : S^V \to (S^V \to S^V)$ sei definiert als $self(s)(s') = [s] \oplus s'$, $s, s' \in S^V$.

Die Denotation $[\![s]\!]$ 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) $[\![[\,]]\!] = id$

(2) $[\![[w]]\!] = f$ für $w \in V$ und $f = D(w)$

(3) $[\![[s]]\!] = self(s)$ für $s \in S^V$.

(4) $[\![s_1 \oplus s_2]\!] = [\![s_1]\!]; [\![s_2]\!]$ 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 $self$ inverse Funktion $self^{-1}$ her. Die Funktion sei definiert über den Zusammenhang

$$(self; self^{-1})(s) = [\![s]\!]$$

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

(5) $[\![[\texttt{call}]]\!] = self^{-1} \Rightarrow [\![[s] \oplus [\texttt{call}]]\!] = [\![s]\!]$

Elementarer kann man eine Programmiersprache kaum mehr formalisieren. Es ist das Minimum dessen, was in Anlehnung an die Kategorientheorie den Formalismus zu einer kartesisch abgeschlossen 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 Consize-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 benamter 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 Consize-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 Consize 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. 33 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 benamte Abstraktionen eingeführt. Unbenamte Quotierungen werden als „anonyme Abstraktionen“ bezeichnet, benamte, sprich über das Wörterbuch assoziierte Quotierungen als „benamte 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 aussagelos. 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.

# Anhang B Patterns and Rewriting Rules

When you write a program in Consize, you are essentially defining just a set of new words. A new word is defined in terms of a sequence of other words. That's it.

While this sounds simple, reading and understanding a program is not. If you go through a program word by word, your brain finds itself easily overloaded by some few stack shuffling operations. Our brains are not perfectly suited for keeping track of what is left where on the datastack.

Take the definition of `each` as an example.

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

The definition is not really complex, but even the stack effect description does not turn out to be helpful. A sequence and a quotation is expected on the datastack, but what exactly is going on here? What is the result on the datastack? If you are somewhat trained, you will see that `each` just drops both the sequence and the quotation if the sequence is empty. But if the sequence is not empty, the behavior of `each` is not self-explanatory.

Our brains are much better at recognizing patterns instead of shuffling data on a stack. In this chapter, we introduce a very simple notation for patterns and learn how to use pattern-based rewriting rules to describe the transformational effects a word has on the datastack as well as on the callstack.

## B.1 Pattern Matching and Instantiation

Let us define pattern matching and pattern instantiation with the help of two imaginary words, `match` and `instantiate`. Both words are not defined in the Prelude, but they could have been. Their existence is not of importance, we use them here to explain what pattern matching and instantiation is and what it means.

The word `match ( stk pat -- matches/f )` expects two stacks on the datastack with `pat` being the pattern the stack `stk` is matched against. The result is a mapping of `matches` or `f`. Let's have a look at an example, assuming that the word `match` would actually be available at the console.

```
> clear [ 1 2 3 4 ] [ #H 2 @T ] match
{ #H 1 @T [ 3 4 ] }
```

The word `match` compares both stacks item by item from left to right (i.e. from top to bottom), nesting included. If the pattern item is a word with a hash symbol (`#`) as its first character, the pattern item is said to match the corresponding data item on the stack. If the pattern item is a word with an at sign (`@`) as its first character, the pattern item matches the rest of the stack. Otherwise, if the pattern item is neither a `#-` nor

an `@`-matcher, the pattern item must match the data item exactly. The resulting mapping is an assembly of matchers and matched values. If something goes wrong, `f` (for *false*) is left on the datastack instead.

In the above example the pattern `[ #H 2 @T ]` only matches successfully (not returning `f`) if the supplied stack contains *at least* two items with the second item being a `2`. A `#`-matcher requires the data item to exist. Here, `#H` matches the very first item, namely `1`, and `@T` matches the rest of the stack which is `[ 3 4 ]`. By convention, the symbols of a word denoting a matcher are written in upper case.

Without matchers, `match` would simply compare for equality and return an empty mapping in case of success or `f` for failure. If there are no matchers in the pattern, the resulting mapping is empty.

```
> clear [ ] [ ] match
{ }
> clear [ 1 2 3 4 ] [ 1 2 3 4 ] match
{ }
> clear [ 1 2 3 4 ] [ 1 2 3 1 ] match
f
> clear [ 1 2 3 4 ] [ 1 2 3 ] match
f
```

Pattern `[ @T ]` matches any stack successfully, even if the matched stack is empty.

```
> clear [ 1 2 3 4 ] [ @T ] match
{ @T [ 1 2 3 4 ] }
> clear [ ] [ @T ] match
{ @T [ ] }
```

The following examples should help you validate your understanding of the concept of matching a pattern against a stack. Note that `#H` demands that there is a data item to match against.

```
> clear [ ] [ #H ] match
f
> clear [ ] [ #H @T ] match
f
> clear [ 1 ] [ #F ] match
{ #F 1 }
> clear [ 1 ] [ #F #S ] match
f
> clear [ 1 ] [ #F @R ] match
{ #F 1 @R [ ] }
```

Usually, `#H` means “head”, `@T` “tail”, and `#F`, `#S`, `@R` typically stand for “first”, “second” and “rest”, respectively.

▷ **Aufgabe B.1** What is the result of the following `match`es?

```
> clear [ 1 [ 2 3 ] 4 ] [ #X #Y #Z ] match
> clear [ 1 [ 2 3 ] 4 ] [ #X @Y ] match
> clear [ 1 [ 2 3 ] 4 ] [ #X #Y #Z #U ] match
> clear [ 1 2 3 ] [ #X #Y #X ] match
> clear [ 1 2 1 ] [ #X #Y #X ] match
```

Patterns with nested stacks enable the extraction of parts of nested stacks.

```
> clear [ [ 1 { 2 3 } 4 5 ] 6 7 ] [ [ #F #S @R ] @T ] match
{ #F 1 #S { 2 3 } @R [ 4 5 ] @T [ 6 7 ] }
> clear [ [ 1 { 2 3 } 4 5 ] 6 7 ] [ #1 @2 ] match
{ #1 [ 1 { 2 3 } 4 5 ] @2 [ 6 7 ] }
```

The opposite of pattern matching is *pattern instantiation*. Taking a mapping and a stack pattern, word `instantiate` replaces `#`- and `@`-matchers in the pattern with their corresponding values and returns the resulting stack. More specifically, a `#`-matcher is replaced by the associated value, a `@`-matcher is replaced by the sequences of values of the associated stack.

```
> clear { #H 1 @T [ 2 3 ] } [ #H @T ] instantiate
[ 1 2 3 ]
```

In the example, `#H` gets replaced by `1` and `@T` by the sequence of associated elements, `2` and `3`.

▷ **Aufgabe B.2** What is the result of the following instantiations?

```
> clear { #X 1 #Y 2 } [ #X 1 #Y 2 ] instantiate
> clear { @T [ 2 3 ] #H 4 } [ #H #H @T ] instantiate
```

## B.2 Rewriting Rules

A rewriting rule consists of two patterns: one pattern describes pattern matching, the other pattern instantiation. Let us call the matching pattern `m-pat` and the instantiation pattern `i-pat`. Rewriting a stack according to a rewriting rule means: first match the stack against `pat-m` and then use the result to instantiate `i-pat`, see the definition of word `rewrite`. If matching or instantiation fails, the result is `f`.

```
: rewrite ( stk m-pat i-pat -- stk'/f )
  [ match ] dip over [ instantiate ] [ drop ] if ;
```

For example, the notion of swapping the two topmost items on a stack can be expressed by the following rewriting rule: `[ #F #S @R ]`, the matching pattern, deconstructs a given stack into a first and second item and remaining items; the instantiation pattern `[ #S #F @R ]` constructs a stack with two items and some remaining items. Observe that in the instantiation pattern `#F` and `#S` are interchanged. A rewriting rule composed of these two patterns does exactly the job of swapping the two topmost elements on a given stack. If the stack does not have enough items rewriting fails.

```
> clear [ x y z u v ] [ #F #S @R ] [ #S #F @R ] rewrite
[ y x z u v ]
> clear [ x ] [ #F #S @R ] [ #S #F @R ] rewrite
f
```

Remember that a continuation captures the state of a computation in Consize and consists of two stacks, the datastack and the callstack.[1] For example, the continuation of the program `1 2 3 swap` just right before interpreting word `swap` reads as

```
[ [ swap ] [ 3 2 1 ] ]
```

[1] A full continuation includes the dictionary as well.

The outer stack serves as a container for the continuation and contains the callstack (being topmost) and the datastack underneath with `3` being the topmost item on the datastack.

The following rule describes rewriting the continuation if and only if `swap` is the topmost element on the callstack. The rewriting rule fails if there are not at least two elements on the datastack of the continuation.

```
[ [ swap @RCS ] [ #Y #X @RDS ] ] [ [ @RCS ] [ #X #Y @RDS ] ]
```

The notation of the rewriting rule is a bit awkward and not very readable. We will agree on interchanging the datastack and the callstack and write them head-to-head with a separating bar (`|`) in between; consequently, the elements of the datastack appear in reverse order. In addition, we include an arrow (`=>`) between the matching and the instantiating pattern. The outer squared brackets are being removed.

```
@RDS #X #Y | swap @RCS => @RDS #Y #X | @RCS
```

Since we assume `@RDS` (rest of datastack) and `@RCS` (rest of callstack) to be present by default, we eliminate their mentioning and indicate so by using a different arrow (`->`). If nothing else effects the callstack, the bar symbol can be omitted on the right hand side of the rule.[2]

```
#X #Y | swap -> #Y #X
```

This style of a rewriting rule for a continuation looks pretty cool and is easy to grasp. Some more examples for some well-known words are:

```
#X | dup -> #X #X
#X | drop ->
#X #Y #Z | rot -> #Y #Z #X
[ @S ] #X | push -> [ #X @S ]
```

A rewriting rule is much more precise than a stack effect description is. We can use rewriting rules in addition to or as a substitute for stack effect descriptions to get a clear understanding of the stack manipulations a words does. Stack rewriting patterns are much more expressive for human readers.

▷ **Aufgabe B.3** Write down the rewriting rules for `over`, `2drop`, `empty?` and `-rot`.

Rewriting rules have another nice feature. We can use them to track the state of a computation when we try to understand a word by resolving and analyzing its definition in a stepwise word-by-word approach. The approach is easy to learn and to perform.

## B.3 Describing and Analyzing Words with Rewriting Rules

Sometimes, the behavior of a word depends on the values or on the structure of some data on the datastack. Different data patterns require different matching patterns and possibly different instantiating patterns. In such cases more than one rewriting rule is needed to properly describe a word's behavior.

[2] If there is no bar symbol on the left hand side (LHS) or right hand side (RHS) of a rewriting rule, assume the bar symbol to be on the outmost left on the LHS and on the outmost right on the RHS. That means, the LHS represents the callstack if the bar is missing, and the RHS represents the datastack if the bar is missing.

Take for example word `top`. Word `top` returns the topmost item of a stack; this is the standard behavior, so to speak. If the stack is empty, `nil` is left of the datastack. And `top` applied on `nil` also returns `nil`. These three alternative behaviors require three rewriting rules.

```
[ #H @T ] | top -> #H
      [ ] | top -> nil
      nil | top -> nil
```

In the same way the following rewriting rules mirror the behavior of word `pop`. The order of rules is important if the pattern matching parts are not exclusive to each other.

```
[ #H @T ] | pop -> @T
      [ ] | pop -> [ ]
```

Using all of the above rewriting rules, one can systematically conclude the rewriting rule(s) of e.g. a word such as `unpush`.

```
: unpush ( stk -- stk' itm ) dup pop swap top ;
```

Even if we would not know the stack effect of `unpush`, a step-by-step analysis of the defining words unveils everything we need to know: The rewriting rule for `dup` tells us that there must be an item on the datastack and that the item is duplicated. The rewriting rules for the following `pop` refine this knowledge: the duplicated item must be a stack, either empty or at least with one item, which determines the result of `pop`. The rewriting rule for `swap` interchanges the two topmost items on the datastack, and the rewriting rule for `top` gives us the two possible results. Thus we conclude the rewriting rules for `unpush` to be

```
      [ ] | unpush ->    [ ] nil
[ #H @T ] | unpush -> [ @T ] #H
```

Although our analysis is informal and *ad hoc*, this example demonstrates how valuable rewriting rules are in understanding the "stack mechanics" of a word. The rewriting rules translate the behavior of `unpush` to a simple mapping of input patterns to output patterns. The rules with their matchers support a very visual style of illustrating the way a word transforms some input to some output. In contrast, the definition of `unpush` is significantly less self-explaining; the stack effect description serves as a reminder on what the word does on the datastack, but the stack effect description is less informative than the rewriting rules are.

The expressivity of rewriting rules includes recursion: at least one rule represents the base case and at least one more rule represents the case of repetition. Take the rewriting rules for reversing the elements of a stack. If the stack is empty, there is nothing to do. Otherwise, the first element of the stack is attached to the end of the reversed stack; here is where recursion comes in.

```
      [ ] | reverse -> [ ]
[ #H @T ] | reverse -> [ @T ] | reverse [ #H ] concat
```

Often, there is more than one way to express a certain behavior. If the pattern extracts the very last element of a stack (an option we rarely make use of), the element must be pushed to the top of the reversed stack.

 

```
      [ ] | reverse -> [ ]
[ @H #T ] | reverse -> [ @H ] | reverse \ #T push
```

Another alternative is defined as follows; we will make use of this very definition of `reverse` later on.

```
      [ ] | reverse -> [ ]
[ @H #T ] | reverse -> [ #T ] [ @H ] | reverse concat
```

▷ **Aufgabe B.4** Try to conclude the rewriting rules for `size` taking the following definition:

```
: size ( stk -- n ) dup empty? [ drop 0 ] [ pop size 1 + ] if ;
```

In an instantiation pattern the combined use of two or more `@`-matchers might be useful, see e.g. `concat` below. In a matching pattern two or more `@`-matchers cause ambiguities. Pattern matching must be unambiguous.

```
[ @S1 ] [ @S2 ] | concat -> [ @S1 @S2 ]
```

▷ **Aufgabe B.5** You have to use pattern `[ #H @T ]` in an alternative set of rewrite rules for `concat`.

## B.4 Examples

### B.4.1 `call/cc`, `continue` and `call`

Since a rewriting rule refers to a continuation, the rules for `call/cc` and `continue` are easy to conclude. Notice that the double arrow `=>` indicates that the patterns for matching and instantiation must address each stack in its entirety.

```
@RDS [ @Q ] | call/cc @RCS => [ @RDS ] [ @RCS ] | @Q
@RDS [ @DS ] [ @CS ] | continue @RCS => @DS | @CS
```

▷ **Aufgabe B.6** Guess what the rewriting rule for `clear` looks like.

In Consize, `call` is defined by a manipulation of the current continuation. The topmost element on the datastack must be a quotation (a stack), which is concatenated with the callstack.

```
: call
  [ swap unpush rot concat continue ] call/cc ;
```

Let us conclude the rewriting rule for `call` by systematically applying the rewriting rules we already have. A prolonged arrow indicates intermediate steps in the chain of reasoning.

```
@RDS [ @Q ] | call @RCS ==>
@RDS [ @Q ] | [ swap unpush rot concat continue ] call/cc @RCS ==>
@RDS [ @Q ] [ swap unpush rot concat continue ] | call/cc @RCS ==>
[ [ @Q ] @RDS ] [ @RCS ] | swap unpush rot concat continue
[ @RCS ] [ [ @Q ] @RDS ] | unpush rot concat continue ==>
[ @RCS ] [ @RDS ] [ @Q ] | rot concat continue ==>
[ @RDS ] [ @Q ] [ @RCS ] | concat continue ==>
[ @RDS ] [ @Q @RCS ] | continue ==>
@RDS | @Q @RCS
```

We can thus conclude that the rewriting rule for `call` is:

```
[ @Q ] | call -> | @Q
```

The above analysis is straight forward and explicates what you would mentally do in order to understand the consequences of stack shuffling and stack manipulations. Rewriting rules formalize this mental process and make it more systematic rather than intuitive.

▷ **Aufgabe B.7** Provide a sound analysis of the rewriting rule for `clear`. Look up the definition for `clear` up and base your analysis on the definition.

▷ **Aufgabe B.8** The backslash word can be defined as follows (as it actually is by the implementation of Consize):

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

Provide a step-by-step analysis of the rewriting rule of `\`.

The exercise shows that any backslashed item will be moved on top of the datastack; so does any backslashed stack, which is – of course – just an item.

```
\ [ @S ] -> [ @S ] |
```

Since any stack will be moved to the datastack by default

```
[ @S ] -> [ @S ] |
```

we can conclude that it does not matter whether a stack is backslashed or not. (Note the bar symbol in the following rewriting rule!)

```
\ [ @S ] -> | [ @S ]
```

▷ **Aufgabe B.9** Did you realize that rewriting rules allow us to formulate most of the internals of the Consize VM? Which words resist being represented as rewriting rules?

### B.4.2 `dip` and `2dip`

The meaning of most combinators is hard to remember, and stack effects are only a poor aid to memory. Rewriting rules come to the rescue. We exemplify this on `dip` and `2dip`.

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

Resolving the definition of `dip` via rewriting rules leads to:

```
#X [ @Q ] | dip -->
#X [ @Q ] | [ ] rot push \ \ push concat call -->
#X [ @Q ] [ ] | rot push \ \ push concat call -->
[ @Q ] [ ] #X | push \ \ push concat call -->
[ @Q ] [ #X ] | \ \ push concat call -->
[ @Q ] [ #X ] \ | push concat call -->
[ @Q ] [ \ #X ] | concat call -->
[ @Q \ #X ] | call -->
| @Q \ #X
```

Word `dip` works almost like `call` with the item in front of the quotation removed from the scope of the call and attached after the call. One dead-simple rule of one line suffices to comprehend the semantics of `dip`.

```
#X [ @Q ] | dip -> | @Q \ #X
```

The definition of `2dip` relies on `dip`. Once you get an understanding of the workings of `dip`, it is not so hard to find out how `2dip` works.

```
: 2dip ( x y quot -- x y ) swap [ dip ] dip ;
```

The resolution of the definition of `2dip` shows that two items in front of the quotation are being "saved" before and "restored" after the call.

```
#X #Y [ @Q ] | 2dip -->
#X #Y [ @Q ] | swap [ dip ] dip -->
#X [ @Q ] #Y | [ dip ] dip -->
#X [ @Q ] #Y [ dip ] | dip -->
#X [ @Q ] | dip \ #Y -->
| @Q \ #X \ #Y
```

During the application of rewriting rules, one risks to mix up with the names of pattern matchers in other rewriting rules – but that is all there is to take care of. To summarize, the rewriting rules for `2dip` is:

```
#X #Y [ @Q ] | 2dip -> | @Q \ #X \ #Y
```

▷ **Aufgabe B.10** Derive the rewriting rules for `keep` and `bi`.

### B.4.3 `t` and `f`, `choose` and `if`

The rules defining the meaning of `t` (*true*) and `f` (*false*) are also strikingly simple. So are the helper words `true` and `false`.

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

Without further ado we can conclude that:

```
#T #F | t -> #T
#T #F | f -> #F
true -> t
false -> f
```

As you might have realized by now, a set of rewriting rules may substitute the dictionary as a storage for the meaning of any kind of word, be it atomic or composite. That is why word `lookup` does not make much sense in the context of rewriting rules: rewriting rules cannot be looked up. Mimicking the existence of a dictionary by a set of rules for `lookup` such as

```
t | lookup -> [ drop ]
f | lookup -> [ swap drop ]
```

points to the absurdity of maintaining a simulated dictionary in addition to rewriting rules. That's not practical at all.

However, the sequence of `lookup` and `call` is unproblematic and can be meaningfully expressed by a single rewriting rule. Moving an item from top of the datastack to the top of the callstack lets us stay inside the system of rewriting rules without referencing the notion of a global dictionary.

```
#W | lookup call -> | #W
```

Now, the definition of `choose` becomes tractable to an analysis with rewriting rules.

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

We first investigate having `f` as the third item on the datastack.

```
f #T #F | choose -->
f #T #F | swap rot false equal? lookup call -->
f #F #T | rot false equal? lookup call -->
#F #T f | false equal? lookup call -->
#F #T f | \ f equal? lookup call -->
#F #T f f | equal? lookup call -->
#F #T t | lookup call -->
#F #T | t -->
#F
```

The investigation of not having `f` but any other item – matched by `#ELSE` – on the third position of the datastack looks very similar.

▷ **Aufgabe B.11** Do the analysis of `choose` for `#ELSE`.

Finally, the two rules for `choose` are:

```
    f #T #F | choose -> #F
#ELSE #T #F | choose -> #T
```

Given `choose` and `call`, we can deduce the rewriting rule for `if`.

```
: if ( f/* then else -- ... ) choose call ;
```

The application and resolution of the rewriting rules is a no-brainer; we can save us the two-step analysis and write down the results immediately.

```
    f [ @T ] [ @F ] | if -> | @F
#ELSE [ @T ] [ @F ] | if -> | @T
```

In case of `f` word `if` `choose`s the "second" quotation and `call`s it; in case of anything `#ELSE`, the "first" quotation is `choose`n and `call`ed.

Whenever there is an `if` in the definition of a word, the `if` can usually be transformed into two rewriting rules. One rule covers the case for *falsity*, the other all remaining cases.

▷ **Aufgabe B.12** Deduce the rewriting rules for `if*`.

### B.4.4 `each` and `map`

Let us come back to the introductory example of word `each`. We have everything in place to analyze how `each` translates to rewriting rules.

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

If the supplied sequence is empty, `each` is done and simply drops both the quotation and the sequence. To save space, the second quotation to `if` is noted as `[ ... ]`.

```
[ ] [ @Q ] | each -->
[ ] [ @Q ] | swap dup empty? [ 2drop ] [ ... ] if -->
[ @Q ] [ ] | dup empty? [ 2drop ] [ ... ] if -->
[ @Q ] [ ] [ ] | empty? [ 2drop ] [ ... ] if -->
[ @Q ] [ ] t | [ 2drop ] [ ... ] if -->
[ @Q ] [ ] | 2drop -->
|
```

If the supplied sequence is not empty, we work with `[ #H @T ]` as the matching pattern, which enforces the stack not to be empty. In the following analysis we skip testing for emptiness and jump right into the `if`-case for *false*. Remember that for any backslashed stack on the callstack the backslash can be removed, see exercise B.8.

```
[ #H @T ] [ @Q ] | each -->
[ @Q ] [ #H @T ] | unpush -rot over [ call ] 2dip each -->
[ @Q ] [ @T ] #H | -rot over [ call ] 2dip each -->
#H [ @Q ] [ @T ] | over [ call ] 2dip each -->
#H [ @Q ] [ @T ] [ @Q ] | [ call ] 2dip each -->
#H [ @Q ] | call \ [ @T ] \ [ @Q ] each -->
#H | @Q [ @T ] [ @Q ] each
```

What `each` does is maybe hard to analyze but easy to describe with patterns: `each` takes one element after another from a given sequence of elements and calls the given quotation on each of the elements.

```
      [ ] [ @Q ] | each -> |
[ #H @T ] [ @Q ] | each -> #H | @Q [ @T ] [ @Q ] each
```

We will finish our examples with `map`. Word `map` is defined in terms of `each`.

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

The rewriting rule up to `each` is easy to derive in your head, which is enough to understand the semantics of `map`: Word `map` applies the quotation `[ @Q push ]` on `each` element of sequence `[ @S ]`; the extra `push` ensures that the result of calling `[ @Q ]` on each element is moved on a result collecting stack, which is initially empty. Since pushing elements on the result stack does reverse the order of the results, `reverse` restores the order.

```
[ @S ] [ @Q ] | map -> [ ] [ @S ] [ @Q push ] | each reverse
```

Note that the correct functioning of `map` requires that the called quotation `[ @Q ]` consumes exactly one element from the datastack and leaves exactly one element on the datastack. If this assumption is not fulfilled, the semantics and the execution of `map` gets corrupted.

It is possible to formalize this constraint using the following notation, saying: The expression on the left hand side (LHS) is equivalent (`==`) to the expression on the right hand side (RHS). The expression on the RHS matches (and must match, that is the constraint) the result of resolving the expression on the LHS.

 

```
#X | @Q == #Y
```

Things get interesting and advanced, if we apply the rewriting rules for `each` and `reverse` so that our explanation of the behavior of `map` does depend neither on `each` nor on `reverse`.

If `[ @S ]` matches an empty stack, the terminating behavior of `map` is three derivation steps away.

```
[ ] [ @Q ] | map -->
[ ] [ ] [ @Q push ] | each reverse -->
[ ] | reverse -->
[ ]
```

If `[ @S ]` matches a non-empty stack, we use `[ #H @T ]` instead.

```
[ #H @T ] [ @Q ] | map -->
[ ] [ #H @T ] [ @Q push ] | each reverse -->
[ ] #H | @Q push [ @T ] [ @Q push ] each reverse -->
(to be continued)
```

For `[ ] #H | @Q push` we know it to be a stack with a single element, say `[ #R ]`, because of the above mentioned constraint. Therefore, we could – just to make things clearer – also write for the last rule:

```
[ #R ] | [ @T ] [ @Q push ] each reverse -->
```

We also know that `[ @T ] [ @Q push ] each` will continue to push elements on top of `[ #R ]`. In other words, `[ #R ]` is the *last* element of a growing stack that is to be `reverse`d after all. It takes some thinking to see that the rules for `reverse` open up a nice "trick".

```
      [ ] | reverse -> [ ]
[ @H #T ] | reverse -> [ #T ] [ @H ] | reverse concat
```

If `[ @T ] [ @Q push ] each` can be managed to be a stack on its own, so that it can represent `[ @H ]` in the second rewriting rule of `reverse`, we have the means to resolve `reverse` once. The introduction of an empty stack does the trick.

```
[ #R ] | [ @T ] [ @Q push ] each reverse -->
[ #R ] | [ ] [ @T ] [ @Q push ] each reverse concat -->
(to be continued)
```

Recognize that a part of the rewriting expression looks familiar: it is a map expression!

```
[ #R ] | [ ] [ @T ] [ @Q push ] each reverse concat -->
[ #R ] | [ @T ] [ @Q ] map concat
```

To summarize the rules for `map`: Word `map` applies quotation `[ @Q ]` to each element of a given stack and assembles the results in a stack.

```
      [ ] [ @Q ] | map -> [ ]
[ #H @T ] [ @Q ] | map -> [ ] #H | @Q push [ @T ] [ @Q ] map concat
```

We might use a feature of Consize, namely parentheses, that resembles much better the notion of `[ #R ]` than the above expression does.

```
      [ ] [ @Q ] | map -> [ ]
[ #H @T ] [ @Q ] | map -> | ( \ #H @Q ) [ @T ] [ @Q ] map concat
```

The analysis for `map` is challenging. But that is due to the formal nature of rewriting rules. The approach not only feels mathematical, it actually is mathematical. Computing is formal and requires analytical skills, sharp thinking and sometimes an creative insight on how to proceed in the chain of reasoning. It can be quite hard to perform a solid analysis. Occasionally, it is easier to "guess" the rewriting rule instead of deriving it from a word's definition.

▷ **Aufgabe B.13** What are the rewriting rules for `reduce`? Please resolve `each` as well and formulate the constraint that applies to `reduce`.

▷ **Aufgabe B.14** Word `reduce` is a very powerful and expressive word, so it is important to get acquainted with it. Define the following words using `reduce`; no conditionals like `if` are required! Hint: This exercise is not about rewriting rules but word definitions.

- `sum ( stk -- n )` takes a stack of numbers and returns the sum of the numbers.
- `my-size ( stk -- n )` takes a stack and returns the number of elements of the stack. Note: `size` is already defined in Consize; the challenge is to find an alternative implementation using `reduce`.
- `my-reverse ( stk -- stk' )` takes a stack of items and returns a stack with the items in reversed order. Note: `reduce` is an atomic words in Consize for performance reasons. Find an implementation using `reduce`.

In addition, write some unit tests to verify correct behavior of your definitions. Do not forget to consider processing an empty stack.

▷ **Aufgabe B.15** Challenge: Define word `my-concat` (which is behaviorally equivalent to `concat`) using *only* stacks and the words `swap`, `push` and `reduce`. Write some unit tests to verify correct behavior of your definition.

▷ **Aufgabe B.16** What are the rewriting rules for `filter`?

## B.5 Closing Remarks

Rewriting rules help you quite much in understanding concatenative programs. You might ask, why we do not define words with rewriting rules, which looks and feels simpler. Why is a word in Consize always defined in terms of other words besides atomic ones?

As a matter of fact, many programming languages, mostly functional and logic programming languages, offer some kind of pattern matching for a good reason: functions or relations become much more readable and understandable.

However, in a language like Consize, rewriting rules for word definitions do more harm than good. It would mess up the design philosophy of a concatenative language. In a concatenative language, quotations (representing programs) are built up from smaller quotations by concatenation. The smallest fragment of a program is a quotation which consists of a single word or data item. Semantically, quotations correspond to functions and concatenation to function composition.

The notion of named and unnamed abstractions builds upon the idea of concatenation and function composition, respectively, which implies a

strictly layered program design and scales seamlessly from atomic words up to the architectural level; it features refactoring and favors the use and detection of design patterns in a way which is unparalleled in the world of programming languages. Rewriting rules do not share these properties and do not blend in a concatenative language.

Take the rewriting rules for `map` and `each` as an example. Looking at the rules, both words seem to be completely unrelated. Considering the word definitions in Consize, the relationship between `map` and `each` is so obvious that you cannot ignore it; it tells you much more about inner dependencies of words than rewriting rules ever could do. In Consize, you design even tiny programs very much like you would do on an architectural level. Rewriting rules, in contrast, focus on input/output relationships rather than on compositional aspects.

To conclude: Concatenative languages are great for their compositional style of programming and somewhat bad at disclosing input/output relationships in terms of stack effects. Rewriting rules excel at revealing stack effects (even on the level of continuations) but do not enforce a compositional style of programming.

We could combine the best of both worlds and use rewriting rules as a formal and superior notation for stack effect descriptions. That means that you are writing word definitions twice: on the one hand as a word made of a sequence of other words and items; on the other hand as a set of rewriting rules to lay out stack effects. Such an approach supports a very consistent and sound regimen on software engineering. Some machine assisted help could support the programmer in deducing the rewriting rules or the word definition.

Some historic remarks: In their bachelor thesis, two of my students, Florian Eitel and Aaron Müller, implemented an interpreter for rewriting rules in Ruby. Their work was awarded by the Thomas Gessmann foundation. Another student of mine, Tim Reichert, was so much inspired by the pattern system outlined in this chapter that he continued to develop it further and extend it even to meta patterns. His impressive work is documented in his PhD thesis.[3]

[3] Reichert, Tim (2011): *A Pattern-based Foundation for Language-Driven Software Engineering*, Doctoral thesis, Northumbria University, Newcastle (UK)

## B.6 Solutions

Exercise B.1 You cannot really know but you might have guessed correctly that in the last two cases `#X` demands the very same value to appear anywhere `#X` asks for a match.

```
> clear [ 1 [ 2 3 ] 4 ] [ #X #Y #Z ] match
{ #X 1 #Y [ 2 3 ] #Z 4 }
> clear [ 1 [ 2 3 ] 4 ] [ #X @Y ] match
{ #X 1 @Y [ [ 2 3 ] 4 ] }
> clear [ 1 [ 2 3 ] 4 ] [ #X #Y #Z #U ] match
f
> clear [ 1 2 3 ] [ #X #Y #X ] match
f
> clear [ 1 2 1 ] [ #X #Y #X ] match
{ #X 1 #Y 2 }
```

Exercise B.2 The solutions are

```
> clear { #X 1 #Y 2 } [ #X 1 #Y 2 ] instantiate
[ 1 1 2 2 ]
> clear { @T [ 2 3 ] #H 4 } [ #H #H @T ] instantiate
[ 4 4 2 3 ]
```

Exercise B.3 The rewriting rules almost match the stack effect descriptions.

```
#X #Y | over -> #X #Y #X
#X #Y | 2drop -> |
  [ ] | empty? -> t
#ELSE | empty? -> f
#X #Y #Z | -rot -> #Z #X #Y
```

Exercise B.4 The solution is

```
      [ ] | size ->        | 0
[ #H @T ] | size -> [ @T ] | size 1 +
```

Exercise B.5 The first solution is almost a no-brainer: move the very first item of the topmost stack to the end of the second stack.

```
[ @S ] [ ] | concat -> [ @S ]
[ @S ] [ #H @T ] | concat -> [ @S #H ] [ @T ] concat
```

The second solution shows how elements from the leftmost stack get pushed to the rightmost stack pertaining the order of elements.

```
[ ] [ @S ] | concat -> [ @S ]
[ #H @T ] [ @S ] | concat -> [ @T ] [ @S ] | concat \ #H push
```

Exercise B.6 You need a continuation pattern to express what `clear` does. It empties the datastack.

```
@RDS | clear @RCS => | @RCS
```

Exercise B.7 The definition of word `clear` relies on word `set-ds`.

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

@RDS | clear @RCS ==>
@RDS | [ ] set-ds @RCS ==>
```

```
@RDS [ ] | set-ds @RCS ==>
@RDS [ ] | [ swap top swap continue ] call/cc @RCS ==>
@RDS [ ] [ swap top swap continue ] | call/cc @RCS ==>
[ [ ] @RDS ] [ @RCS ] | swap top swap continue ==>
[ @RCS ] [ [ ] @RDS ] | top swap continue ==>
[ @RCS ] [ ] | swap continue ==>
[ ] [ @RCS ] | continue ==>
| @RCS
```

Exercise B.8 In the analysis, sometimes two words are rewritten at once.

```
@RDS | \ #I @RCS ==>
@RDS | [ dup top rot swap push swap pop continue ] call/cc #I @RCS ==>
@RDS [ dup top rot swap push swap pop continue ] | call/cc #I @RCS ==>
[ @RDS ] [ #I @RCS ] | dup top rot swap push swap pop continue ==>
[ @RDS ] [ #I @RCS ] #I | rot swap push swap pop continue ==>
[ #I @RCS ] #I [ @RDS ] | swap push swap pop continue ==>
[ #I @RCS ] [ #I @RDS ] | swap pop continue ==>
[ #I @RDS ] [ @RCS ] | continue ==>
@RDS #I | @RCS
```

Exercise B.9 The rewriting rules refer to stack transformations only. We have not introduced adequate means to rewrite mappings or to rewrite words, I/O activities etc. The scope of rewrite rules is limited and involves only stack shufflers, stack words and continuations.

Exercise B.10 The solutions according to the following definitions are:

```
: keep  ( x quot -- x ) [ dup ] dip dip ;
: bi ( x p q -- ) [ keep ] dip call ;
```

```
#X [ @Q ] | keep -->
#X [ @Q ] | [ dup ] dip dip -->
#X [ @Q ] [ dup ] | dip dip -->
#X | dup \ [ @Q ] dip -->
#X #X | \ [ @Q ] dip -->
#X #X [ @Q ] | dip -->
#X | @Q \ #X
```

```
#X [ @P ] [ @Q ] | bi -->
#X [ @P ] [ @Q ] | [ keep ] dip call -->
#X [ @P ] [ @Q ] [ keep ] | dip call -->
#X [ @P ] | keep \ [ @Q ] call -->
#X | @P \ #X [ @Q ] call -->
#X | @P \ #X @Q
```

Exercise B.11 No solution provided. I think you can do the analysis yourself, don't you?!

Exercise B.12 The assumption is that you can derive the rewriting rules for `pick` and `2nip` yourself and use them in the process of concluding the rewriting rules for `if*`.

```
: if* ( f/* [ @T ] [ @F ] )
  pick [ drop call ] [ 2nip call ] if ;
```

```
f [ @T ] [ @F ] | if* -->
f [ @T ] [ @F ] | pick [ drop call ] [ 2nip call ] if -->
f [ @T ] [ @F ] f [ drop call ] [ 2nip call ] | if -->
f [ @T ] [ @F ] | 2nip call -->
[ @F ] | call -->
```

```
| @F

#ELSE [ @T ] [ @F ] | if* -->
#ELSE [ @T ] [ @F ] | pick [ drop call ] [ 2nip call ] if -->
#ELSE [ @T ] [ @F ] #ELSE [ drop call ] [ 2nip call ] | if -->
#ELSE [ @T ] [ @F ] | drop call -->
#ELSE [ @T ] | call -->
#ELSE | @T
```

In contrast to `if`, word `if*` keeps the value for "truth".

Exercise B.13 The definition of `reduce` is suprinsingly close to `each`.

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

[ @S ] #I [ @Q ] | reduce -->
[ @S ] #I [ @Q ] | [ swap ] dip each -->
[ @S ] #I [ @Q ] [ swap ] | dip each -->
#I [ @S ] [ @Q ] | each
```

What is the purpose of item `#I`? Why does it matter? The resolution of `each` helps getting the point.

```
      [ ] [ @Q ] | each -> |
[ #H @T ] [ @Q ] | each -> #H | @Q [ @T ] [ @Q ] each

      [ ] #I [ @Q ] | reduce --> |
[ #H @T ] #I [ @Q ] | reduce --> #I #H | @Q [ @T ] [ @Q ] each
```

Word `reduce` is only meaningful if `@Q` processes two items on the datastack, namely `#I` and `#H`, and leaves exactly one item on the datastack. This constraint can be expressed by

```
#X #Y | @Q == #Z
```

Item `#I` serves as a sort of accumulator; `reduce` gets called with an initial value for the accumulator and continues to be updated with each turn of `each`.

Exercise B.14 The solutions are:

```
: sum ( stk -- n ) 0 [ + ] reduce ;
: my-size ( stk -- n ) 0 [ drop 1 + ] reduce ;
: my-reverse ( stk -- stk' ) [ ] [ push ] reduce ;
```

Here is a proposal for a minimal set of unit tests: the base case (empty stack) and some other scenario is tested.

```
[ 0 ] [ [ ] sum ] unit-test
[ 10 ] [ [ 1 2 3 4 ] sum ] unit-test
[ 0 ] [ [ ] my-size ] unit-test
[ 4 ] [ [ x x x x ] my-size ] unit-test
[ [ ] ] [ [ ] my-reverse ] unit-test
[ [ 3 2 1 ] ] [ [ 1 2 3 ] my-reverse ] unit-test
```

Exercise B.15 The key point is using `reduce` in the definition of `my-concat`.

```
: my-concat ( stk1 stk2 -- stk3 )
  swap reverse [ push ] reduce reverse ;
```

Now, resolve word `reduce` (see `my-reduce` in exercise B.14) and you are done.

```
: my-concat ( stk1 stk2 -- stk3 )
  swap [ ] [ push ] reduce [ push ] reduce [ ] [ push ] reduce ;
```

The following four types of unit tests are a must!

```
[ [ 1 2 3 4 5 6 7 ] ] [ [ 1 2 3 ] [ 4 5 6 7 ] my-concat ] unit-test
[ [ 1 2 3 ] ] [ [ 1 2 3 ] [ ] my-concat ] unit-test
[ [ 4 5 6 7 ] ] [ [ ] [ 4 5 6 7 ] my-concat ] unit-test
[ [ ] ] [ [ ] [ ] my-concat ] unit-test
```

Exercise B.16 The definition of `filter` adds some code to `quot` and then runs `each` and `reverse`.

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

From the analysis experience you have you might see that

```
[ ] [ @Q ] | filter --> [ ]
```

It is not much meaningful to go too much into depth with the standard case.

```
[ #H @T ] [ @Q ] | filter -->
[ ] [ #H @T ] [ dup @Q [ push ] [ drop ] if ] | each reverse
```

This rule looks very much like a resolved `map`; in fact, it works like a `map` for all elements in the given sequence for which `#H | @Q true and == t`. The elements which do not satisfy the predicate are dropped and not included in the resulting sequence.