Einführung in die MIDI-Programmierung mit Q

Albert Gräf
Bereich Musikinformatik, Musikwissenschaftliches Institut
Johannes Gutenberg-Universität Mainz

Februar 2003

Vorwort

Die vorliegende kleine Einführung in die MIDI-Programmierung mit der Programmiersprache Q entstand als begleitendes Skript zur Übung „MIDI-Programmierung“ im Wintersemester 2002/03 an der Abteilung Musikinformatik des Musikwissenschaftlichen Instituts der Johannes Gutenberg-Universität Mainz.

Die symbolische Verarbeitung musikalischer Daten mit dem Computer hat, gemessen an der Geschwindigkeit, mit der uns neue Entwicklungen in der Computer-Technik überrollen, bereits eine lange Tradition. Die Entwicklung der MIDI-Technik („Music Instruments Digital Interface“) in den 1980ern war dabei ein wichtiger Meilenstein. MIDI-Programmierung ist nicht nur ein wesentliches Handwerkszeug des Komponisten, Musikers oder Musikwissenschaftlers, der komplexe Anwendungen z.B. zur algorithmischen Komposition oder musikalischen Analyse mit dem Computer selbst erstellen möchte. Sie eröffnet auch einen Einblick in die Anwendung Computer-wissenschaftlicher Prinzipien und Verfahren bei der Programmierung von Computer-Musik-Applikationen.

Im Gegensatz zur meisten existierenden Literatur wird für diese Einführung nicht eine konventionelle Programmiersprache wie z.B. C zu Grunde gelegt, sondern die funktionale Programmiersprache Q. Dies hat den Nachteil, dass konventionelle Programmiersprachen auch heute noch wesentlich weiter verbreitet sind. Ein Trend zur Anwendung funktionaler Programmiersprachen im Bereich der Computer-Musik ist aber bereits deutlich spürbar; es ist nur natürlich, dass diese auch in der Forschung und Lehre eine zunehmende Rolle spielen. Die Nutzung einer modernen funktionalen Sprache ermöglicht die Software-Entwicklung auf einem wesentlichen höheren Abstraktions-Niveau, so dass man sich nicht mit der Vielzahl unwesentlicher technischer Details auseinandersetzen muss, die die MIDI-Programmierung in C erheblich komplizieren. Dass die Wahl dabei auf die Programmiersprache Q, eine Entwicklung des Autors, gefallen ist, liegt vor allem daran, dass andere funktionale Sprachen zur Zeit noch über keine MIDI-Schnittstelle verfügen, mit der die Entwicklung von MIDI-Echtzeit-Anwendungen auf einfache und portable Weise möglich ist. Q hat aber zu den anderen modernen funktionalen Sprachen wie Haskell und ML große Ähnlichkeiten, so dass sich die hier diskutierten Beispiele sicher ohne größere Mühen übertragen lassen werden.

Hinweis: Um die in dieser Einführung vorgestellten Beispiele selbst am PC nachvollziehen zu können, benötigen Sie eine Installation der Q-Programmierumgebung auf Ihrem Computer. Die Q-Midi-Schnittstelle ist sowohl für Linux- als auch Windows-Systeme einsetzbar, eine Portierung auf das Betriebssystem Mac OSX der Firma Apple soll demnächst auch zur Verfügung stehen. Die Einführung bezieht sich vor allem auf das Linux-System, das im Rahmen der Lehrveranstaltung verwendet wurde; im Verlauf des Textes werden aber auch Hinweise für Windows-Benutzer gegeben. Die vorgeführten Beispiele sollten unverändert sowohl unter Linux als auch unter Windows funktionieren. Weitergehende Hinweise zur Einrichtung der Q-Programmierumgebung und der Q-Midi-Schnittstelle entnehmen Sie bitte dem Anhang.

Inhaltsverzeichnis

Teil I: Einführung

1 MIDI-Grundlagen....1
1.1 Was ist MIDI?....1
1.2 Kurze MIDI-Geschichte....2
1.3 Das MIDI-Format....3
2 Grundlagen Programmierung....5
2.1 Warum MIDI-Programmierung?....5
2.2 MIDI-Programmierung mit Q....6
2.3 Ein einfaches Q-Midi-Beispiel....7
2.4 Q-Midi Player....7
3 Einführung in Q....9
3.1 Erstellung eines Skripts....9
3.2 Anatomie eines Skripts....11
3.3 Deklarationen....12
3.4 Ausdrücke....13
Einfache Ausdrücke....13
Zusammengesetzte Ausdrücke....14
3.5 Vordefinierte Operatoren und Funktionen....17
Standard-Bibliotheks-Funktionen....19
3.6 Gleichungen....22
Rekursion und Iteration....24
Definition von Listen-Funktionen....25
3.7 Auswertung von Ausdrücken....27
3.8 Variablen-Definitionen....30
Lokale Variablen....31
3.9 Datentypen....33
Eingebaute Datentypen....34
Benutzer-definierte Datentypen....35
3.10 Parallel-Verarbeitung....36

Teil II: MIDI-Programmierung mit Q

4 Grundlegendes....39
4.1 Registrieren eines MidiShare-Clients....39
4.2 Clients für die MIDI-Ein- und Ausgabe....41
4.3 Herstellen von Verbindungen zwischen Clients....42
4.4 MIDI-Ein- und Ausgabe....43
MIDI-Eingabe....43
MIDI-Ausgabe....44
4.5 Filterfunktionen....45
5 MIDI-Nachrichten und -Ereignisse....47
5.1 Nachrichten-Kategorien....47
5.2 Noten....47
5.3 Instrumentierung und Controller-Nachrichten....48
5.4 Weitere Voice-Nachrichten....48

5.5 System-Common-Nachrichten....49
5.6 System-Realtime-Nachrichten....49
5.7 Kodierung von MIDI-Ereignissen....50
6 Echtzeit-Verarbeitung....51
6.1 Grundform eines Echtzeit-Programms....51
6.2 Echtzeit-Programme mit Gedächtnis....53
7 Sequencing....57
7.1 Aufnahme....57
7.2 Wiedergabe....58
7.3 Gleichzeitige Aufnahme und Wiedergabe....62
7.4 Mehrspurige Sequenzen....65
8 MIDI-Dateien....69
8.1 Musikalische Zeit vs. Computer-Zeit....69
8.2 Meta-Nachrichten....70
8.3 Einlesen von MIDI-Dateien....71
8.4 Speichern von MIDI-Dateien....76
8.5 Bearbeiten von MIDI-Dateien....79

Anhang

A Installationshinweise....83

Teil I: Einführung

1 MIDI-Grundlagen

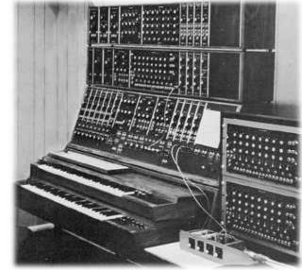
1.1 Was ist MIDI?

MIDI („Musical Instruments Digital Interface") wurde in den 1980ern von verschiedenen amerikanischen und japanischen Herstellern als standardisierte Schnittstelle zur Steuerung und Kopplung von Synthesizern und Klangmodulen entwickelt. Der MIDI-Standard [http://www.midi.org] spezifiziert eigentlich drei verschiedene Aspekte der Schnittstelle:

- Eine *Hardware-Schnittstelle* zur Kommunikation zwischen verschiedenen MIDI-Geräten. Dabei handelt es sich um eine Art serielle Schnittstelle mit einer Datenübertragungsrate von 31250 bps (Bits pro Sekunde). Synthesizer und andere MIDI-Geräte werden üblicherweise mittels 5-poliger DIN-Buchsen verbunden. Die meisten MIDI-Geräte verfügen über drei solche Buchsen für MIDI Input, MIDI Output und (optional) MIDI „Thru". Letztere ermöglicht die Weitergabe empfangener MIDI-Daten. Damit können Ketten von MIDI-Geräten gebildet werden, die alle mit dem gleichen MIDI-Datenstrom arbeiten, wie z.B. verschiedene Synthesizer, Drum-Boxes etc., die von einem MIDI-Keyboard aus gesteuert werden.
- Ein *Kommunikationsprotokoll*, das die Syntax und Semantik der verschiedenen Typen von *MIDI-Nachrichten* spezifiziert, mit denen MIDI-Geräte gesteuert werden können. Die wichtigsten Nachrichten sind jene, mit denen Noten begonnen und beendet werden, so genannte „Note On"- und „Note Off"-Nachrichten. Diese Nachrichten werden z.B. von MIDI-Keyboards gesendet, wenn eine Taste angeschlagen bzw. losgelassen wird. Mit anderen Typen von Nachrichten kann der gewählte Instrumenten-Klang und der Status verschiedener Kontrollelemente verändert werden. Diese Nachrichten korrespondieren typischerweise mit Knöpfen, Schiebereglern u.ä. auf einem MIDI-Instrument.
- Ein (binäres) *Dateiformat* zur permanten Speicherung von Folgen („Sequenzen") von *MIDI-Ereignissen*, d.h. mit einem Zeitstempel versehenen MIDI-Nachrichten. Die Zeitstempel, die in einer MIDI-Datei als Zeitdifferenzen (so genannte „Delta"-Werte) gespeichert werden, geben an, zu welchem Zeitpunkt die Wiedergabe eines Ereignisses beim Abspielen einer MIDI-Datei erfolgen soll. Eine MIDI-Datei kann mehrere Sequenzen (so genannte „Spuren") enthalten. Tatsächlich gibt drei verschiedene Typen von MIDI-Dateien: „Typ 0", mit dem eine einzelne Sequenz gespeichert wird; „Typ 1" zur Speicherung eines einzelnen Stückes, das aus mehreren Spuren besteht; und „Typ 2" zur Speicherung mehrerer Stücke (jeweils eines in jeder Spur). Neben den üblichen Typen von MIDI-Nachrichten enthalten MIDI-Dateien auch so genannte „Meta"-Ereignisse, z.B. Tonart-, Tempo- und Metrum-Angaben, Spur-Beschriftungen u.ä.

WICHTIG: Im Unterschied zu digitalem Audio (z.B. WAV oder MP3) enthalten MIDI-Datenströme *keine* tatsächlichen Klänge – nur die zur Steuerung eines Klangmoduls notwendigen Kontrollinformationen werden übertragen. Daher benötigt MIDI erheblich weniger Bandbreite als die Klänge, die daraus in einem angeschlossenen Synthesizer erzeugt werden. Dadurch wird es möglich, MIDI-Ströme programmgesteuert zu analysieren, zu modifizieren und zu synthetisieren, sogar in „Echtzeit". Mögliche Anwendungen werden in Kapitel 2 skizziert.

1.2 Kurze MIDI-Geschichte



Vor MIDI

1965: Erster kommerziell verfügbarer Moog-Synthesizer

1960er, 70er: CV-basierte Analog-Synthesizer

Ende 1970er: erste digitale Synthesizer

Wichtige MIDI-Meilensteine

Juni 1981: Treffen von I. Kakehashi (Roland Corporation), Tom Oberheim (Oberheim Electronics) und Dave Smith (Sequential Circuits) bei der Trade Show der National Association of Music Merchants (NAMM)

November 1981: Dave Smith stellt ersten Entwurf des so genannten „Universal Synthesizer Interface“(USI) bei der Audio Engineers Society vor

Januar 1982: Japanische Hersteller (Korg, Kawai, Yamaha) schließen sich der Kooperation an

Juni 1982: Grundlegende Elemente der MIDI-Spezifikation werden bei der NAMM vorgestellt

1982-83: erste Implementierungen

August 1983: Veröffentlichung des MIDI 1.0 Standards

1985: MIDI ist de facto Industriestandard

1991: General MIDI (GM) Standard, MIDI Tuning Standard

Vor MIDI konnten Synthesizer verschiedener Hersteller nur unter großem technischen Aufwand miteinander verbunden werden. Mit MIDI ist dies nun auf einfache Weise möglich. Darüber hinaus verfügen MIDI-Geräte auch über Möglichkeiten zur digitalen Speicherung von Instrumenten-klängen und Sequenzen; die aufwendige „Programmierung“ der analogen Synthesizer mittels „Patches“ entfällt. Tatsächlich können analoge Synthesizer (deren Klänge sich heute wieder zunehmender Beliebtheit erfreuen) mittlerweile vollständig in Software durch MIDI-Synthesizer und PC-Programme simuliert werden. MIDI hat sich wegen seiner vielen Vorteile als sehr erfolg-reich erwiesen und bereits in den 80er Jahren große Verbreitung gefunden. Heute kommt praktisch kein neuer Synthesizer mehr auf den Markt, der nicht MIDI-kompatibel ist.

MIDI ist ein kommerzieller Standard, der von zwei Organisationen, der „MIDI Manufacturer's Association“ (MMA) [http://www.midi.org] und ihrem japanischen Gegenstück, dem „Japan MIDI Standards Committee“, gehütet wird. Alle Änderungen des Standards benötigen die Zustimmung beider Organisationen.

Quelle: Kristopher D. Giesing: A Brief History of MIDI.
http://ccrma-www.stanford.edu/~kgiesing/Midi/ (letzter Zugriff: 11.9.2002).

1.3 Das MIDI-Format

MIDI-Nachrichten setzen sich aus einem oder mehreren Bytes zusammen: ein so genanntes *Status*-Byte (d.h. Befehls-Byte) gefolgt von einer Folge von *Daten*-Bytes. Bis auf die so genannten *Sysex*-Nachrichten, die eine variable Anzahl von Daten-Bytes haben können, haben alle Typen von MIDI-Nachrichten stets 0, 1, 2 oder 3 Daten-Bytes, je nach Befehlstyp. Status- und Daten-Bytes werden danach unterschieden, ob das höchste Bit gesetzt ist. Status-Bytes haben stets Werte zwischen (hexadezimal) 0x80 und 0xFF (dezimal 128 bis 255), Daten-Bytes Werte zwischen 0x00 und 0x7F (dezimal 0 bis 127).

Exkurs: Bits und Bytes

Zur Darstellung von MIDI-Befehlen verwendet man meist *Hexadezimalzahlen*, bei der die Byte-Werte in Basis 16 mit den Ziffern 0 bis 9 und A bis F ausgedrückt werden. Zur besonderen Kennzeichnung schreiben wir Hexadezimalzahlen mit der Präfix 0x. In der Hexadezimal-Schreibweise lassen sich leicht die Bestandteile eines MIDI-Befehls, insbesondere des Status-Bytes erkennen:

- 8 Bit = 1 Byte = 2 Hexadezimalziffern. Das *Byte* ist die kleinste adressierbare Speichereinheit im Computer, sie umfasst 256 Werte von 0x00 bis 0xFF (dezimal: 0 bis 255). Ein *Bit* ist die kleinste Informationseinheit im Computer, entsprechend einer einzigen Binärziffer (0 oder 1 = „aus" oder „an").
- 4 Bit = 1 Nibble = 1 Hexadezimalziffer. Das *Nibble* ist ein „halbes Byte" mit Werten im Bereich 0x0 bis 0xF (0 bis 15). Innerhalb eines Bytes unterscheidet man das Lo-Nibble (Bits 0 bis 3) und das Hi-Nibble (Bits 4 bis 7). Z.B.: Byte-Wert = 0xA5 ⇒ Hi-Nibble = 0xA, Lo-Nibble = 0x5.

Man unterscheidet so genannte „Voice"- und „System"-Nachrichten. Erstere sind für einen bestimmten *MIDI-Kanal* bestimmt, während letztere für alle angeschlossenen MIDI-Module gelten. MIDI unterstützt 16 Kanäle (0 bis 15). Angeschlossene Geräte können je nach Bedarf auf bestimmte Kanäle eingestellt werden, so dass sie nur Voice-Nachrichten des jeweiligen Kanals bearbeiten bzw. erzeugen. Außerdem verfügt auf so genannten *multitimbralen Synthesizern* jeder MIDI-Kanal über seine eigenen Instrumenten- und Controller-Einstellungen.

Das Status-Byte von Voice-Nachrichten setzt sich zusammen aus einem Befehlscode im Hi-Nibble und der Kanalnummer im Lo-Nibble. Folgende Typen von Voice-Nachrichten sind vorhanden:

Status-Byte[1]	Beispiel	Bedeutung
0x8*n*	0x80 0x3C 0x40	„Note Off" auf Kanal 0, Note #60 (Mittel-C), Stärke 64
0x9*n*	0x92 0x3C 0x40	„Note On" auf Kanal 2, Note #60, Stärke 64
0xA*n*	0xA5 0x3C 0x7F	„Key Pressure" auf Kanal 5, Note #60, Wert 127
0xB*n*	0xB0 0x07 0x7F	„Control Change" auf Kanal 0, Controller #7 (Volume/Coarse), Wert 127
0xC*n*	0xC0 0x05	„Program Change" auf Kanal 0, Instrument #5 (GM-Standard: Electric Piano 1)
0xD*n*	0xD0 0x7F	„Channel Pressure" auf Kanal 0, Wert 127
0xE*n*	0xE0 0x00 0x40	„Pitch Wheel" auf Kanal 0, Wert 0x2000 = 8192 (Mittelstellung)

1 *n* = Kanalnummer

Status-Bytes im Bereich zwischen 0xF0 und 0xFF kennzeichnen System-Befehle, die weiter in die so genannten „System Common"- und „System Realtime"-Befehle unterteilt werden. Auf letztere Befehle sollen MIDI-Geräte sofort, in Echtzeit, reagieren. Zwei wichtige System-Befehle sind:

- *Sysex* („System Exclusive", 0xF0): ein System Common-Befehl, beginnt mit einem 0xF0-Status, endet mit 0xF7, dazwischen eine Folge variabler Länge von Daten-Bytes. Mit diesem Befehl können eine Vielzahl von geräteabhängigen Synthesizer-Parametern und -Funktionen gesteuert werden.
- *Reset* (0xFF): ein System Realtime-Befehl, der angeschlossene MIDI-Geräte in den Ausgangszustand zurückversetzt.

Daneben gibt es noch eine ganze Reihe weiterer, so genannter „Meta"-Nachrichten, die man ausschließlich in MIDI-Dateien findet, wo sie dazu dienen, bestimmte zusätzliche Informationen wie z.B. Tonart, Taktart und Tempo eines Stückes zu speichern. Meta-Nachrichten beginnen stets mit einem Status-Byte von 0xFF (das eigentlich vom MIDI-Standard für „Reset" reserviert ist; es ist also nicht möglich, in MIDI-Dateien einen Reset-Befehl zu speichern). Eine typische Meta-Nachricht ist z.B. die „Tempo"-Nachricht, die mit 0xFF 0x51 0x03 beginnt, gefolgt von drei Daten-Bytes, die zusammen das Tempo des Stückes in der Einheit „Mikrosekunden je Viertel-Note" bezeichnen. Z.B. beschreibt die Folge 0xFF 0x51 0x03 0x07 0xA1 0x20 ein Tempo von 500.000 µs/Viertel, also 120 BPM.

Weitere Informationen und eine genaue Beschreibung aller MIDI-Befehle finden sich auf Jeff Glatts Website „MIDI Technical Docs and Programming"[http://www.borg.com/~jglatt/].

Hinweis: Die technischen Details des MIDI-Nachrichten-Formats haben wir nur angeführt, um Ihnen einen Blick „hinter die Kulissen" zu ermöglichen. Sie werden für die Benutzung der in Kapitel 2 eingeführten Q-Midi-Schnittstelle nicht benötigt, da Q-Midi die MIDI-Nachrichten mehr oder weniger „im Klartext"dargestellt, z.B.: `note_on 0 60 64`, siehe Kapitel 5.

2 Grundlagen Programmierung

2.1 Warum MIDI-Programmierung?

Heutige PCs sind standardmäßig mit Sound-Karten ausgestattet, die über eine MIDI-Schnittstelle und größtenteils auch einen eingebauten MIDI-Synthesizer verfügen. Dies eröffnet die Möglichkeit, mit eigenen Programmen auf dem PC MIDI-Daten zu bearbeiten, auch in Echtzeit. Mittels Programmierung können z.B. komplexe Bearbeitungsfunktionen automatisiert oder Stücke direkt in MIDI „komponiert“ werden.

Exkurs: Was ist ein Programm?

Programme (auch „Software“ genannt) sind die Folgen von Instruktionen, die ein Computer ausführt, um ein bestimmtes Ergebnis zu erreichen, beispielsweise das Ausdrucken eines Dokuments oder die Aufzeichnung einer MIDI-Sequenz. Programme werden in speziellen Sprachen formuliert, die der Computer „versteht“, so genannten Programmiersprachen.

Solange ein Computer läuft, führt er stets irgendwelche Programme aus. Wenn ein Computer gestartet wird, übernimmt ein System-Programm die Kontrolle, dass die System-Ressourcen (z.B. Speicher und externe Geräte) verwaltet und es dem Benutzer erlaubt, weitere so genannte Anwendungs-Programme zu starten.

Durch Erstellung eigener Programme lassen sich MIDI-Anwendungen realisieren, die über den Funktionsumfang gängiger Sequenzer-Programme hinausgehen, z.B.:

- *Algorithmische Komposition:* Musik lässt sich beschreiben als ein Prozess, der sich in der Zeit vollzieht, und hat in diesem Sinn Ähnlichkeiten mit einem Computerprogramm. Es liegt daher nahe, Programme zu verwenden, um Musik zu erzeugen. Dies kann nach einem vorgegeben Schema, entweder deterministisch oder zufallsgesteuert, geschehen. Durch Echtzeit-Verarbeitung eingehender MIDI-Signale kann der Kompositionsprozess auch dynamisch, während der Ausführung eines Stückes, gesteuert werden. Diese technischen Möglichkeiten finden in der zeitgenössischen Musik häufig Verwendung.
- *Musikalische Analyse:* MIDI-Dateien oder in Echtzeit eintreffende MIDI-Daten können programmgesteuert analysiert werden. Ein bekanntes Beispiel hierfür ist die Markov-Analyse, mit der z.B. die Häufigkeit bestimmter Notenfolgen untersucht werden kann. Die Analyseergebnisse können dann verwendet werden, um neue Stücke zu synthetisieren, was eine Verbindung mit algorithmischen Kompositionstechniken ermöglicht.
- *Bearbeitung von MIDI-Dateien:* Mit Programmen können beliebig komplexe und umfangreiche Bearbeitungsschritte automatisiert werden, z.B. das Herausfiltern und Modifizieren einzelner Typen von MIDI-Nachrichten, Quantisierung von Noten-Werten, Änderung der Anschlagsdynamik, Variation von Kontrollparametern, Verwendung von reinen oder mikrotonalen Stimmungen, etc.

Programmierung ist ein kreativer Prozess, der viele Aspekte umfasst. Wir können hier nicht auf alle Aspekte eingehen, wollen aber an Hand einfacher Beispiele die wesentlichen Grundlagen der MIDI-Programmierung kennen lernen, die zur Erstellung eigener MIDI-Applikationen notwendig sind. Dabei wird insbesondere auf die Bearbeitung vorhandener MIDI-Sequenzen und die Verarbeitung von MIDI-Ereignissen in Echtzeit eingegangen.

2.2 MIDI-Programmierung mit Q

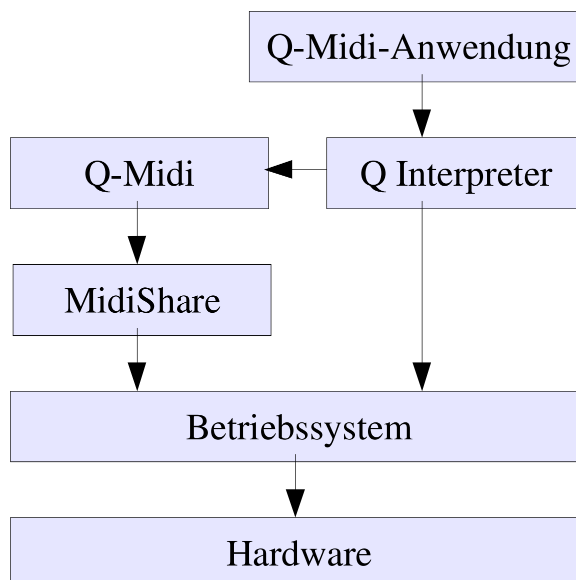
Zur Programmierung von MIDI-Anwendungen benötigen wir eine *Programmiersprache*, in der die auszuführenden Verarbeitungsfunktionen formuliert werden können, und ferner eine *Programmierschnittstelle* (API = „Application Programmer Interface“), mit der wir auf die MIDI-Schnittstelle des Computers zugreifen können. Wir verwenden hier die Programmiersprache *Q* (eine *interpretierte*, *funktionale* Programmiersprache), die die Programmierung der MIDI-Schnittstelle ermöglicht, ohne dass man sich mit den vielen kleinen technischen Details auseinandersetzen muss, die die MIDI-Programmierung z.B. in der Programmiersprache C recht mühselig machen. Q verfügt über eine MIDI-Schnittstelle, *Q-Midi* genannt, mit der auf einfache Weise auf die wesentlichen MIDI-Funktionen der zugrundeliegenden APIs der Betriebssysteme Linux und Windows zugegriffen werden kann.

Exkurs: Programmiersprachen

In der Computer-Technik unterscheidet man zwischen *imperativen*, *funktionalen* und *logischen* Programmiersprachen. Programme imperativer Programmiersprachen wie C und Pascal sind Befehlsfolgen, die genau festlegen, welche Instruktionen in welcher Reihenfolge ausgeführt werden müssen. Demgegenüber erlauben funktionale und logische Programmiersprachen wie z.B. Lisp und Prolog, die gewünschte Lösung mehr in abstrakter Form, entweder als mathematische Funktion oder als logischen Ausdruck, anzugeben, was die Programmierung oft erheblich vereinfacht.

Ferner unterscheidet man zwischen *interpretierten* und *compilierten* Sprachen. Programme einer interpretierten Sprache wie Lisp oder Basic werden von einem speziellen Programm, dem *Interpreter*, ausgeführt, während Programme einer compilierten Sprache wie C oder Pascal zunächst von einem so genannten *Compiler* in Maschinensprache übersetzt werden müssen. Programme interpretierter Sprachen lassen sich komfortabel im Dialog mit dem Interpreter ausführen, dafür bieten compilierte Sprachen normalerweise eine höhere Ausführungsgeschwindigkeit der Programme.

Um die Portierung auf verschiedene Betriebssysteme zu erleichtern, verwendet Q-Midi nicht direkt die MIDI-API des Betriebssystems, sondern die von Grame in Lyon entwickelte *MidiShare*-Bibliothek, die für eine ganze Reihe unterschiedlicher Betriebssysteme verfügbar ist [http://www.grame.fr/MidiShare/]. Eine Q-Midi-Anwendung ist ein Q-Programm, das durch den Q-Interpreter ausgeführt wird, und über Q-Midi und MidiShare auf die MIDI-Schnittstelle zugreift. Den Aufbau dieses Systems zeigt folgende Abbildung:



2.3 Ein einfaches Q-Midi-Beispiel

Um einen ersten Eindruck vom Aufbau und der Benutzung eines Q-Midi-Programmes zu vermitteln, zeigen wir ein einfaches Programm, das eingehende MIDI-Ereignisse mit einer gewissen Verzögerung wieder ausgibt, also eine Art „MIDI-Echo“realisiert.

```
/* bsp01.q: einfaches Q-Midi-Beispielprogramm */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, APP = midi_open "bsp01",
  _ = midi_connect IN APP || midi_connect APP OUT;

delay DT             = midi_flush APP || loop DT (midi_get APP);

loop _ (_,_,_,stop)  = ();
loop DT (_,_,T,MSG)  = midi_send APP PORT (T+DT,MSG) ||
                       loop DT (midi_get APP);
```

Um das Programm auszuprobieren, öffnen Sie die Datei `bsp01.q` mit `xemacs` und starten Sie den Q-Interpreter mit der Tastenfolge [Strg][C] [Strg][C]. (Unter Windows können Sie das Programm auch mit der Qpad-Anwendung öffnen und dann mit [F9] starten.) Der Eingabeprompt des Interpreters erscheint:

```
==>
```

Geben Sie nun den Funktionsnamen `delay` ein, gefolgt von der gewünschten Verzögerungsrate in Millisekunden. Schließen Sie Ihre Eingabe mit der [Return]-Taste ab. Z.B. (Eingaben sind kursiv gekennzeichnet):

```
==> delay 1000
```

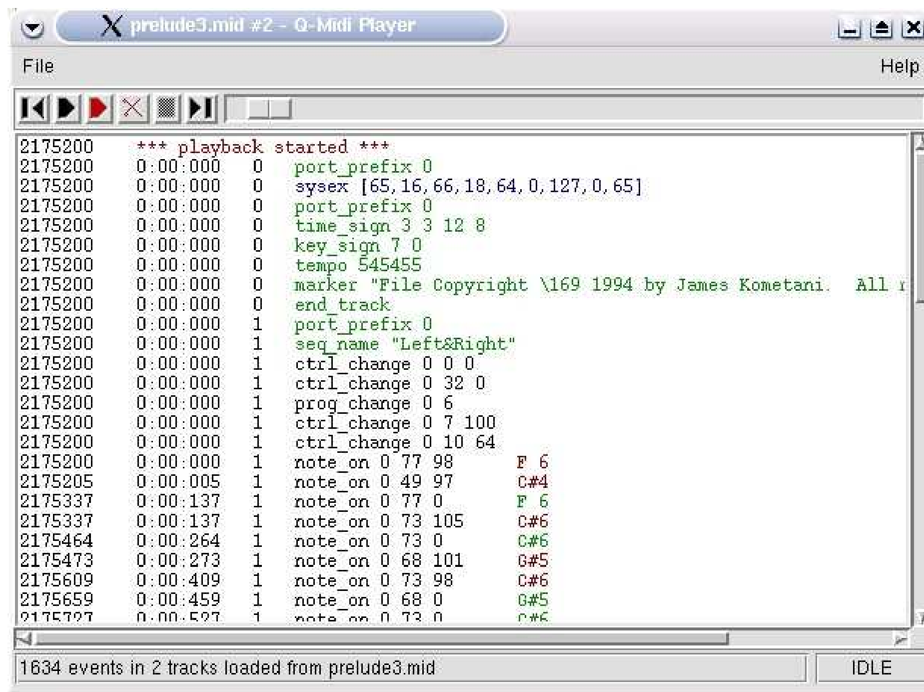
Auf dem angeschlossenen MIDI-Keyboard gespielte Noten sollten nun mit einer Verzögerung von einer Sekunde erneut wiedergegeben werden. (Damit dies auch bei kürzeren Verzögerungszeiten funktioniert, muss Ihr Synthesizer die selbe Note mehrfach auf dem gleichen MIDI-Kanal wiedergeben können.) Zum Beenden des Programms betätigen Sie auf dem MIDI-Keyboard die Stop-Taste (soweit vorhanden) oder beenden Sie den Interpreter mit der Tastenfolge [Strg][\] (einzugeben im Q-Eval-Puffer von XEmacs).

2.4 Q-Midi Player

Die Q-Midi-Installation umfasst auch ein mit Q-Midi realisiertes graphisches Programm zur Wiedergabe und Aufnahme von MIDI-Sequenzen, den *Q-Midi Player*. Sie können dieses Programm entweder in Ihre eigenen Programme einbinden order auch direkt von der Kommandozeile aus starten, z.B.:

```
$ player prelude3.mid
```

Die folgende Abbildung zeigt den Q-Midi Player in Aktion:



Weitere Informationen zu diesem Programm finden Sie in der Datei `etc/README-Player` im Q-Verzeichnis (`/usr/share/q` für Linux bzw. `/Programme/Qpad` für Windows).

3 Einführung in Q

Vor den Erfolg haben die Götter bekanntlich den Schweiß gesetzt, und so müssen wir nun zunächst die Programmiersprache Q kennen lernen, bevor wir mit der eigentlichen MIDI-Programmierung beginnen können. Glücklicherweise ist Q eine verhältnismäßig einfache Sprache, mit der man sich auch ohne tiefere Vorkenntnisse durchaus in einigen Tagen vertraut machen kann.

Hinweis: Wir können im folgenden nur auf die wichtigsten Elemente der Programmiersprache Q eingehen. Weitere Details entnehmen Sie bitte bei Bedarf dem „Q-Handbuch" *The Q Programming Language*, das auch „online" verfügbar ist. Das Handbuch, den Interpreter und das Q-Midi-Modul finden Sie zum Download unter der URL http://www.musikwissenschaft.uni-mainz.de/~ag/q.

3.1 Erstellung eines Skripts

Q-Programme werden auch kurz „Skripts" genannt. Um mit einem Skript zu arbeiten, müssen Sie es in einer Datei (normalerweise mit der Endung `.q`) speichern, und dann den Q-Interpreter aufrufen. Dies lässt sich am bequemsten erledigen, indem Sie das Skript mit dem XEmacs-Editor erstellen; sie können dann den Interpreter einfach mit der Tasten-Kombination [Strg][C] [Strg][C] starten.

Zur Übung erstellen wir ein kleines Skript, mit dem wir die Zeichenfolge „Hello, world!" ausgeben können. (Dies ist traditionell das erste Programm, dass man in jeder Programmiersprache kennenlernt.) Starten Sie XEmacs wie folgt:

```
$ xemacs hello.q
```

Geben Sie nun die folgenden Zeilen ein:

```
/* hello.q: Mein erstes Q-Skript. */

hello      = writes "Hello, world!\n";
```

Speichern Sie das Skript mit [Strg][X] [Strg][S] und starten Sie den Q-Interpreter mit [Strg][C] [Strg][C]. Das Editor-Fenster wird geteilt und in der unteren Hälfte läuft nun der Interpreter. Am Eingabeprompt ==> des Interpreters geben Sie `hello` ein, gefolgt von einem Zeilenvorschub:

```
==> hello
Hello, world!
()

==>
```

Die Zeichenkette „Hello, world!" wird wie verlangt mit einem Zeilenvorschub am Ende ausgegeben. (Der Wert `()` in der darauffolgenden Zeile stellt das „Ergebnis" der aufgerufenen `writes`-Funktion dar. Dazu später mehr.) Anschließend steht der Kursor wieder hinter dem Eingabeprompt des Interpreters, der nun Ihre nächste Eingabe erwartet. Um den Interpreter zu beenden, geben Sie nun entweder `quit` ein oder die Tastenkombination [Strg][D]. Sie können XEmacs auch ganz beenden durch Eingabe von [Strg][X] [Strg][C].

Um das Skript nach Beendigung von XEmacs erneut aufzurufen, starten Sie XEmacs mit dem Namen des Skripts wie oben, oder öffnen Sie innerhalb von XEmacs eine Skript-Datei mit [Strg][X] [Strg][F]. Sie können auch den Interpreter ohne ein Skript starten mit der Tastenkombination [Strg][C] [Strg][Q]. Die Online-Version des Q-Handbuchs erhalten Sie in XEmacs mit der Tastenkombination [Strg][H] [Strg][Q].

Hinweis: Unter Windows können Sie auch die „Qpad"-Anwendung verwenden, um Skripts zu erstellen und auszuführen. Starten Sie dazu das Qpad-Programm mit dem entsprechenden Symbol auf der Arbeitsfläche, geben Sie das Skript in das obere Editor-Fenster ein und speichern Sie es mit [Strg][S]. Mit der Taste [F9] können Sie das Skript nun starten. Für eine genauere Beschreibung von Qpad siehe die Hilfefunktion des Programms. In der Qpad-Hilfe finden Sie auch das komplette Q-Handbuch.

Exkurs: Arbeiten mit dem Q-Interpreter

Statt wie oben gezeigt mit XEmacs oder Qpad können Sie den Interpreter auch direkt von der Kommandozeile aus aufrufen, durch Eingabe des Kommandos `q` gefolgt von dem Dateinamen des Skripts, z.B.:

```
$ q hello.q
```

Dazu müssen Sie das Skript zunächst mit einem Editor erstellen. Sie können den Interpreter auch ohne den Namen eines Skripts aufrufen, in diesem Fall stehen nur die vordefinierten Funktionen zur Verfügung.

Wenn der Eingabe-Prompt des Interpreters erscheint, können Sie einen beliebigen Ausdruck eingeben. Der Interpreter wertet den eingegebenen Ausdruck aus und zeigt den errechneten Wert an, z.B:

```
==> sqrt (16.3805*5)/.05
181.0
```

Sie können die so genannte „anonyme Variable" _ verwenden, um auf das Ergebnis der letzten Berechnung zuzugreifen:

```
==> 16.3805*5
81.9025

==> sqrt _/.05
181.0
```

Zwischenergebnisse können auch explizit in Variablen gespeichert werden:

```
==> def X = 16.3805*5

==> sqrt X/.05
181.0
```

Die Definition einer Variablen können Sie mit `undef` wieder löschen:

```
==> undef X

==> X
X
```

Mehrere Kommandos und auszuwertende Ausdrücke können auf der selben Zeile eingegeben werden. Dazu trennen Sie die verschiedenen Kommandos mit einem Semikolon voneinander ab:

```
==> def X = 16.3805*5; sqrt X/.05; undef X
181.0
```

Der Interpreter verfügt über eine so genannte „Kommando-Geschichte", in der eingegebene Kommandos gespeichert werden. Sie können vorangegangene Kommandos mit den vertikalen Pfeiltasten [↑] und [↓] wieder aufrufen. (Dies funktioniert auch in XEmacs und Qpad, wenn Sie die Pfeiltasten mit der [Strg]-

Taste kombinieren.) Innerhalb der Kommandozeile kann der Kursor mit den horizontalen Pfeiltasten [←] und [→] bewegt werden, und Sie können die üblichen Editier-Funktionen (z.B. den „Backspace“ oder die [Entf]-Taste) verwenden, um Ihre Eingabe zu bearbeiten, bevor Sie sie mit der Return-Taste „abschicken“. Nützlich ist auch die „Komplettierungs“-Funktion, mit der Sie einen Funktions- oder Variablen-Namen vervollständigen können; geben Sie dazu den Beginn des Namens ein und betätigen Sie die Tabulator-Taste.

Der Interpreter kennt eine ganze Reihe weitere spezielle Kommandos, auf die wir hier nicht alle eingehen können. Z.B. können Sie alle definierten Variablen auf einen Schlag mit dem Kommando `clear` löschen. Sie können auch ein Skript innerhalb des Interpreters mit dem Kommando `edit` bearbeiten, oder ein neues Skript mit dem `run`-Kommando aufrufen. Mit den Kommandos `cd` und `pwd` können Sie das aktuelle Verzeichnis wechseln und sich das eingestellte Verzeichnis anzeigen lassen. Eine vollständige Übersicht aller Interpreter-Kommandos finden Sie im Abschnitt „Using Q“ des Q-Handbuchs. Wenn Sie das GNU Info-Programm installiert haben, können Sie die Online-Version des Handbuchs auch direkt im Interpreter mit dem Kommando `help` aufrufen. Um z.B. eine Übersicht der Interpreter-Kommandos zu erhalten, geben Sie das folgende Kommando ein:

```
==> help commands
```

Verwenden Sie die [Bild↑]- und [Bild↓]-Tasten, um in der Anzeige zu blättern. Wenn Sie mit Lesen fertig sind, geben Sie `q` ein, um das Info-Programm zu beenden.

Um den Interpreter zu beenden, verwenden Sie die Funktion `quit` oder geben Sie am Beginn der Kommandozeile die Tastenkombination [Strg][D] ein. Wenn Sie den Interpreter von XEmacs oder Qpad aus gestartet haben, wird bei Beenden von XEmacs bzw. Qpad natürlich auch automatisch der Q-Interpreter beendet.

3.2 Anatomie eines Skripts

Wir wollen uns nun etwas genauer mit dem Aufbau von Q-Skripts befassen. Ein Q-Skript besteht im wesentlichen aus den folgenden Elementen, die in beliebiger Reihenfolge aufgeführt werden können:

- *Kommentare:* Kommentare beginnen mit `/*` und enden mit `*/`. Der dazwischenliegende Text darf beliebig lang sein und mehrere Zeilen umfassen. Außerdem können zeilenorientierte Kommentare wie in C++ oder in Prolog angegeben werden, d.h. durch `//` bzw. `%` gefolgt von beliebigem Text bis zum Zeilenende.
- *Deklarationen:* Deklarationen dienen dazu, neue Funktions-, Variablen- und Typ-Symbole zu vereinbaren, oder auf die Symbole und Definitionen anderer Skripts zuzugreifen. Siehe Abschnitt 3.3.
- *Gleichungen:* Diese bilden den Kern eines Q-Skripts. Mit ihnen werden die Funktionen eines Skripts definiert, wie z.B. die Funktion `hello` im vorangegangenen Abschnitt. Eine Gleichung besteht immer aus einer linken und einer rechten Seite, die voneinander durch das Symbol = getrennt sind. Beide Seiten einer Gleichung können im Prinzip beliebige „Ausdrücke“ (s. Abschnitt 3.4) sein. Der Interpreter wendet Gleichungen immer von links nach rechts an, indem er innerhalb eines auszuwertenden Ausdrucks eine passende linke Seite durch die entsprechende rechte Seite ersetzt. Man nennt dies auch „Termersetzung“. Mehr dazu in Abschnitt 3.6 und 3.7.
- *Variablen-Definitionen:* Man kann den Wert eines Ausdrucks auch in einer Variablen speichern. Der entsprechende Wert kann dann auf der rechten Seite einer Gleichung oder auf der Eingabezeile des Interpreters durch Angabe des Variablen-Namens verwendet werden. Siehe Abschnitt 3.8.

3.3 Deklarationen

In Q müssen Variablen- und Funktions-Symbole nicht deklariert werden; der Interpreter nimmt beim Fehlen einer Deklaration für ein neues Symbol automatisch an, dass es sich um ein „privates“ Symbol handelt, das nur innerhalb des aktuellen Skripts verwendet wird. Soll ein Symbol zur Verwendung in anderen Skripts bereitgestellt („exportiert“) werden, so muss es explizit als „public“ vereinbart werden, z.B.:

```
public hello, foo X Y Z;
public var BAR;
```

Die erste Deklaration führt zwei Funktionssymbole ein, `hello` und `foo`. Dabei wird `foo` als ein Funktionssymbol mit drei Parametern vereinbart, hier mit `X`, `Y` und `Z` bezeichnet. Die Kennzeichnung der Parameter ist optional, und ist als eine Zusicherung zu verstehen, dass die entsprechende Funktion drei Argumente erwartet – wird ein Symbol mehrfach deklariert, so müssen die Anzahl der Parameter (nicht jedoch deren Namen) übereinstimmen.

Die zweite Deklaration führt ein Variablen-Symbol `BAR` ein. Diesem kann später mittels `def` ein Wert zugewiesen werden, vgl. Abschnitt 3.8.

Generell muss die erste Deklaration eines Symbols jeweils vor seiner ersten Verwendung stehen. Mehrere Deklarationen des gleichen Symbols sind zulässig, die Deklarationen müssen aber miteinander konsistent sein.

Man kann ein Symbol auch explizit als „private“ vereinbaren:

```
private hallo, bar X Y;
private var PRIVATE;
```

Dies ist dann notwendig, wenn ein privates Symbol eingeführt werden soll, das anderswo (d.h. in einem importierten Skript, s.u.) bereits als „public“ vereinbart wurde.

Um auf die „public“-Symbole und Definitionen eines anderen Skripts zuzugreifen, benötigt man eine „import“-Deklaration. Solche Deklarationen stehen üblicherweise am Beginn eines Skripts:

```
import midi, mididev;
```

Hier werden zwei Skripts `midi.q` und `mididev.q` importiert (der Interpreter ergänzt die Endung `.q` des Dateinamens automatisch).

Statt des Schlüsselwortes `import` kann man auch `include` verwenden. Der Unterschied zur „import“-Deklaration besteht darin, dass die „public“-Symbole eines mit `include` importierten Skripts „reexportiert“, also zusammen mit den anderen „public“-Symbolen des Skripts exportiert werden, als ob das importierte Skript Bestandteil des importierenden Skripts wäre. Dies ist insbesondere dann nützlich, wenn verschiedene bereits vorhandene Skripts zu einem neuen Skript zusammengefasst werden sollen.

Ein weiterer Deklarations-Typ ermöglicht die Vereinbarung von Datentypen. Z.B. lässt sich ein „Aufzählungstyp“ mit den Tagen der Woche wie folgt vereinbaren:

```
public type Day = const sun, mon, tue, wed, thu, fri, sat;
```

Auch Datentyp-Deklarationen können als „public" oder „private" gekennzeichnet werden. Das Schlüsselwort `const` in obiger Deklaration legt fest, dass es sich bei den Funktions-Symbolen `sun`, `mon`, etc. um „Konstanten" handelt, die nicht als linke Seite einer Gleichung auftreten, also nicht neu definiert werden dürfen. Der `MidiMsg`-Datentyp der Q-Midi-Schnittstelle ist auf ähnliche Weise vereinbart. Weitere Informationen zur Deklaration und Verwendung von Datentypen finden sich in Abschnitt 3.9.

3.4 Ausdrücke

Wie in den meisten funktionalen Programmiersprachen dreht sich in Q alles um die Auswertung von *Ausdrücken*. Wir unterscheiden genau wie in anderen Programmiersprachen *einfache* (oder *elementare*) und *zusammengesetzte Ausdrücke*. Ein Unterschied zu den meisten anderen Programmiersprachen besteht allerdings darin, dass der Wert eines Ausdrucks selbst wieder ein zusammengesetzter Ausdruck sein kann; darauf werden wir in Abschnitt 3.7 näher eingehen.

Einfache Ausdrücke

- *Ganze Zahlen* werden als eine dezimale Ziffernfolge ohne Dezimalpunkt notiert. Negative Zahlen werden durch ein vorangestelltes Minuszeichen gekennzeichnet. Im Unterschied zu vielen anderen Programmiersprachen können ganze Zahlen in Q beliebig große und kleine Werte annehmen. Beispiele: `0`, `-5`, `17`, `-99`, `192837190379190237`. Ganze Zahlen können auch in Hexadezimal-Notation angegeben werden, indem man ihnen das Präfix `0x` voranstellt; z.B. entspricht `0x7F15B` der Dezimalzahl `520539`. Die Hexadezimalziffern `A` bis `F` dürfen auch in Kleinschrift angegeben werden, z.B. `0x7f15b`. Auch eine Angabe als *Oktalzahl* (d.h. zur Basis 8, mit Ziffern 0-7) ist möglich; dazu wird der Zahl die Ziffer `0` vorangestellt, z.B. `033` = `27`.
- *Fließkommazahlen* sind Ziffernfolgen, die einen Dezimalpunkt und/oder einen Exponenten (zur Basis 10) enthalten, z.B.: `0.`, `.1`, `-0.7`, `1.23456E78`. Letztere stellt die Zahl 1.23456×10^{78} dar. Fließkommazahlen werden in Q stets mit 64 Bit dargestellt (entsprechend einem absoluten Darstellungsbereich von etwa `1.7E-308` bis `1.7E308` mit ca. 15 Dezimalstellen Genauigkeit).
- *Zeichenketten* bestehen aus beliebigen druckbaren ASCII-Zeichen (außer `"` und `\`), die in doppelte Anführungszeichen eingefasst werden, z.B.: `""` (leere Zeichenkette), `"a"` (einzelnes Zeichen), `"abc"`, `"!$%&"` oder `"Hello, world!"`. Um die Zeichen `"`, `\` und spezielle nicht druckbare Zeichen wie z.B. den Zeilenvorschub anzugeben, wird das `\`-Zeichen („Backslash") in Kombination mit anderen Zeichen verwendet (so genannte „Escape-Sequenzen"). Die gebräuchlichsten Escape-Sequenzen sind in folgender Tabelle zusammengefasst:

`\n`	Zeilenvorschub (Newline)
`\r`	Wagenrücklauf (Return)
`\t`	Tabulator-Zeichen
`\\`	Backslash
`\"`	doppelte Anführungszeichen
`\`*n*	ASCII-Zeichen Nr. *n* (dezimal/hexadezimal/oktal)

Z.B. hat das so genannte „Escape"-Zeichen den ASCII-Code 27, lässt sich also als `"\27"`, `"\033"` oder `"\0x1b"` darstellen, je nachdem, ob man die dezimale, oktale oder hexadezimale Schreibweise vorzieht.

Exkurs: Der ASCII-Code

Bei dem so genannten *ASCII-Code* („American Standard Code for Information Interchange") handelt es sich um einen weit verbreiteten Zeichen-Code, in dem jedes Zeichen durch eine 7-Bit-Zahl dargestellt wird. Heutige Computer verwenden meist einen erweiterten ASCII-Code, in dem jedes Zeichen 8 Bit (also ein Byte) umfasst. In solchen Codes lassen sich 256 verschiedene Zeichen, mit den Codes 0 bis 255, darstellen. Der ASCII-Code ist so aufgebaut, dass die druckbaren Zeichen bei Code 32 (dem Leerzeichen) beginnen, und die Ziffern in numerischer, die Groß- und Kleinbuchstaben jeweils in alphabetischer Reihenfolge angeordnet sind. Die Ziffern beginnen bei ASCII-Code 48, die Großbuchstaben bei Code 65 und die Kleinbuchstaben bei Code 97. Die Buchstaben und Ziffern werden auch zusammen *alphanumerische Zeichen* genannt. Der ASCII-Code 0 wird in vielen Programmiersprachen (auch in Q) benutzt, um das Ende einer Zeichenkette anzuzeigen. Die Codes 1 bis 31 (die so genannten *Kontrollzeichen*) sind allesamt nicht druckbar, sondern dienen zur Steuerung eines Ausgabegeräts. Die wichtigsten Kontrollzeichen sind ASCII-Code 10 (Zeilenvorschub), der allgemein verwendet wird, um das Zeilenende anzuzeigen, und Code 7, das Tabulatorzeichen, das einen Sprung an die nächste Tabulatorposition darstellt.

- *Funktions-* und *Variablen-Symbole* werden durch *Bezeichner* angegeben, Folgen von alphanumerischen Zeichen, die mit einem Buchstaben beginnen (der Unterstrich _ zählt dabei als Buchstabe). Wie in Prolog werden Symbole, die mit einem Großbuchstaben beginnen, als Variablen-Symbole aufgefasst. Bezeichner, die mit einem Kleinbuchstaben (inklusive _) anfangen, sind Funktions-Symbole. (Eine Sonderrolle spielt das Symbol _, die so genannte „anonyme Variable", siehe Abschnitt 3.6.)

  Auf Symbole in anderen Skripts kann auch mit so genannten *qualifzierten Bezeichnern* der Form *Skriptname*`::`*Bezeichner* zugegriffen werden. Dies ist manchmal notwendig, um Mehrdeutigkeiten aufzulösen. Exportieren z.B. zwei Skripts `foo1.q` und `foo2.q` jeweils ein Funktions-Symbol mit Namen `foo`, so wird ersteres mit `foo1::foo` und letzteres mit `foo2::foo` bezeichnet.

  Wichtig: Bestimmte alphanumerische Zeichenfolgen sind in Q als *Schlüsselworte* reserviert und können *nicht* als Bezeichner verwendet werden. Es sind dies die folgenden:

```
as          and      const    def      div     else    extern
if          import   in       include  mod     not     or
otherwise   private  public   special  then    type    undef
var         where
```

Zusammengesetzte Ausdrücke

Beginnend mit den elementaren Ausdrücken können wir nun schrittweise immer kompliziertere Ausdrücke zusammensetzen. Dazu gibt es in Q die folgenden Konstruktionen:

- *Listen* werden wie in Prolog durch eine in eckigen Klammern eingeschlossene Aufzählung der Listen-Elemente dargestellt, z.B. ist `[1,2,3]` die Liste der drei Zahlen `1`, `2` und `3`. Listen dürfen verschiedenartige Elemente enthalten und geschachtelt werden, wie z.B. in `[[1,2,3], ["world",3.14]]`. Listen werden wie in Prolog als mit dem „Listen-Operator" `[|]` gebildete rechts-rekursive Strukturen repräsentiert und können auch so notiert werden; z.B. ist `[1,2]`

identisch mit `[1|[2|[]]]`. Generell stellt `[]` die leere Liste, und `[X|Xs]` eine Liste mit erstem Element `X` und Rest-Liste `Xs` dar. Diese Schreibweise ist nützlich, wenn rekursive Listen-Operationen definiert werden sollen, vgl. Abschnitt 3.6.

- *Tupel* sind analog zu Listen aufgebaut, werden aber intern als so genannte „Vektoren" repräsentiert, was eine platzsparende Speicherung und schnellen Zugriff auf einzelne Elemente ermöglicht. Im Unterschied zu Listen werden Tupel, wie in der Mathematik üblich, in runde Klammern eingefasst, z.B. `()` (leeres Tupel), `(99)` (1-Tupel mit `99` als einzigem Element), `(1,2,(a,b))` (Tripel, bestehend aus den Zahlen `1`, `2` und dem Paar `(a,b)`). Wie bei Listen kann die Schreibweise `(X|Xs)` verwendet werden, um ein Tupel mit Anfangselement `X` und Rest-Tupel `Xs` darzustellen.
- *Funktions-Anwendungen* werden in Q durch einfaches Nebeneinander-Schreiben notiert, wie z.B. in `sin 0.5`. Hier wird ein Funktions-Symbol `sin` (die „eingebaute" Sinus-Funktion, vgl. Abschnitt 3.5) auf die Fließkomma-Zahl `0.5` angewendet, was bei der Auswertung im Interpreter den Wert des Sinus an der Stelle 0.5 ergibt. Im allgemeinen Fall können sowohl die angewendete Funktion als auch das Funktions-Argument selbst wieder beliebig komplizierte Ausdrücke sein, wobei geschachtelte Funktions- und Operator-Anwendungen im Argument geklammert werden müssen, wie z.B. in `sin (3.1415/2)`.

  Funktions-Anwendungen mit mehreren Argumenten werden ebenfalls durch Nebeneinander-Schreiben notiert, z.B. `max 7 12`. Dabei wird implizit Links-Klammerung angenommen, d.h., `max X Y` ist dasselbe wie `(max X) Y`. Der Gedanke dabei ist, dass `max X` selbst wieder eine Funktion darstellt, die bei Anwendung auf das Argument `Y` den Wert `(max X) Y = max X Y` liefert; diese Art, Funktionen mehrerer Veränderlicher zu notieren, wird nach dem amerikanischen Logiker Haskell B. Curry als *Currying* bezeichnet und ist in modernen funktionalen Programmiersprachen sehr verbreitet.
- *Operator-Anwendungen* sind spezielle Funktions-Anwendungen, bei denen ein vordefinierter Operator wie z.B. + die Rolle der Funktion übernimmt. Der Unterschied besteht darin, dass zur Vereinfachung der Notation bei Operatoren die gebräuchliche Infix-Schreibweise verwendet wird. Der Interpreter kennt die üblichen Präzedenz-Regeln, bei davon abweichender Auswertungs-Reihenfolge müssen Klammern gesetzt werden. Beispiele: `(X+1)*(Y-1)`, `X+3*(Y-1)`, `X+1/sin(Y-1)`. Man beachte, dass Funktions-Anwendungen stets Vorrang vor Operatoren haben (vgl. letztes Beispiel!).

  Wie auch in anderen funktionalen Programmiersprachen (z.B. Haskell) sind Operator-Anwendungen nur eine bequeme Kurzform für entsprechende Funktions-Anwendungen. Man kann jeden Operator in eine gewöhnliche Präfix-Funktion verwandeln, indem man ihn einklammert. Z.B. ist `X+Y` genau dasselbe wie `(+) X Y`. Außerdem kann man bei Infix-Operatoren so genannte *Operator-Sektionen* bilden, bei denen entweder das linke oder rechte Argument fehlt. So ist z.B. `(1/)` die reziproke Funktion: `(1/) X = 1/X`, `(*2)` die Verdopplungs-Funktion: `(*2) X = X*2`.

Exkurs: Operatoren

Operatoren werden verwendet, um gängige arithmetische und logische Operationen wie Addition, Multiplikation, Division und Vergleiche sowie logische Verknüpfungen („und", „oder", „nicht") auszudrücken. Man unterscheidet *unäre* (einstellige) und *binäre* (zweistellige) Operatoren, je nach Anzahl der Operator-Argumente (die man auch *Operanden* nennt). Erstere werden meist als *Präfix* notiert (z.B. `-X` für unäres Minus, `not X` für logische Negation), letztere in *Infix*-Schreibweise, d.h. zwischen den Operanden (z.B. `X*Y` für Multiplikation, `X<=Y` für „kleiner oder gleich", `X and Y` für logisches „und").

Wie in den meisten Programmiersprachen werden auch in Q Operatoren nicht einfach von links nach rechts ausgewertet, sondern es werden die üblichen Präzedenzregeln beachtet (z.B. „Punkt vor Strich“). Bei Operatoren gleicher Präzedenz wird normalerweise von links nach rechts ausgewertet, z.B. `X-Y-Z = (X-Y)-Z`. Man nennt solche Operatoren auch *links-assoziativ*. Eine Ausnahme ist der Exponentiations-Operator `^` („`X` hoch `Y`“), der entsprechend üblichem mathematischen Gebrauch *rechts-assoziativ* ist, d.h. `X^Y^Z = X^(Y^Z)`. Gleiches gilt für den Index-Operator: `X!I!J = X!(I!J)`; Mehrfach-Indizes müssen also geklammert werden: `(X!I)!J`. Ein anderer Sonderfall sind die Vergleichs-Operatoren (`<`, `>`, `<=`, `>=`, etc.), die *nicht-assoziativ* sind, d.h. Kombinationen wie `X<Y<Z` sind nicht erlaubt; um auszudrücken, dass sowohl `X<Y` als auch `Y<Z` gelten soll, muss man eine logische Verknüpfung verwenden: `(X<Y) and (Y<Z)`. (Die Klammern sind hier notwendig, da in Q ähnlich wie in der Programmiersprache Pascal die logischen Operatoren Vorrang vor den Vergleichs-Operatoren haben.)

Die folgende Tabelle listet alle für uns wichtigen Operatoren in absteigender Präzedenz-Reihenfolge auf. (Eine vollständige Auflistung findet man im Q-Handbuch. Für eine Beschreibung der Funktionsweise der wichtigsten Operatoren siehe Abschnitt 3.5.)

Gruppe	Operatoren	Bedeutung	Beispiel
Exponentiation/ Subskript	`^` `!`	Exponentiation Index	`X^Y` `X!I`

unäre Präfix-Operatoren	`-`
	`#`
	`not`
		unäres Minus
		Anzahl
		logisches „nicht“
			`-X`
			`#X`
			`not X`

Multiplikations-Operatoren	`*`
	`/`
	`div`
	`mod`
	`and`
	`and then`
		Multiplikation
		Division
		ganzzahlige Division
		Rest der ganzzahligen Division
		logisches „und“
		logisches „und dann“
			`X*Y`
			`X/Y`
			`X div Y`
			`X mod Y`
			`X and Y`
			`X and then Y`

Additions-Operatoren	`+`
	`-`
	`++`
	`or`
	`or else`
		Addition
		Subtraktion
		Konkatenation
		logisches „oder“
		logisches „oder sonst“
			`X+Y`
			`X-Y`
			`X++Y`
			`X or Y`
			`X or else Y`

Relationale Operatoren	`<`
	`>`
	`<=`
	`>=`
	`=`
	`<>`
	`in`
		„kleiner als“
		„größer als“
		„kleiner oder gleich“
		„größer oder gleich“
		„gleich“
		„ungleich“
		„in“
			`X<Y`
			`X>Y`
			`X<=Y`
			`X>=Y`
			`X=Y`
			`X<>Y`
			`X in Y`

Sequenz-Operator	`||`	Hintereinander-Ausführung	`X||Y`

Zum Abschluss fassen wir die verschiedenen Typen von Ausdrücken in einer kleinen Übersichtstabelle zusammen:

Einfache Ausdrücke		Zusammengesetzte Ausdrücke	
Ausdrucks-Typ	Beispiele	Ausdrucks-Typ	Beispiele

Ganze Zahl	`12345678`
	`0xa0`
	`-033`
		Liste	`[]`
			`[a]`
			`[a,b,c]`

Fließkommazahl	`0.`
	`-1.0`
	`1.2345E-78`
		Tupel	`()`
			`(a)`
			`(a,b,c)`

Zeichenkette	`""`
	`"abc"`
	`"Hello, world!\n"`
		Funktions-Anwendung	`sin 0.5`
			`max X (sin Y)`
			`(*2) (sin X)`

Symbol	`foo`
	`BAR`
	`foo1::foo`
		Operator-Anwendung	`-X`
			`X+Y`
			`X or Y`

3.5 Vordefinierte Operatoren und Funktionen

In der Programmiersprache Q sind eine große Anzahl von Operatoren und Funktionen bereits zur sofortigen Verwendung vordefiniert. Wir können hier nicht auf alle diese Operationen eingehen, wollen aber die wichtigsten Funktionsgruppen kurz vorstellen.

- *Arithmetische Operationen:* Die gebräuchlichsten arithmetischen Operatoren (Addition, Subtraktion, Multiplikation, Division, Exponentiation) können sowohl auf ganze als auch auf Fließkomma-Zahlen angewendet werden. Bei Addition, Subtraktion und Multiplikation entspricht der Typ des Ergebnisses stets den Operanden, z.B. ergibt `123*456` die ganze Zahl `56088`, während `123.0*456.0` die Fließkommazahl `56088.0` liefert. Bei gemischten Operanden wird eine Fließkommazahl zurückgeliefert: `123*456.0 = 56088.0`. Der Divisions-Operator liefert stets eine Fließkommazahl, z.B. `12/3 = 4.0`, genau wie der Exponentiations-Operator: `5^3 = 125.0`. Außerdem gibt es die Operatoren `div` und `mod`, die den Wert und den Rest der ganzzahligen Division liefern: `17 div 3 = 5` und `17 mod 3 = 2`.
- *Numerische Operationen:* Die üblichen trigonometrischen Funktionen wie `sin` und `cos`, Quadratwurzeln (`sqrt`), Logarithmen (`ln`, `log`) und die Exponentialfunktion (`exp`) sind alle vordefiniert. Außerdem gibt es die `random`-Funktion zur Erzeugung von gleichverteilten Pseudo-Zufallszahlen.
- *Zeichenketten-Operationen:* Zeichenketten können mit dem `++`-Operator „konkateniert", d.h. aneinandergehängt werden: `"abc"++"def" = "abcdef"`. Der Anzahl-Operator `#` liefert die Länge einer Zeichenkette: `#"abc" = 3`. Mit dem Index-Operator kann man auf die einzelnen Zeichen einer Zeichenkette zugreifen: `"abcde"!2 = "c"`. (Man beachte, dass Indizes wie in der Programmiersprache C stets bei 0 beginnen, `S!0` ist also das erste, `S!(#S-1)` das letzte Zeichen einer Zeichenkette `S`.) Daneben gibt es noch die Funktion `sub`, mit der man einen bestimmten Abschnitt einer Zeichenkette extrahieren kann (z.B. `sub "abcde" 1 3 = "bcd"`), und die Funktion `pos`, die das erste Vorkommen einer Zeichenkette in einer anderen Zeichenkette liefert (z.B. `pos "cd" "abcde" = 2`).
- *Listen- und Tupel-Operationen:* Konkatenation, Längen-Bestimmung, Element-Indizierung und die Extraktion von Teil-Sequenzen funktionieren auch bei Listen und Tupeln, z.B.: `[a,b,c]++[d,e,f] = [a,b,c,d,e,f]`, `#[a,b,c] = 3`, `(a,b,c,d,e)!2 = c`, `sub (a,b,c,d,e) 1 3 = (b,c,d)`. Daneben definiert die so genannte „Standard-Biblio-

thek“ eine große Zahl weiterer nützlicher Listen-Funktionen; auf diese kommen wir später zurück.

- *Vergleichs-Operationen:* Die Operatoren <, >, <=, >=, = und <> können verwendet werden, um zwei Zahlen, Zeichenketten, Listen oder Tupel miteinander zu vergleichen. Der Vergleich zweier Zeichenketten oder Listen erfolgt dabei „lexikographisch“ (d.h. elementweise von links nach rechts); Tupel können nur auf Gleichheit/Ungleichheit miteinander verglichen werden. Das Ergebnis des Vergleichs ist ein so genannter *Wahrheitswert*: entweder `true` (wahr) oder `false` (falsch). Die Werte `true` und `false` sind vordefinierte Konstantensymbole. Auch die Wahrheitswerte selbst können verglichen werden, wobei `false < true` gilt; dies ist nützlich, um z.B. logische Implikationen zu testen.
- *Logische Operationen:* Wahrheitswerte können mit den logischen Operatoren `not`, `and` und `or` in der üblichen Weise verknüpft werden. Das Ergebnis ist jeweils wieder ein Wahrheitswert, z.B. `true and true = true`, `false or false = false`, `not true = false`. (Außerdem können diese Operatoren auch als „bitweise“ Operationen auf ganzzahlige Werte angewendet werden; siehe das Q-Handbuch für Details.) Daneben gibt es noch die so genannten „Kurzschluss“-Operatoren `and then` und `or else`. Diese funktionieren im wesentlichen wie die oben genannten Verknüpfungen `and` und `or`, werten den zweiten Operanden aber nur aus, wenn der erste Operand noch keine Entscheidung über das Ergebnis erlaubt. Z.B. ist `false and then X = false`, gleichgültig was der Wert von `X` ist; `true and then X` ergibt dagegen den Wert von `X`. Ein mit diesen Operatoren gebildeter logischer Ausdruck wird also nur so weit ausgewertet, wie es zur Bestimmung des Ergebnisses notwendig ist. Er ist darum für komplexe Bedingungen, deren einzelne Teilbedingungen viel Rechenzeit erfordern können, vorzuziehen.
- *Umrechnungs-Operationen:* Eine Fließkommazahl kann mit `round` auf eine ganze Zahl gerundet werden. Z.B.: `round 1.5 = 2`. Mittels `trunc` wird dagegen der Teil hinter dem Dezimalpunkt einfach abgeschnitten: `trunc 1.5 = 1`. Umgekehrt wandelt die `float`-Funktion eine ganze Zahl in die entsprechende Fließkomma-Zahl um: `float 0 = 0.0`. Auch zur Umrechnung zwischen Zahlen und Zeichenketten gibt es einige nützliche Funktionen. So liefert `ord` den zu einem Zeichen gehörigen ASCII-Code: `ord "A" = 65`; umgekehrt wandelt die `chr`-Funktion eine Zahl zwischen 0 und 255 in das entsprechende ASCII-Zeichen um: `chr 65 = "A"`. Wichtig sind außerdem die Funktionen `str` und `val`, mit denen ein beliebiger Q-Ausdruck in eine Zeichenkette, und umgekehrt eine Zeichenkette wieder in den entsprechenden Ausdruck umgewandelt werden kann: `str (2*(X+1)) = "2*(X+1)"`, `val "2*(X+1)" = 2*(X+1)`. Schließlich gibt es noch die Funktionen `list` und `tuple`, mit denen man Tupel in Listen und umgekehrt konvertieren kann.
- *Ein-/Ausgabe-Operationen:* Zur Ein- und Ausgabe verfügt Q über eine ganze Reihe verschiedener Operationen. Für den Anfang sind vor allem zwei Funktionen wichtig: `writes`, mit der eine Zeichenkette auf dem Terminal angezeigt wird, und `reads`, mit der man eine Zeile vom Terminal einlesen kann. Diese beiden Operationen werden oft mit dem Sequenz-Operator `||` verknüpft, um einen Dialog mit dem Benutzer zu realisieren. Z.B. gibt der folgende Ausdruck,

```
writes "Bitte Dateinamen eingeben: " || reads
```

eine Meldung auf dem Terminal aus, wonach eine Eingabe des Benutzers eingelesen wird. Die `writes`-Funktion liefert als Wert stets die Konstante `()`, während `reads` die gelesene Zeichenkette zurückgibt. Mit dem `||`-Operator werden die beiden Operationen hintereinandergeschaltet; das Ergebnis ist der Rückgabewert des letzten Teilausdrucks, im Beispiel ist dies das Ergebnis der `reads`-Funktion.

Es ist auch möglich, beliebige Ausdrücke vom Terminal einzulesen und auszugeben; dazu werden die Funktionen `read` und `write` verwendet:

```
def X = writes "Bitte X eingeben: " || read
writes "Das Ergebnis ist: " || write (X/sin X) || writes "!\n"
```

Eine andere nützliche Funktion ist `printf`, die der C-`printf`-Routine nachempfunden ist und eine formatierte Ausgabe ermöglicht, z.B:

```
printf "Das Ergebnis ist: %g!\n" (X/sin X)
```

Der „Platzhalter" `%g` zeigt hier an, wo eine Fließkommazahl in den ausgegebenen Text eingebettet werden soll. Ähnliche Symbole gibt es auch zur Einfügung von ganzen Zahlen und Zeichenketten.

Außerdem gibt es auch Operationen, die die Eingabe von und die Ausgabe in Dateien ermöglichen, die auf der Festplatte gespeichert sind, wie z.B. die `fwrites`- und `freads`-Operationen; da wir diese im folgenden nicht benötigen, verweisen wir für eine Beschreibung dieser Funktionen auf das Q-Handbuch.

- *Spezielle Operationen:* Hier ist insbesondere die `halt`-Funktion zu nennen, mit der die Auswertung eines Ausdrucks abgebrochen werden kann, und die `quit`-Funktion, mit der der Interpreter verlassen wird. Die `time`-Funktion liefert die Systemzeit in Sekunden seit 00:00:00 UTC (Coordinated Universal Time), 1. Januar 1970, was zum Beispiel zur Zeitmessung nützlich ist. Mit der Standard-Bibliotheks-Funktion `ctime` kann dieser Wert in aktuelles Datum und Uhrzeit der eingestellten Zeitzone umgerechnet werden, z.B.: `ctime time = "Tue Sep 17 17:17:54 2002"`. Mit der `sleep`-Funktion schließlich kann die aktuelle Berechnung für eine gegebene Zeitspanne unterbrochen werden. Gönnen wir dem Interpreter (und uns!) doch einmal eine Pause von zwei Minuten: `sleep 120`.

Standard-Bibliotheks-Funktionen

Die meisten der oben genannten Operationen sind „eingebaut", also im Interpreter fest verdrahtet. Daneben gibt es aber noch eine große Zahl von Funktionen, die in der so genannten *Standard-Bibliothek* definiert sind. Dabei handelt es sich um eine Sammlung von Q-Skripts mit allgemein nützlichen Funktionen, die in jeder Q-Installation enthalten sind und vom Interpreter immer automatisch geladen werden. Auch auf diese Funktionen kann also stets zugegriffen werden. Die Standard-Bibliothek enhält u.a. eine Sammlung zusätzlicher Zeichenketten- und Listen-Funktionen, weitere numerische Funktionen, Operationen mit komplexen Zahlen, wichtige „Container"-Daten - typen (das sind indizierte Datentypen wie „Arrays" oder „Dictionaries", in denen beliebige Informationen abgelegt werden können), und zusätzliche System-Funktionen. Für unsere Zwecke sind insbesondere die zusätzlichen Listen-Funktionen wichtig, da wir mit ihrer Hilfe MIDI-Sequenzen erzeugen und manipulieren werden. Die folgenden Funktionen sind alle im Standard-Bibliotheks-Skript `stdlib.q` definiert:

- `map F Xs`: `map` wendet die Funktion `F` auf jedes Element der Liste `Xs` an.

  Beispiel: Addiere `1` zu jedem Element der Liste `[1,2,3]`:
  `map (+1) [1,2,3] = [2,3,4]`

- `do F Xs`: `do` wendet genau wie `map` die Funktion `F` auf jedes Element der Liste `Xs` an, gibt aber statt der Liste aller Ergebnisse der Funktionsanwendung einfach `()` zurück. Dies ist dann

nützlich, wenn eine Operation nur wegen ihrer Nebeneffekte auf die Listenelemente angewendet werden soll.

Beispiel: Gib eine Liste von ganzen Zahlen Zeile für Zeile auf dem Terminal aus:
`do (printf "%g\n") [1,2,3] = ()`

- `filter P Xs`: `filter` liefert die Liste aller Elemente der Liste `Xs`, für die das Prädikat `P` erfüllt ist, d.h. den logischen Wert `true` liefert. (Ein *Prädikat* ist eine Funktion, die stets einen Wahrheitswert liefert.)

  Beispiel: Alle positiven Elemente einer Liste:
  `filter (>0) [1,-1,2,0,3] = [1,2,3]`

- `hd Xs`: `hd` („Head") liefert das erste Element einer Liste.

  Beispiel: `hd [1,2,3] = 1`

- `tl Xs`: `tl` („Tail") entfernt das erste Element aus einer Liste.

  Beispiel: `tl [1,2,3] = [2,3]`

- `cons Xs`: `cons` fügt ein Element am Beginn einer Liste oder eines Tupels ein.

  Beispiel: `cons 1 [2,3] = [1,2,3]`

- `push Xs`: `push` fügt genau wie `cons` ein Element am Beginn einer Liste oder eines Tupels ein; allerdings ist die Reihenfolge der Argumente hier umgekehrt.

  Beispiel: `push [2,3] 1 = [1,2,3]`

- `pop Xs`: `pop` entfernt das erste Element aus einer Liste oder einem Tupel. Die `pop`-Funktion arbeitet genau wie `tl`, kann aber auch auf Tupel angewendet werden. Zusammen werden `push` und `pop` zur Realisierung so genannter „Stacks"verwendet.

  Beispiel: `pop [1,2,3] = [2,3]`

- `take N Xs`, `takewhile P Xs`: `take` liefert die Liste der ersten `N` Elemente der Liste `Xs`, während `takewhile` die Liste der Anfangs-Elemente von `Xs` bestimmt, die das Prädikat `P` erfüllen.

  Beispiel: Die ersten drei Elemente einer Liste:
  `take 3 [-3,-2,-1,0,1,2,3] = [-3,-2,-1]`

  Die negativen Elemente am Beginn einer Liste:
  `takewhile (<0) [-3,-2,-1,0,1,2,3] = [-3,-2,-1]`

- `drop N Xs`, `dropwhile P Xs`: `drop` und `dropwhile` sind die Gegenstücke von `take` und `takewhile`, die Elemente vom Beginn einer Liste entfernen.

  Beispiel: Eine Liste ohne die ersten drei Elemente:
  `drop 3 [-3,-2,-1,0,1,2,3] = [0,1,2,3]`

  Entferne die negativen Elemente am Beginn einer Liste:
  `dropwhile (<0) [-3,-2,-1,0,1,2,3] = [0,1,2,3]`

- `all P Xs`, `any P Xs`: `all` liefert `true` genau dann wenn alle Elemente von `Xs` das Prädikat `P` erfüllen, `any` liefert `true` genau dann wenn mindestens ein Element `P` erfüllt.

  Beispiel: Bestimme, ob alle Elemente einer Liste positiv sind:
  `all (>0) [-1,0,1] = false`

  Bestimme, ob mindestens ein Element positiv ist:
  `any (>0) [-1,0,1] = true`

- `foldl F A Xs`, `foldr F A Xs`: `foldl` und `foldr` wenden eine binäre Operation `F` beginnend mit einem Startwert `A` auf alle Elemente einer Liste `Xs` an; der Rückgabewert ist das Ergebnis der letzten Anwendung von `F`, oder der Startwert, falls `foldl`/`foldr` auf die leere Liste angewendet wird. Die beiden Funktionen unterscheiden sich darin, wie die rekursiven Anwendungen von `F` geklammert werden: bei `foldl` wird „nach links" geklammert (also z.B. `foldl F 0 [1,2] = F (F 0 1) 2`), bei `foldr` „nach rechts" (`foldr F 0 [1,2] = F 1 (F 2 0)`).

  Beispiel: Berechnung der Summe aller Listen-Elemente:
  `foldl (+) 0 [1,2,3] = 6`

  Die Summen-Funktion ist übrigens auch direkt unter dem Namen `sum` verfügbar:
  `sum [1,2,3] = 6`

Wir finden in der Standard-Bibliothek auch einige nützliche Funktionen zur Konstruktion von Listen:

- `mklist X N`: `mklist` konstruiert eine Liste von `N` `X`'s.

  Beispiel: Liste mit drei Nullen:
  `mklist 0 3 = [0,0,0]`

- `nums N M`: `nums` liefert die Liste aller Zahlen zwischen `N` und `M`, in Einser-Schritten.

  Beispiel: Liste aller ganzen Zahlen von `0` bis `10`:
  `nums 0 10 = [0,1,2,3,4,5,6,7,8,9,10]`

- `numsby K N M`: `numsby` funktioniert wie `nums`, erlaubt aber die Angabe einer Schrittweite `K`.

  Beispiel: Liste aller geraden Zahlen von `0` bis `10`:
  `numsby 2 0 10 = [0,2,4,6,8,10]`

- `iter N F A`: `iter` liefert ausgehend von einem Startwert `A` die Liste der ersten `N` wiederholten Anwendungen von `F` (also `A`, `F A`, `F (F A)`, ...).

  Beispiel: Die ersten sieben Zweierpotenzen:
  `iter 7 (2*) 1 = [1,2,4,8,16,32,64]`

- `while P F A`: `while` liefert ausgehend von einem Startwert `A` die Liste aller wiederholten Anwendungen von `F`, die das Prädikat `P` erfüllen.

  Beispiel: Liste aller Zweierpotenzen ≤ `1000`:
  `while (<=1000) (2*) 1 = [1,2,4,8,16,32,64,128,256,512]`

- `zip Xs Ys`: `zip` bildet die Liste der Paare entsprechender Elemente der Listen `Xs` und `Ys`.

  Beispiel: Tabelle der ersten sieben Zweierpotenzen:
  ```
  zip (nums 0 6) (iter 7 (2*) 1) =
    [(0,1),(1,2),(2,4),(3,8),(4,16),(5,32),(6,64)]
  ```

- `unzip XYs`: `unzip` ist das Gegenstück von `zip`, das eine Liste von Paaren in ein Paar von Listen zerlegt.

  Beispiel: Zerlege die oben berechnete Tabelle der Zweierpotenzen:
  ```
  unzip [(0,1),(1,2),(2,4),(3,8),(4,16),(5,32),(6,64)] =
    ([0,1,2,3,4,5,6],[1,2,4,8,16,32,64])
  ```

- `listof X C`: `listof` ist eine allgemeine Listen-Konstruktions-Funktion, mit der Listen auf eine Weise spezifiziert werden können, die der mathematischen Beschreibung einer Menge entspricht (so genannte „list comprehensions"). Dabei kann eine „Lauf-Variable" nacheinander mit den Werten einer Liste belegt werden, und es werden alle Elemente herausgefiltert, die die angegebenen Bedingungen nicht erfüllen.

  Beispiel: Liste aller Primzahl-Paare zwischen `1` und `100`:
  ```
  listof (I,I+2) (I in nums 1 100, isprime I and isprime (I+2)) =
   [(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61),(71,73)]
  ```

  (Hier wird die Standardbibliotheks-Funktion `isprime` verwendet, um festzustellen, bei welchen Elementen es sich tatsächlich um Primzahl-Paare handelt.)

3.6 Gleichungen

Mittels der eingebauten Funktionen von Q lassen sich bereits viele nützliche Berechnungen vornehmen. Wir benutzen dabei den Interpreter wie eine Art Taschenrechner. Um aber neue Funktionen zu definieren, müssen wir ein Skript, d.h. ein Programm, schreiben. In Q ist das Programmieren neuer Funktionen verhältnismäßig einfach. Alle Definitionen haben die Form von *Gleichungen*. Trotz dieser einfachen Form ist die Programmiersprache Q im technischen Sinne „universell"; d.h., es lassen sich alle Funktionen realisieren, die man überhaupt in irgendeiner Programmiersprache programmieren kann.

Jede Gleichung besteht aus zwei Ausdrücken, der *linken* und *rechten Seite*, die durch das Symbol = voneinander getrennt sind. Am Ende jeder Gleichung steht ein Semikolon. Eine Gleichung kann auch eine Bedingung der Form `if` *Ausdruck* oder den Zusatz `otherwise` zur Kennzeichnung eines Standard-Falls enthalten; außerdem können mehrere aufeinanderfolgende Gleichungen mit derselben linken Seite zusammengefasst werden. (Das Schlüsselwort `otherwise` wird vom Interpreter wie ein Kommentar behandelt, es dient nur dazu, bestimmte Definitionen lesbarer zu machen.)

Gleichungen sind als Ersetzungsregeln zu lesen, die stets von links nach rechts angewendet werden. Immer dann, wenn ein Ausdruck mit der Form der linken Seite auftritt, kann er durch die entsprechende rechte Seite ersetzt werden. Betrachten wir dazu zunächst ein einfaches Beispiel: die Funktion `sqr` (für „square") soll es ermöglichen, eine (ganze oder Fließkomma-) Zahl zu quadrieren. Dazu soll das Argument der `sqr`-Funktion mit sich selbst multipliziert werden. Die entsprechende Gleichung lautet:

```
sqr X                  = X*X;
```

Einfach genug, oder nicht? Tatsächlich sieht unser „Programm" eher wie eine mathematische Definition aus, und tatsächlich ist es das auch. Die linke Seite der Gleichung steht für einen beliebigen Ausdruck der Form `sqr X`, wobei die Variable `X` ein Platzhalter für das tatsächliche Argument der `sqr`-Funktion ist. (Zur Erinnerung: `X` wird vom Interpreter automatisch als eine Variable erkannt, da es sich um ein Symbol handelt, das mit einem Großbuchstaben anfängt.) Um die Definition zu testen, erstellen Sie ein Skript mit obiger Zeile und führen Sie es aus:

```
==> sqr 4
16

==> sqr 3.5
12.25
```

Auf einen wichtigen Unterschied zwischen Q und praktisch allen anderen Programmiersprachen wollen wir gleich an dieser Stelle hinweisen. Q-Definitionen sind *symbolische Ersetzungsregeln*, die auf beliebige Ausdrücke (auch solche mit Variablen) angewendet werden können. Z.B.:

```
==> sqr (Y+2)
(Y+2)*(Y+2)
```

Tatsächlich kann auf der linken Seite einer Gleichung im Prinzip ein beliebiger Ausdruck stehen, auch verschachtelte Ausdrücke und Ausdrücke, die Operatoren enthalten. Kombinieren wir spaßeshalber einmal unsere Definition der `sqr`-Funktion mit einigen bekannten Rechenregeln für arithmetische Ausdrücke:

```
/* bsp02.q: symbolische Definitionen */

sqr X               = X*X;

// Distributiv-Gesetz

(A+B)*C             = A*C+B*C;
A*(B+C)             = A*B+A*C;

// Assoziativ-Gesetz

A+(B+C)             = (A+B)+C;
A*(B*C)             = (A*B)*C;
```

Wir erhalten nun:

```
==> sqr (Y+2)
Y*Y+Y*2+2*Y+4
```

Man erkennt, dass die zusätzlichen Gleichungen verwendet wurden, um den Ausdruck `(Y+2)*(Y+2)` weiter zu „vereinfachen". (Wir könnten an dieser Stelle weitere symbolische Regeln hinzufügen, um den Ausdruck in eine noch einfachere Form zu bringen, wollen es aber dabei bewenden lassen.)

Betrachten wir als nächstes ein etwas schwierigeres Beispiel, die so genannte *Fibonacci-Funktion*, die im Zusammenhang mit natürlichen Wachstumsvorgängen und dem „goldenen Schnitt" eine Rolle spielt. Die Fibonacci-Funktion wird wie folgt definiert:

```
/* bsp03.q: Fibonacci-Funktion */

fib 0                = 0;
fib 1                = 1;
fib N                = fib (N-2) + fib (N-1) if N>1;
```

Zu dieser Definition sind zwei Dinge anzumerken:

1. Es handelt sich um eine *rekursive* Definition, d.h. die Funktion `fib` wird durch sich selbst definiert. Damit dies funktioniert, muss die Definition „induktiv" sein, d.h. die rekursiven Anwendungen von `fib` müssen „einfacher" zu berechnen sein als die linke Seite – es macht z.B. wenig Sinn, `fib N` durch `fib (N+1)` definieren zu wollen. Unsere Definition erfüllt diese Bedingung, da nur auf kleinere Werte von `fib` zurückgegriffen wird, und die ersten beiden Werte durch die ersten beiden Gleichungen festgelegt sind.
2. Die letzte Gleichung in der Definition ist eine *bedingte Gleichung*, deren Anwendbarkeit durch eine logische Bedingung beschränkt wird. Die Bedingung ist hier, dass der Parameter `N` größer als `1` ist, was durch den Zusatz `if N>1` ausgedrückt wird.

Um die ersten paar Werte der Fibonacci-Funktion zu berechnen, können wir eine vordefinierte Listenfunktion, die `map`-Funktion verwenden (vgl. Abschnitt 3.5). Mit dieser Funktion lässt sich eine beliebige Funktion auf alle Elemente einer Liste anwenden:

```
==> map fib [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
```

Man erkennt hier leicht das Bildungsgesetz der Fibonacci-Folge: Jedes Glied der Folge ist die Summe der beiden vorherigen Glieder. (So ist die Funktion ja auch definiert.)

Rekursion und Iteration

Unsere Definition der Fibonacci-Funktion ist zwar korrekt, hat aber einen wesentlichen Mangel, der schnell deutlich wird, wenn Sie versuchen, `fib N` für größere Werte von `N` zu berechnen: die Rechenzeit steigt sehr schnell an. Dies liegt daran, dass die rekursive Gleichung Nummer 3 der Definition *zwei* rekursive Anwendungen von `fib` enthält. Tatsächlich ist die Anzahl der Rechenschritte zur Bestimmung von `fib N` proportional zu `fib N` selbst, und wie man an der Tabelle der ersten paar Folgenglieder schon erkennen kann, wächst diese Funktion recht schnell. Mit etwas Mathematik zeigt man leicht, dass das Wachstum sogar *exponentiell* ist, d.h. proportional zu 2^N.

Nun gibt es Funktionen, deren Berechnung notwendigerweise exponentielle Rechenzeit benötigt. Die Fibonacci-Funktion lässt sich aber glücklicherweise auch so definieren, dass man nur eine zu `N` proportionale Anzahl von Rechenschritten braucht. Wie man an der rekursiven Definition unmittelbar sieht, brauchen wir nämlich zur Berechnung jedes Wertes der Funktion jeweils nur die zwei vorhergehenden Werte. Wenn wir diese Werte irgendwie „zwischenspeichern" können, so brauchen wir in jedem Berechnungsschritt nur eine zusätzliche Operation (eine Addition) auszuführen.

Die Lösung besteht darin, eine zusätzliche Funktion fib2 einzuführen, die die zu speichernden Werte als Argumente mitführt:

```
/* bsp04.q: Fibonacci-Funktion, iterative Version */

fib N                = fib2 0 1 N;
fib2 A B N           = fib2 B (A+B) (N-1) if N>0;
                     = A otherwise;
```

In der ersten Gleichung rufen wir fib2 mit den ersten beiden Funktionswerten (A=0, B=1) und dem gewünschten Parameter N auf. Die zweite Gleichung ist eine Art „Schleife", in der die Funktion fib2 sich immer wieder selbst „aufruft", bis die notwendige Anzahl von Additionen durchgeführt worden ist. Sobald dies der Fall ist, liefert die dritte Gleichung das Endergebnis. Solch einen Rechenprozess bezeichnet man auch als *Iteration* oder *End-Rekursion.*

Um die beiden Definitionen der Fibonacci-Funktion zu vergleichen, laden Sie zunächst das Skript bsp03.q, dann bsp04.q, und berechnen Sie jeweils z.B. fib 30. Der Unterschied in den Rechenzeiten sollte offensichtlich sein.

Ein weiterer wünschenswerter Effekt unserer iterativen fib-Version besteht übrigens darin, dass sie bei der Auswertung im Interpreter weniger Speicherplatz benötigt. Für die erste fib-Version muss der Interpreter nämlich Zwischenergebnisse speichern, so lange er die rekursiven fib-Anwendungen bearbeitet. Der benötigte Speicherplatz wächst hier mit der Größe von N. Dies wird in der zweiten Version vermieden, da sich die Funktion fib2 nach Berechnung der neuen Parameter-Werte direkt selbst aufruft; der zusätzliche Speicherbedarf ist hier eine konstante Größe.

Definition von Listen-Funktionen

Um zu zeigen, dass Gleichungs-Definitionen nicht nur zur Berechnung von numerischen Funktionen geeignet sind, betrachten wir abschließend einige einfache Listen-Funktionen. Wie wir in Abschnitt 3.4 gesehen haben, haben Listen entweder die Form [] (leere Liste) oder die Form [X|Xs], wobei X für das erste Element der Liste und Xs für die Liste der restlichen Elemente steht. Zwei nützliche Listen-Funktionen sind hd („head") und tl („tail"), mit denen man das Anfangs-Element X und die Rest-Liste Xs einer nichtleeren Liste bestimmen kann. Wir können diese Funktionen wie folgt definieren. (Da auch die Standard-Bibliothek Definitionen für hd und tl enthält, verwenden wir hier eine explizite private-Deklaration, um sicherzustellen, dass der Interpreter für unser Skript neue private Funktionssymbole hd und tl verwendet.)

```
/* bsp05.q: hd- und tl-Funktionen */

private hd Xs, tl Xs;

hd [X|_]   = X;
tl [_|Xs] = Xs;
```

In diesen Definitionen haben wir die so genannte *anonyme Variable* _ verwendet. Die anonyme Variable steht für einen beliebigen Wert, den wir für die weitere Berechnung nicht benötigen. So ist es zum Beispiel für die Definition von hd egal, welchen Wert die Rest-Liste hat; wir benötigen ja nur den Wert des Anfangs-Elements. Man beachte, dass die anonyme Variable nur auf der linken Seite verwendet werden kann, niemals auf der rechten Seite oder im Bedingungs-Teil einer Gleichung.

(Tatsächlich kann die anonyme Variable auf der linken Seite mehrfach verwendet werden, und steht dann u.U. jedesmal für einen anderen Wert.)

Das folgende Beispiel zeigt die `hd`- und `tl`-Funktionen in Aktion:

```
==> hd [1,2,3]
1

==> tl [1,2,3]
[2,3]
```

Ein etwas interessanteres Beispiel einer *rekursiven* Listen-Funktion ist die Funktion `map`, die weiter oben schon verwendet wurde, um eine Funktion auf jedes Element einer Liste anzuwenden. Wir können `map` wie folgt definieren (auch hier setzen wir wieder eine `private`-Deklaration ein, da die Standard-Bibliothek bereits eine Funktion namens `map` bereitstellt):

```
/* bsp06.q: map-Funktion */

private map F Xs;

map F []            = [];
map F [X|Xs]        = [F X|map F Xs];
```

Bei `map` müssen wir den Fall der leeren Liste `[]` und den der zusammengesetzten Liste `[X|Xs]` unterscheiden. Im ersten Fall ist das Ergebnis einfach wieder die leere Liste. Im zweiten Fall wird die als Parameter `F` übergebene Funktion auf das erste Element angewendet und die Funktion `map` dann rekursiv für die Rest-Liste aufgerufen.

Um die Korrektheit der Definition zu testen, wenden wir `map` zunächst auf ein symbolisches Funktions-Argument an, z.B.:

```
==> map F [1,2,3]
[F 1,F 2,F 3]
```

Versuchen wir es nun mit einer konkreten Funktion, z.B. der „Verdopplungs"-Funktion `(*2)`:

```
==> map (*2) [1,2,3]
[2,4,6]
```

Um die Arbeitsweise von `map` genauer zu verstehen, können wir auch den „Debugger" verwenden, der die einzelnen Berechnungsschritte während der Auswertung eines Ausdrucks anzeigt. Der Debugger wird später auch nützlich sein, um „Bugs" in fehlerhaften Skripts zu entdecken. (Daher der Name.) Um den Debugger zu aktivieren, geben Sie im Interpreter das Kommando `debug on` ein; mit `debug off` schalten Sie den Debugger wieder aus. Wenn der Debugger aktiv ist, zeigt er seinen Prompt, einen Doppelpunkt am Beginn der Zeile. Hier können Sie einige einfache Kommandos eingeben; versuchen Sie z.B. einmal das Kommando `?` für „Hilfe". Um die laufende Auswertung abzubrechen, geben Sie das Kommando `h` für „Halt" ein. Zur Fortsetzung der Auswertung betätigen Sie jeweils einfach die Return-Taste. Sobald der Interpreter mit der Anwendung einer bestimmten

Gleichung beginnt oder zu ihr zurückkehrt, wird die Gleichung im Debugger angezeigt, unter Angabe des Skript-Namens und der Zeile, in der sich die Gleichung befindet. Wenn die Auswertung der rechten Seite beendet ist, wird das jeweilige Zwischenergebnis in einer Meldung der Form „** *linke Seite* ==> *rechte Seite*“ vermerkt. Beispiel:

```
==> debug on

==> map F [1,2]
  0>  bsp06.q, line 6: map F [1,2]  ==>  [F 1|map F [2]]
(type ? for help)
:
  1>  bsp06.q, line 6: map F [2]  ==>  [F 2|map F []]
:
  2>  bsp06.q, line 5: map F []  ==>  []
:
**  map F []  ==>  []
  1>  bsp06.q, line 6: map F [2]  ==>  [F 2|map F []]
:
**  map F [2]  ==>  [F 2]
  0>  bsp06.q, line 6: map F [1,2]  ==>  [F 1|map F [2]]
:
**  map F [1,2]  ==>  [F 1,F 2]
[F 1,F 2]

==>
```

3.7 Auswertung von Ausdrücken

In diesem Abschnitt befassen wir uns etwas genauer mit der Frage, wie der Q-Interpreter Ausdrücke auswertet. Wie wir bereits gesehen haben, wendet der Interpreter dazu Gleichungen an, wobei stets die linke Seite einer Gleichung durch die entsprechende rechte Seite ersetzt wird. Dieser Ersetzungsprozess wird fortgesetzt, bis keine Gleichungen mehr anwendbar sind. Dies entspricht ziehmlich genau der algebraischen Manipulation von Formeln, so wie man dies in der Schule lernt. Man bezeichnet diesen Prozess auch als *Termersetzung*, und betrachtet die Gleichungen in diesem Zusammenhang als *Termersetzungs-Regeln*.

Betrachten wir zunächst den Ersetzungs-Schritt selbst. Als erstes muss der Interpreter feststellen, welche linke Seite einer Gleichung auf einen Teil-Ausdruck des auszuwertenden Ausdrucks „passt“, und welche Werte die Variablen der linken Gleichungs-Seite dafür annehmen müssen. Ein passender Teilausdruck des auszuwertenden Ausdrucks wird auch als *Redex* bezeichnet. Um einen Ersetzungs-Schritt durchzuführen, eine so genannte *Reduktion*, wird der Redex durch das so genannte *Redukt* ersetzt, die entsprechende rechte Seite der Gleichung, wobei für die Variablen der rechten Seite die entsprechenden Werte eingesetzt werden.

Wenn wir z.B. die Gleichung `sqr X = X*X` aus dem vorangegangenen Abschnitt auf den Ausdruck `sqr 3` anwenden, so ist der gesamte Ausdruck ein Redex, wobei für die Variable `X` der Wert `3` einzusetzen ist. Die Anwendung der Gleichung liefert das Redukt `X*X = 3*3`. Wir haben hier also die Reduktion `sqr 3` ⇒ `3*3` durchgeführt. Dies funktioniert genauso, wenn der Redex ein Teilausdruck des auszuwertenden Ausdrucks ist, und die einzusetzenden Variablen-Werte selbst wieder zusammengesetzte Ausdrücke sind, z.B.: <u>`sqr (Y+2)`</u>`*2` ⇒ <u>`(Y+2)*(Y+2)`</u>`*2`, wobei Redex und Redukt jeweils durch Unterstreichen hervorgehoben wurden.

Dieser Prozess wird in entsprechender Weise auch auf die „eingebauten“ Operationen angewendet, wobei diese behandelt werden, als ob sie durch eine große Zahl „eingebauter Gleichungen“ definiert

wären. Z.B. liefert die eingebaute Definition des Multiplikations-Operators für den Ausdruck `3*3` die Zahl `9`. Dies ist eine Konstante, auf die keine weiteren Gleichungen mehr angewendet werden können. Man bezeichnet solche nicht mehr weiter reduzierbaren Ausdrücke auch als *Normalformen*, und betrachtet sie als den Wert eines Ausdrucks. Man beachte, dass Normalformen nicht unbedingt elementare Ausdrücke sein müssen; z.B. ist die Normalform des Ausdrucks `sqr (sin Y)` der zusammengesetzte Ausdruck `sin Y*sin Y`. Auch gibt es Ausdrücke, die überhaupt keine Normalform haben; wie auch bei konventionellen Programmiersprachen kann die Auswertung eines Ausdrucks in eine endlose Rekursion führen, die nie eine Antwort liefert. (Man versuche dies z.B. mit dem Ausdruck `loop 0` und der Gleichung `loop N = loop (N+1)`.)

Richtig schwierig wird die Sache allerdings erst, wenn es im auszuwertenden Ausdruck mehrere Redizes gibt, oder mehrere anwendbare Gleichungen. Betrachten wir z.B. den Ausdruck `sqr (3+3)`. Sollen wir zunächst `3+3` reduzieren oder den gesamten Ausdruck? Es gibt hier drei mögliche Rechenwege:

- `sqr (3+3)` ⇒ `sqr 6` ⇒ `6*6` ⇒ `36`.
- `sqr (3+3)` ⇒ `(3+3)*(3+3)` ⇒ `6*(3+3)` ⇒ `6*6` ⇒ `36`.
- `sqr (3+3)` ⇒ `(3+3)*(3+3)` ⇒ `(3+3)*6` ⇒ `6*6` ⇒ `36`.

In diesem Fall ist der schließlich berechnete Wert, die Normalform, dieselbe, nicht aber die Anzahl der notwendigen Berechnungsschritte. Im allgemeinen Fall kann durchaus auch das Endergebnis und sogar die Existenz einer Normalform vom gewählten Rechenweg abhängen. Die leidvollen Erfahrungen vieler Gymnasiasten, die sich in Mathe-Klausuren mit komplexen algebraischen Umformungen herumquälen müssen, bestätigen dies.

Um solche Mehrdeutigkeiten aufzulösen, verwendet der Q-Interpreter eine so genannte *Auswertungs-Strategie*. Diese bestimmt genau, welcher Redex in jedem Schritt reduziert werden soll, und welche Gleichung dafür anzuwenden ist. Die Standard-Auswertungs-Strategie des Q-Interpreters lässt sich kurz mit „von links nach rechts und innen nach außen“ und „die erste anwendbare Gleichung“ zusammenfassen. Genauer:

- Ausdrücke werden stets *von links nach rechts* ausgewertet, und *von innen nach außen*. Man nennt diese Berechnungs-Reihenfolge auch „leftmost-innermost“, da immer derjenige Redex gewählt wird, der am weitesten links liegt und der keinen weiteren Redex mehr enthält. Eine andere Bezeichnung dafür lautet „call by value“, da die Argumente einer Funktion vor der Funktionsanwendung ausgewertet werden, eine Funktion also stets mit den *Werten* ihrer Argumente „gefüttert“ wird. Dies entspricht der in vielen Programmiersprachen gebräuchlichen Auswertungs-Strategie, und ist auch häufig die Art und Weise, in der Menschen Berechnungen manuell ausführen. Der erste der oben beschriebenen Rechenwege ist also derjenige, den der Q-Interpreter bei der Auswertung des Ausdrucks `sqr (3+3)` tatsächlich verwendet.
- Sind mehrere Gleichungen anwendbar, so wird immer die erste anwendbare Gleichung verwendet, gemäß der Reihenfolge der Gleichungen im Skript. Dabei haben „eingebaute“ Definitionen stets Vorrang vor den Gleichungen eines Skripts. Eine wichtige Konsequenz dieser Regel ist, dass „speziellere“ Gleichungen vor „allgemeineren“ aufgelistet werden müssen. Als Beispiel betrachte man die Definition der `fib2`-Funktion im vorangegangenen Abschnitt. Hier wird die speziellere Gleichung für den Fall `N>0` vor dem „Standard-Fall“ aufgeführt. (Würde man die Reihenfolge der beiden Gleichungen umkehren, so wäre der Wert der `fib2`-Funktion stets `0`.)

Es bleibt anzumerken, dass die oben skizzierte „call by value“-Strategie nicht die einzige Auswertungs-Strategie ist, die der Q-Interpreter kennt; man kann auch mittels so genannter „Spezialformen“ die Reihenfolge der Auswertung von Argument-Ausdrücken selber kontrollieren,

was für spezielle Anwendungen nützlich ist. Wir werden dies aber im folgenden nicht verwenden. Für weitere technische Details der Ausdrucks-Auswertung, die den Rahmen dieser Einführung sprengen würden, verweisen wir den Leser wieder auf das Q-Handbuch.

An dieser Stelle müssen wir auch auf einen weiteren wesentlichen Unterschied zwischen Q und anderen funktionalen Programmiersprachen hinweisen. In Sprachen wie Haskell und ML gibt es eine grundlegende Dichotomie zwischen Funktions- und so genannten „Konstruktor“-Symbolen. Letztere dienen ausschließlich dazu, Werte zu repräsentieren, während Funktions-Symbole immer vollständig definierte Funktionen darstellen, die letztlich irgendeinen Wert liefern müssen. Eine Funktions-Anwendung stellt also immer eine Anweisung zur Berechnung eines Wertes dar, und steht nie für sich selbst. Wenn eine Funktion bei der Berechnung auf irgendeine „Ausnahme-Situation“ stößt, die es unmöglich macht, die Berechnung fortzusetzen (z.B. Division durch Null), so wird ein entsprechender „Laufzeit-Fehler“ ausgegeben.

Als reine Termersetzungs-Sprache unterscheidet Q dagegen *nicht* zwischen Konstruktoren und „definierten“ Funktionen. Tatsächlich kennt der Q-Interpreter das Konzept einer Funktions-Definition überhaupt nicht; er wendet nur „blind“ die Gleichungen eines Skripts auf den auszuwertenden Ausdruck an. Ist keine Gleichung mehr anwendbar, so ist der resultierende Ausdruck in Normalform. Normalformen stehen immer für sich selbst, und stellen „Werte“ in der Programmiersprache Q dar. Die eingebauten „konstanten“ Objekte (Zahlen, Zeichenketten, sowie aus Konstanten zusammengesetzte Listen und Tupel) sind stets Normalformen, dies gilt aber auch für alle Symbole und Funktions-Anwendungen, die nicht durch eine Gleichung „definiert“ sind. Insbesondere führen Fehlerbedingungen wie Division durch Null *nicht* standardmäßig zur Generierung eines Laufzeit-Fehlers, sondern der entsprechende Ausdruck stellt dann eine Normalform dar. Z.B.:

```
==> 23/0
23/0
```

Es mag auf den ersten Blick etwas befremdlich erscheinen, dass ein Ausdruck wie `23/0` in Q tatsächlich ein zulässiger Wert ist, dies stellt aber eine der wichtigsten Eigenschaften einer auf Termersetzung beruhenden Programmiersprache dar, und erhöht die Flexibilität der Programmiersprache erheblich. In Q ist es jederzeit möglich, die Definition einer eingebauten oder vom Benutzer definierten Operation durch weitere Gleichungen zu „verfeinern“. Wenn gewünscht, kann man also ohne weiteres eine „Fehler-Regel“ zur Behandlung bestimmter Ausnahmefälle hinzufügen, z.B.:

```
X/0                  = throw "Division durch Null!";
```

Tatsächlich gibt es bestimmte kritische Situationen, in denen auch der Q-Interpreter von selbst einen Laufzeit-Fehler generiert, z.B. dann, wenn der verfügbare Hauptspeicher zur Neige geht, wenn der Bedingungs-Teil einer Gleichung keinen Wahrheits-Wert als Ergebnis liefert, oder wenn der Benutzer mittels [Strg][C] die Auswertung abbricht. Ein Q-Skript kann auch selbst Ausnahme-Fehler wie oben gezeigt mit der eingebauten Funktion `throw` generieren. Tritt ein Ausnahme-Fehler auf, so kann die Auswertung des aktuellen Ausdrucks nicht fortgesetzt werden; es ist aber möglich, solche Fehlerbedingungen mit der eingebauten Funktion `catch` abzufangen. Zur genaueren Beschreibung dieser Funktionen wird auf das Q-Handbuch verwiesen.

3.8 Variablen-Definitionen

Wie wir in den beiden vorangegangenen Abschnitten gesehen haben, finden Variablen auf der linken Seite einer Gleichung Verwendung als „Platzhalter" für die tatsächlichen Werte in einem auszuwertenden Ausdruck. Man bezeichnet diese Variablen auch als *gebunden*. Variablen können aber auch *frei* auftreten, nämlich als Variablen innerhalb eines auszuwertenden Ausdrucks, oder als Variablen auf der rechten Seite (oder im Bedingungs-Teil) einer Gleichung, die nicht auf der linken Seite auftreten. Als Beispiel einer Gleichung, die auf der rechten Seite eine freie Variable enthält, betrachte man die folgende Definition:

```
foo X                = C*X;
```

Die Variable `C` tritt hier frei auf der rechten Seite auf. Solange wir dieser Variablen keinen Wert zuordnen, steht sie einfach für sich selbst. Z.B:

```
==> foo 99
C*99
```

Um einer freien Variablen einen Wert zuzuordnen, können wir eine *Variablen-Definition* verwenden. Diese sieht syntaktisch wie eine Gleichung aus, wird aber mit dem Schlüsselwort `def` eingeleitet. Man kann eine Variable direkt im Interpreter definieren:

```
==> def C = 2; foo 99
198
```

Wie man sieht, wurde nun der Wert von `C` verwendet, um das Endergebnis `C*99 = 2*99 = 198` zu berechnen. Wir können die Definition von `C` auch wieder rückgängig machen; dafür gibt es den Befehl `undef`:

```
==> undef C; foo 99
C*99
```

Im Interpreter werden Variablen häufig verwendet, um Zwischenergebnisse zu speichern, z.B.:

```
==> def X = 16.3805*5; sqrt X/.05
181.0
```

Nach dem Schlüsselwort `def` können mehrere Variablen-Definitionen stehen, die mit einem Komma voneinander abgetrennt werden. Auch `undef` erlaubt die Angabe mehrerer Variablen-Symbole.

```
==> def X = 16.3805*5, Y = .05; sqrt X/Y
181.0

==> undef X, Y
```

Wie bei Gleichungen kann die linke Seite einer Variablen-Definition auch ein zusammengesetzter Ausdruck sein, wobei die Variablen in der linken Seite mit den entsprechenden Werten der rechten Seite belegt werden:

```
==> def (X,Y) = (16.3805*5, .05); sqrt X/Y
181.0
```

Dabei muss der Ausdruck auf der rechten Seite natürlich „passen“, sonst zeigt der Interpreter einen Fehler an:

```
==> def (X,Y) = [16.3805*5, .05]
! Value mismatch in definition
>>> def (X,Y) = [16.3805*5, .05]
                                 ^
```

Variablen-Definitionen können auch in einem Skript stehen; hier müssen sie wie eine Gleichung mit einem Semikolon abgeschlossen werden:

```
def C = 2;
foo X                = C*X;
```

Variablen-Definitionen in einem Skript werden nur *einmal* ausgewertet (nämlich zu der Zeit, wenn das Skript vom Interpreter geladen wird) und werden in der Reihenfolge ihres Auftretens im Skript abgearbeitet, so dass jede Definition auf bereits vorher definierte Variablen zurückgreifen kann. Man verwendet solche Definitionen häufig dazu, um irgendwelche Tabellen oder spezielle Datenobjekte zu speichern, die innerhalb des Skripts verwendet werden, und deren Berechnung einen gewissen Rechenaufwand erfordert oder Operationen mit „Nebeneffekten“ involviert. Entsprechende Beispiele werden wir später kennenlernen.

Man beachte auch, dass Variablen zwar mit `def` und `undef` innerhalb des Interpreters verändert werden können, niemals aber innerhalb einer Gleichung. (Es ist allerdings möglich, eine Variable „lokal“, d.h. innerhalb einer Regel neu zu definieren, s.u.) Daher hat eine Variable während der Auswertung eines Ausdrucks stets den selben Wert (obwohl sie bei *zwei* Auswertungen des selben Ausdrucks verschiedene Werte annehmen kann). Man nennt dies auch „referentielle Transparenz“. Referentielle Transparenz stellt eine wesentliche Eigenschaft moderner funktionaler Programmiersprachen dar. Der Ausdruck bedeutet, dass man stets „Gleiches mit Gleichem“ ersetzen kann, um einen Ausdruck auszuwerten. (Q ist allerdings nur in bedingtem Maße referentiell transparent, da einige Operationen so genannte „Nebeneffekte“ haben. Dies gilt insbesondere für die Ein-/Ausgabe-Operationen.)

Lokale Variablen

Die mittels `def` definierten Variablen eines Skripts werden auch *globale Variablen* genannt, da ihre Gültigkeit sich auf alle Gleichungen des Skripts erstreckt. Daneben kennt Q auch *lokale Variablen*, deren Gültigkeit jeweils auf die rechte Seite (und den Bedingungs-Teil) einer Gleichung beschränkt ist. Diese sind nützlich, wenn der gleiche Wert auf der rechten Seite mehrfach verwendet werden soll. Mittels einer lokalen Definition, auch „`where`-Klausel“ genannt, kann man einen solchen Wert einer Variablen zuweisen, so dass der Wert nur einmal berechnet werden muss. Die Form

einer lokalen Definition entspricht der einer `def`-Anweisung, außer dass die Definition mit dem Schlüsselwort `where` beginnt und am Ende einer Gleichung steht.

Betrachten wir als Beispiel die Auflösung einer quadratischen Gleichung der Form

$$x^2 + px + q = 0.$$

Die Lösungen einer solchen Gleichung sind bekanntlich gegeben durch

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}\,.$$

Damit überhaupt eine (reelle) Lösung existiert, muss der Ausdruck unter der Wurzel, die so genannte *Diskriminante*

$$D = \frac{p^2}{4} - q$$

einen Wert ≥ 0 haben. Wir können also die Funktion `solve`, die als Argumente die Parameter p und q erhält und als Ergebnis ein Tupel mit den beiden Lösungen liefert (sofern diese existieren), wie folgt definieren:

```
/* bsp07.q: quadratische Gleichungen */

solve P Q            = (-P/2 + sqrt D, -P/2 - sqrt D) if D >= 0
                         where D = P^2/4-Q;
```

Einige Beispiele für die Anwendung von `solve`:

```
==> solve 0 4                                // X^2+4 = 0 (keine reelle Lösung)
solve 0 4

==> solve 0 0                                // X^2 = 0 (eine Lösung)
(0.0,0.0)

==> solve 0 (-4)                             // X^2-4 = 0 (zwei Lösungen)
(2.0,-2.0)

==> solve 1 (-4)                             // X^2+X-4 = 0 (zwei Lösungen)
(1.56155281280883,-2.56155281280883)
```

Genau wie `def`-Anweisungen können auch `where`-Klauseln mehrere Definitionen umfassen, die mittels Kommas voneinander abgetrennt werden. Die einzelnen Definitionen werden in der Reihenfolge ihres Auftretens ausgewertet, und jede Definition kann auf alle bereits definierten Werte und die Variablen der linken Seite zugreifen. Wir können z.B. die mehrfache Berechnung des Werts von `P/2` vermeiden, indem wir dafür eine weitere Variable `P_2` einführen:

```
solve P Q            = (-P_2 + sqrt D, -P_2 - sqrt D) if D >= 0
                         where P_2 = P/2, D = P_2^2-Q;
```

Eine Gleichung kann allgemein auch mehrere Bedingungen und `where`-Klauseln in beliebiger Reihenfolge umfassen, die in der umgekehrten Reihenfolge ihres Auftretens ausgewertet werden. Dies

ist z.B. dann nützlich, wenn für den Bedingungs-Teil einer Gleichung einige Variablen definiert werden sollen, andere Variablen aber nur innerhalb der rechten Seite verwendet werden. Letztere kann man oberhalb der Bedingung definieren, sie werden dann nur berechnet, wenn die Bedingung gültig ist. Um z.B. eine weitere Variable `E` für die in der rechten Seite der Definition von `solve` zweimal auftretende Quadratwurzel der Diskriminante zu definieren, gehen wir wie folgt vor:

```
solve P Q            = (-P_2 + E, -P_2 - E) where E = sqrt D if D >= 0
                         where P_2 = P/2, D = P_2^2-Q;
```

Schließlich kann wie bei `def` auch die linke Seite einer lokalen Definition ein beliebiger zusammengesetzter Ausdruck sein, der mit dem Wert auf der entsprechenden rechten Seite verglichen wird. Solche Definitionen bilden gleichzeitig auch zusätzliche Bedingungen; die Gleichung kann nur dann angewendet werden, wenn der Ausdruck auf der linken Seite jeder Definition mit dem Wert der entsprechenden rechten Seite zusammenpasst (dies ist natürlich gewährleistet, wenn die linke Seite wie oben immer eine Variable ist).

Definitionen mit zusammengesetzter linker Seite sind insbesondere dann nützlich, wenn eine Funktion wie `solve` ein zusammengesetztes Ergebnis liefert, dessen Bestandteile für weitere Berechnungen auf der rechten Seite einer Gleichung benötigt werden. Beispiel:

```
test P Q             = (X1^2+P*X1+Q, X2^2+P*X2+Q)
                         where (X1,X2) = solve P Q;
```

Hier werden die Lösungen der `solve`-Funktion in die entsprechende quadratische Gleichung eingesetzt, um das Ergebnis zu kontrollieren. Das Ergebnis der `test`-Funktion ist `(0.0,0.0)`, wenn `solve` die Gleichung im Rahmen der Rechengenauigkeit exakt gelöst hat.

```
==> test 0 4                                // X^2+4 = 0, keine Lösung
test 0 4

==> test 1 (-4)                             // X^2+X-4 = 0, korrekt gelöst
(0.0,0.0)
```

Man beachte, dass die erste `test`-Rechnung einfach den eingegebenen Ausdruck zurückliefert, da für die gegebenen Parameter keine Lösung existiert und `solve` daher auch kein Tupel liefert, die `where`-Klausel in der Definition von `test` also „scheitert".

3.9 Datentypen

Unter einem *Datentyp* versteht man allgemein eine Klasse gleichartiger Werte, auf denen bestimmte Operationen in gleicher Weise arbeiten. Wie wir gesehen haben, kennt Q eine gewisse Anzahl eingebauter Datentypen, nämlich ganze Zahlen, Fließkommazahlen, Zeichenketten, Listen und Tupel. Außerdem kann man in Q auch eine Menge von Funktions-Symbolen und -Anwendungen als Datentyp vereinbaren. Wir wollen hier nur die wichtigsten Grundkonzepte der Q-Datentypen kurz vorstellen, weitere Details und ausführliche Beispiele findet man im Q-Handbuch.

Exkurs: Datentypen

So gut wie alle Programmiersprachen unterscheiden zwischen verschiedenen eingebauten Datentypen, und die meisten heutigen Programmiersprachen erlauben auch die Definition neuer Anwendungs-spezifischer Datentypen. Es gibt jedoch zwischen den Programmiersprachen wesentliche Unterschiede in der Art und Weise, wie Datentypen erkannt und angewendet werden.

Der wichtigste Unterschied ist der zwischen *statischer* und *dynamischer Typisierung*. In Programmiersprachen mit *statischer Typisierung*, wie z.B. C, Fortran und Pascal, ist der Typ eines Ausdrucks bereits zur Compilier-Zeit bekannt, und eine Variable kann stets nur den Wert eines vorgegebenen Datentyps annehmen. Dagegen liegt bei Programmiersprachen mit *dynamischer Typisierung*, wie z.B. Lisp, Prolog und Smalltalk, der Typ eines Ausdrucks erst zur Laufzeit, nach der Auswertung des Ausdrucks, fest. Hier kann eine Variable normalerweise Werte eines beliebigen Datentyps repräsentieren. Auch Q ist eine Programmiersprache mit dynamischer Typisierung.

Der Hauptvorteil der statischen Typisierung besteht darin, dass bereits der Compiler bestimmte Unstimmigkeiten in einem Programm, so genannte *Typ-Fehler*, ausfindig machen kann, was die Fehlersuche vereinfacht. Demgegenüber erlaubt die dynamische Typisierung eine größere Flexibilität bei der Definition von *polymorphen* Operationen, die auf Werte verschiedener Datentypen angewendet werden können. Compilierte Sprachen verwenden hauptsächlich statische Typisierung, während interpretierte Sprachen häufig mit dynamischer Typisierung arbeiten. Einige neuere compilierte Programmiersprachen wie Ada und C++ verwenden statische Typisierung zusammen mit zusätzlichen Elementen zur Konstruktionen von „generischen" Datentypen, um damit einen Teil der Flexibilität dynamischer Typisierung zu gewinnen. Darüberhinaus werden Datentypen in modernen statisch typisierenden funktionalen Sprachen wie Haskell und ML automatisch erkannt, so dass Datentypen nicht unbedingt deklariert werden müssen.

Eingebaute Datentypen

Wenden wir uns zunächst den eingebauten Datentypen von Q zu. Wie bereits gesagt, handelt es sich dabei um die Typen der ganzen Zahlen, Fließkommazahlen, Zeichenketten, Listen und Tupel, die mit den Symbolen `Int`, `Float`, `String`, `List` und `Tuple` bezeichnet werden. (Außerdem gibt es auch noch den Type `File` zur Darstellung von Datei-Objekten, auf diesen werden wir aber in dieser Einführung nicht näher eingehen.) Daneben gibt es noch den Typ `Num`, der sowohl Werte vom Typ `Int` als auch `Float` umfasst, und den Type `Char`, der die Teilmenge der aus exakt einem Zeichen bestehenden Zeichenketten bezeichnet. (Technisch gesehen ist `Num` der „Supertyp" von `Int` und `Float`, während `Char` ein „Subtyp" von `String` ist. Es ist in Q möglich, Datentypen aus anderen Datentypen „abzuleiten", was dem Klassen-System mit „einfacher Vererbung" in objektorientierten Programmiersprachen wie Smalltalk entspricht. Weitere Hinweise dazu findet man im Q-Handbuch.) Schließlich gibt es auch noch den Typ `Bool`, der die Wahrheitswerte `true` und `false` umfasst.

Typ-Symbole können auf der linken Seite einer Gleichung oder einer Variablen-Definition in einem Skript als „Typ-Wächter" verwendet werden, um den Typ einer Variablen einzuschränken. Die Definition ist dann nur anwendbar, wenn der tatsächliche Wert der Variable dem angegebenen Typ entspricht. Um z.B. sicherzustellen, dass die `sqr`-Funktion aus Abschnitt 3.6 stets nur auf Fließkommazahlen angewendet wird, können wir die linke Seite mit einem Typ-Wächter ausstatten:

```
sqr X:Float          = X*X;
```

Wollen wir die Funktion auch auf ganze Zahlen anwenden, so können wir stattdessen den übergeordneten Datentyp `Num` verwenden:

```
sqr X:Num           = X*X;
```

Benutzer-definierte Datentypen

Q gestattet auch die Definition neuer Datentypen, wozu Typ-Deklarationen verwendet werden (vgl. Abschnitt 3.3). Die Werte solcher Datentypen können entweder Funktions-Symbole sein oder die Anwendung eines Funktions-Symbols auf ein oder oder mehrere Argumente. Man spricht daher auch von „algebraischen“ Datentypen. Die Funktions-Symbole werden normalerweise als `const` vereinbart, da es sich bei ihnen um „Konstruktoren“ handelt, mit denen konstante Werte konstruiert werden sollen.

Um z.B. einen Datentyp zu vereinbaren, dessen Elemente dazu geeignet sind, eine binäre Baum-Datenstruktur zu repräsentieren (solche Datenobjekte spielen z.B. beim Suchen und Sortieren eine Rolle), verwendet man eine Deklaration wie die folgende:

```
type BinTree = const nil, bin X T1 T2;
```

Hier werden mit dem Datentyp `BinTree` zwei entsprechende Konstruktor-Symbole vereinbart: das Symbol `nil` zur Darstellung eines „leeren“ Baums (dieses Symbol erwartet keine Argumente, und wird daher auch als *Konstanten-Symbol* bezeichnet) und der Konstruktor `bin`, mit dem ein Baum mit der Information `X` und den beiden „Teilbäumen“ `T1` und `T2` repräsentiert wird. Ein Datenobjekt des Typs `BinTree` hat z.B. folgende Gestalt:

```
bin 5 (bin 3 nil nil) (bin 7 nil (bin 9 nil nil))
```

Der neue Datentyp kann dann wie eines der vordefinierten Typ-Symbole als Wächter auf der linken Seite einer Gleichung eingesetzt werden.

Ein Hinweis zu den Typ-Symbolen: Typ-Bezeichner bilden eine spezielle Symbol-Klasse, können also nicht mit Funktions- und Variablen-Symbolen „kollidieren“. Üblicherweise beginnen Typ-Symbole mit einem Großbuchstaben, dies ist aber nicht zwingend vorgeschrieben. Die Verwendung „qualifizierter“ Bezeichner zur Auflösung von Mehrdeutigkeiten bei gleichnamigen Typen in verschiedenen Skripts wird wie bei Funktions- und Variablen-Symbolen gehandhabt (vgl. Abschnitt 3.4).

Eine spezielle Form der Typ-Deklaration dient dazu, einen so genannten „Aufzählungs-Typ“ zu definieren, dessen Konstruktoren alle Konstanten-Symbole sind. Z.B.:

```
type Day = const sun, mon, tue, wed, thu, fri, sat;
```

Aufzählungs-Typen werden in Q speziell unterstützt. Die Werte eines Aufzählungs-Typs können miteinander verglichen werden, wobei die Elemente in der Reihenfolge angeordnet werden, in der sie in der Typ-Deklaration aufgeführt sind, also im vorstehenden Beispiel `sun < mon`, `mon < tue`, `tue < wed`, etc. Außerdem liefert die eingebaute `ord`-Funktion angewendet auf ein Element des Aufzählungs-Typs die Ordnungszahl der Konstante (`ord sun = 0`, `ord mon =`

`1`, usw.) und mit den Funktionen `succ` und `pred` kann der Nachfolger und der Vorgänger eines Elements bestimmt werden: `succ sun = mon`, `pred mon = sun`.

Ein Beispiel eines eingebauten Aufzählungs-Typs ist der Typ `Bool`, den man sich wie folgt deklariert denken kann:

```
public type Bool = const false, true;
```

Auch der eingebaute `Char`-Datentyp wird als Aufzählungstyp behandelt; dabei sind die einzelnen Zeichen entsprechend dem ASCII-Code angeordnet.

Die Q-Standard-Bibliothek enthält übrigens ein Sortiment verschiedener nützlicher Datentypen zusammen mit den entsprechenden Operationen; siehe den Abschnitt „Standard Types“ in Kapitel 11 des Q-Handbuchs.

3.10 Parallel-Verarbeitung

Verschiedene Typen von MIDI-Programmen, so z.B. die gleichzeitige Wiedergabe und Aufzeichnung von MIDI-Sequenzen, lassen sich am einfachsten unter Einsatz der Parallel-Verarbeitung (so genanntes „Multithreading“) realisieren. Dabei wertet der Q-Interpreter gleichzeitig mehrere Ausdrücke aus. Dies wird ermöglicht durch die Standardbibliotheks-Funktion `thread`, die als Argument einen auszuwertenden Ausdruck erwartet. Dieser wird dann sozusagen „im Hintergrund“ verarbeitet, während die `thread`-Funktion unmittelbar ein „Thread“-Objekt zurückliefert, über das zu gegebener Zeit das Ergebnis der Auswertung abgefragt werden kann.

Exkurs: Prozesse und Threads

Frühe Computer-Systeme konnten jeweils nur ein Programm ausführen. Mit der Einführung der sogenannten Timesharing-Systeme wurde es aber notwendig, die Eingaben einer großen Zahl verschiedener Benutzer gleichzeitig zu verarbeiten. Dazu wurden so genannte Mehr-Benutzer und Mehr-Prozess-Betriebssysteme entwickelt. Auch die heutigen PC-Betriebssysteme sind in der Lage, eine Vielzahl von Programmen gleichzeitig auszuführen.

Ein im Computer ablaufendes Programm wird auch als *Prozess* bezeichnet. Ein Mehr-Prozess-Betriebssystem verteilt die auszuführenden Befehle verschiedener Prozesse auf die zur Verfügung stehenden Prozessoren (CPUs = Central Processing Units). Ist, wie in den meisten heutigen PCs, nur eine CPU vorhanden, so werden jeweils immer nur ein paar Befehle eines Prozesses auf der CPU ausgeführt, danach kommt der nächste Prozess an die Reihe, usw. Auf diese Weise können auch auf einer einzigen CPU mehrere Prozesse gleichzeitig ausgeführt werden. (In Wirklichkeit wird natürlich zu jedem Zeitpunkt immer nur *ein* Befehl *eines* Prozesses von der CPU bearbeitet. Da die verschiedenen Prozesse sich aber in sehr schneller Folge abwechseln, entsteht dabei die Illusion einer gleichzeitigen Ausführung.)

Heutige Betriebssysteme gestatten aber nicht nur die gleichzeitige Ausführung verschiedener Prozesse, sondern auch die weitere Aufteilung von Prozessen in parallele Ausführungs-„Fäden“, die *Threads* genannt werden. Bei einem Thread handelt es sich also sozusagen um einen „Prozess innerhalb eines Prozesses“. Im Unterschied zu verschiedenen Prozessen, deren Daten fein säuberlich voneinander getrennt sind, können verschiedene Threads innerhalb eines Prozesses gemeinsam auf die Daten des Programms zugreifen. Jeder Prozess verfügt stets über mindestens einen Thread, den so genannten „Main Thread“, kann aber daneben auch weitere Threads erzeugen, in denen verschiedene Programmteile gleichzeitig abgearbeitet werden. Dann spricht man von *Multithreading*.

Damit man später auf das Ergebnis eines Threads zugreifen kann, muss das von `thread` gelieferte Thread-Objekt z.B. in einer Variablen zwischengespeichert werden. Beispiel:

```
==> def TH = thread (sum (nums 1 1000000))
```

Die Berechnung der Summe findet nun im Hintergrund statt, man kann währenddessen also weitere Ausdrücke berechnen:

```
==> 2*17/9
3.77777777777778

==> writes "Hello!\n"
Hello!
()
```

Will man schließlich das Resultat des Threads abfragen, so wendet man dazu die Standardbibliotheks-Funktion `result` auf das Thread-Objekt an. Die `result`-Funktion wartet gegebenenfalls, bis die Berechnung abgeschlossen ist und liefert dann das Ergebnis:

```
==> result TH
500000500000
```

Man kann die `thread`-Funktion aber auch verwenden, um mehrere Berechnungen simultan auszuführen. Zum Beispiel startet die folgende `main`-Funktion zwei Threads, die gleichzeitig in zufälligen Zeitintervallen mit der `printf`-Funktion Ausgaben auf dem Terminal erzeugen:

```
task N          = sleep_some || printf "task #%d\n" N || task N;
sleep_some      = sleep (random/0x100000000);
main            = (thread (task 1), thread (task 2));
```

Beispiel:

```
==> def (TH1,TH2) = main

==> task #1
task #2
task #1
task #1
task #2
...
```

Teil II: MIDI-Programmierung mit Q

4 Grundlegendes

Nachdem wir uns nun mit den Grundlagen der Programmierprache Q vertraut gemacht haben, zeigen wir im folgenden, wie die Q-Midi-Schnittstelle zur Programmierung einfacher MIDI-Anwendungen in Q eingesetzt wird. Auch hier können wir nur auf die wichtigsten Funktionen eingehen. Weiterführende Informationen findet man in der `etc/README-Midi`-Datei im Q-Verzeichnis (`/usr/share/q` für Linux bzw. `/Programme/Qpad` für Windows), und im Q-Midi-Skript `lib/midi.q`.

Zunächst müssen wir uns damit vertraut machen, wie man in einem Q-Skript auf die Q-Midi-Funktionen zugreift. Da diese Funktionen nicht zur Standard-Bibliothek gehören, müssen sie explizit importiert werden. Dies geschieht durch eine `import`-Deklaration am Beginn des Q-Skripts:

```
import midi, mididev;
```

Das Skript `midi.q` enthält die eigentlichen Q-Midi-Funktionen. Wir importieren außerdem das Skript `mididev.q`, das einige weitere Variablen und Funktionen zum portablen Zugriff auf die MIDI-Ein-/Ausgabegeräte bereitstellt. Diese werden im folgenden noch näher erläutert.

4.1 Registrieren eines MidiShare-Clients

Bevor man mit den Q-Midi-Funktionen MIDI-Ereignisse einlesen oder ausgeben kann, muss zunächst ein „Client" bei MidiShare registriert werden. Innerhalb eines Skripts können auch mehrere Clients angemeldet und verwendet werden. Beipielsweise könnte ein Sequencer-Programm mit „Record"-Funktion so aufgebaut sein, dass es zwei separate Clients, einen für die Wiedergabe und einen für die Aufzeichnung, umfasst. Zur Registrierung eines Clients wird die Funktion `midi_open` verwendet, wobei der gewünschte Name des Clients zu übergeben ist. Bei dem Namen kann es sich um eine beliebige Zeichenkette handeln, man sollte allerdings vermeiden, dass der gleiche Name mehrfach verwendet wird. Die Funktion liefert die „Referenznummer" des Clients zurück, auf die dann in allen den Client betreffenden MIDI-Ein- und Ausgabe-Operationen Bezug genommen wird. Die Referenznummer muss also für die spätere Verwendung gespeichert werden. Am einfachsten erreicht man dies, indem man das Ergebnis der `midi_open`-Funktion einer globalen Variablen zuweist, z.B.:

```
==> def REF = midi_open "Beispiel"
```

Sobald ein Client registriert wurde, erscheint seine Referenznummer in der Liste aller registrierten MidiShare-Clients, die man mit der Funktion `midi_clients` abfragen kann:

```
==> midi_clients
[0,1]
```

Den Namen eines durch seine Referenznummer gegebenen MidiShare-Clients erhält man mit der Funktion `midi_client_name`. Z.B. kann man die Liste der Namen aller MidiShare-Clients durch Anwendung der `midi_client_name`-Funktion auf die Elemente der `midi_clients`-Liste abfragen:

```
==> map midi_client_name midi_clients
["MidiShare","Beispiel"]
```

Mit der `midi_client_set_name`-Funktion kann man den Namen eines registrierten Clients ändern:

```
==> midi_client_set_name 1 "Bsp"
()

==> map midi_client_name midi_clients
["MidiShare","Bsp"]
```

Die `midi_client_ref`-Funktion liefert die Referenznummer für einen durch seinen Namen gegebenen Client:

```
==> midi_client_ref "Bsp"
1
```

Wird ein Client nicht weiter benötigt, so kann er bei MidiShare mit der Funktion `midi_close` abgemeldet werden:

```
==> midi_close 1
()

==> midi_clients
[]
```

Man beachte, dass der Client mit der Referenznummer 0 und dem Namen `"MidiShare"` stets durch die MidiShare-Bibliothek vordefiniert wird, sobald MidiShare aktiv ist, d.h. wenn mindestens ein Anwendungs-Client definiert wurde. Dieser Client dient unter Windows zur Adressierung der physikalischen MIDI-Ein-/Ausgabegeräte; in der Linux-Version von MidiShare hat er zur Zeit keine besondere Funktion, da die MIDI-Ein-/Ausgabe über andere, spezielle „Treiber"-Clients erfolgt (s.u.).

Ein weiterer Unterschied zwischen der Windows- und der Linux-Version von MidiShare besteht darin, welche Clients jeweils in einer Q-Midi-Anwendung sichtbar sind. Unter Linux enthält die `midi_clients`-Liste *alle* registrierten MidiShare-Clients, auch die Clients anderer Programme. Dies ist praktisch, wenn man z.B. die Ausgabe eines Programms direkt mit der Eingabe eines anderen Programms verknüpfen möchte. Unter Windows ist dies derzeit nicht möglich, da stets nur die innerhalb der jeweiligen Anwendung registrierten MidiShare-Clients sichtbar sind.

4.2 Clients für die MIDI-Ein- und Ausgabe

Auch zum Zugriff auf die physikalischen MIDI-Ein- und Ausgabegeräte werden entsprechende MidiShare-Clients verwendet. Diese werden normalerweise vor der Ausführung des Skripts automatisch geladen, müssen also *nicht* in der oben beschriebenen Weise explizit registriert werden. Um die Referenznummer eines solchen Clients zu erhalten, kann die oben bereits erwähnte Funktion `midi_client_ref` benutzt werden. Allerdings bestehen bei dem Zugriff auf MIDI-Ein-/Ausgabegeräte mittels MidiShare große Unterschiede zwischen Linux und Windows. Während nämlich unter Windows die MIDI-Ein- und Ausgabe stets über den MidiShare-Client mit der Referenznummer 0 erfolgt, sind dafür in der Linux-Version (zur Zeit, d.h. für MidiShare 1.86) verschiedene andere, so genannte „Treiber"-Clients zuständig, die separat gestartet werden müssen. Wir verwenden daher im folgenden stattdessen die portablen MIDI-Geräte-Definitionen in `mididev.q`. Diese erlauben einen vom jeweiligen Betriebssystem unabhängigen Zugriff auf die vorhandenen MIDI-Geräte.

Zu diesem Zweck definiert das `mididev`-Skript eine Gerätetabelle in Form einer Liste, auf die über die globale Variable `MIDIDEV` zugegriffen werden kann. Diese Tabelle muss ggf. an die vorhandene Systemkonfiguration angepasst werden. Im weiteren gehen wir von der folgenden Standard-Belegung der Einträge der `MIDIDEV`-Tabelle aus:

- `MIDIDEV!0` repräsentiert die externe MIDI-Schnittstelle (z.B. ein angeschlossener MIDI-Synthesizer), die sowohl für die Eingabe als auch für die Ausgabe verwendet werden kann.
- `MIDIDEV!1` repräsentiert den internen Synthesizer, der eine direkte MIDI-Ausgabe über die Soundkarte ermöglicht. Dieses Gerät kann normalerweise nur für die Ausgabe verwendet werden.
- `MIDIDEV!2` repräsentiert das Netzwerk (z.B. ein lokales Ethernet, über das verschiedene Rechner miteinander vernetzt sind). Auf diesem Gerät ist sowohl Ein- als auch Ausgabe möglich.

Jeder Eintrag der `MIDIDEV`-Tabelle ist ein Tripel `(NAME,REF,PORT)` mit den folgenden Informationen:

- `NAME`: Der Name des Geräts, eine im Prinzip frei wählbare Zeichenkette. Diese dient nur zu Informationszwecken, z.B. wenn man eine Liste der zur Verfügung stehenden Geräte anzeigen möchte.
- `REF`: Die Referenznummer des Clients, über den das Gerät angesprochen wird. Unter Windows ist dies stets der Wert 0 (der Standard-MidiShare-Client), während unter Linux hier die aktuellen Referenznummern der geladenen Treiber-Clients erscheinen. Das `mididev`-Skript sorgt auch dafür, dass die Treiber-Clients bei Bedarf automatisch gestartet werden.
- `PORT`: Die „Portnummer" des Geräts. Unter Linux ist diese bedeutungslos, da die Adressierung von MIDI-Ereignissen allein über die Referenznummern der Treiber-Clients vorgenommen wird. Unter Windows findet man hier die logische MidiShare-Portnummer, die mit dem `msDrivers`-Programm eingestellt wurde. Die Portnummern werden unter Windows verwendet, um ein auszugebendes MIDI-Ereignis an ein bestimmtes MIDI-Gerät zu adressieren.

Um ein Eingabegerät in einer Q-Midi-Anwendung verwenden zu können, benötigt man normalerweise nur die Client-Nummer. Für die portable Adressierung eines Ausgabegeräts wird dagegen sowohl die Client- als auch die Portnummer verwendet. Diese Werte kann man aus der `MIDIDEV`-Tabelle abrufen und entsprechenden globalen Variablen zuweisen. Ein Q-Midi-Skript beginnt daher oft mit einer Zeile wie der folgenden, in der neben der Abfrage der Gerätenummern auch die bereits oben diskutierte Registrierung eines eigenen Clients erfolgt. (Man vergleiche auch unser erstes Q-Midi-Beispiel, `bsp01.q`, in Abschnitt 2.3.)

```
def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, REF = midi_open "Bsp";
```

4.3 Herstellen von Verbindungen zwischen Clients

Die zum Zugriff auf die MIDI-Ein-/Ausgabe notwendigen Vorbereitungen sind nun fast abgeschlossen. Zum Schluss müssen nur noch Verbindungen zwischen dem registrierten MidiShare-Client und den Clients für die Ein- und Ausgabe geschaltet werden. Dies erreicht man mit der Funktion `midi_connect`, die als Argumente die Referenznummern eines *Ursprungs-* und eines *Ziel*-Clients erhält. Im folgenden setzen wir Definitionen der Variablen `IN`, `OUT` und `REF` wie im vorangegangenen Abschnitt voraus (die `PORT`-Variable wird hier noch nicht benötigt, sondern erst bei der MIDI-Ausgabe; s. folgenden Abschnitt). Die Verbindungen zwischen unserem Client `REF` und der MIDI-Ein- und Ausgabe stellen wir dann wie folgt her:

```
==> midi_connect IN REF || midi_connect REF OUT
```

Natürlich können wir die Verbindungen auch innerhalb des Skripts einrichten. Wir verwenden hier eine Variablendefinition, um die notwendigen Initialisierungen beim Starten des Skripts durchzuführen, z.B.:

```
def _ = midi_connect IN REF || midi_connect REF OUT;
```

Man beachte, dass diese Zeile nach der Definition der Variablen `IN`, `OUT` und `REF` stehen muss. Da wir das Ergebnis der `midi_connect`-Aufrufe nicht benötigen, haben wir auf der linken Seite der Definition die anonyme Variable verwendet.

Jedes Q-Midi-Skript, mit dem von der MIDI-Eingabe gelesen und auf die MIDI-Ausgabe geschrieben werden soll, benötigt mindestens die oben skizzierten Verbindungen. Verbindungen können allgemein aber auch zwischen beliebigen Clients hergestellt werden. Enthält ein Skript mehrere Clients, so kann man diese also auf beliebige Weise untereinander verknüpfen. Unter Linux können auch Verbindungen zu Clients in anderen Q-Midi-Anwendungen hergestellt werden. Dies kann entweder innerhalb des Q-Midi-Programms mittels `midi_connect` geschehen, oder auch mit dem externen `msconnect`-Programm, das zusammen mit der MidiShare-Bibliothek installiert wird. Das `msconnect`-Programm ist auch nützlich, um die aktuell registrierten MidiShare-Clients und deren Verbindungen zu überprüfen, wenn eine Q-Midi-Anwendung gestestet werden soll.

Innerhalb eines Q-Midi-Skripts kann mittels `midi_disconnect` eine bestehende Verbindung auch wieder gelöst werden. Darüberhinaus lässt sich der Status einer Verbindung mit dem Prädikat `midi_connected` überprüfen, und man erhält die Liste aller Client-Nummern der eingehenden und ausgehenden Verbindungen eines Clients mit `midi_in_connections` und `midi_out_connections`:

```
==> midi_clients
[0,1,2,3,4,5]

==> map midi_client_name _
["MidiShare","/dev/midi","iiwusynth","msWANDriver","localhost","Bsp"]
```

```
==> midi_connected REF OUT
true

==> midi_in_connections REF; midi_out_connections REF
[1]
[1]

==> midi_disconnect REF OUT
()

==> midi_connected REF OUT
false
```

4.4 MIDI-Ein- und Ausgabe

Sobald ein MidiShare-Client registriert und die Verbindungen zu den MIDI-Ein- und Ausgabegeräten geschaltet wurden, können MIDI-Nachrichten mit der Funktion `midi_get` von der MIDI-Eingabe gelesen und mit der `midi_send`-Funktion an die MIDI-Ausgabe gesendet werden. Um dies zu testen, erstellen wir zunächst ein einfaches Skript, das die in den vorangegangenen Abschnitten besprochenen Initialisierungen vornimmt:

```
/* bsp08.q: Demonstration MIDI-Ein- und Ausgabe */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, REF = midi_open "bsp08",
  _ = midi_connect IN REF || midi_connect REF OUT;
```

Nachdem das Skript gestartet wurde, ist also ein MidiShare-Client namens `"bsp08"` registriert und mit der MIDI-Ein- und Ausgabe verknüpft. Über diesen Client laufen nun die weiteren Operationen.

MIDI-Eingabe

Betrachten wir zunächst die MIDI-Eingabe. Die `midi_get`-Funktion wird mit einem einzigen Argument aufgerufen, der Referenznummer des Clients. Wir können also einzelne MIDI-Ereignisse vom angeschlossenen Keyboard wie folgt einlesen:

```
==> midi_get REF
(1,0,3068410,note_on 0 60 117)

==> midi_get REF
(1,0,3068670,note_on 0 60 0)
```

Die `midi_get`-Funktion liefert für jedes MIDI-Ereignis ein Tupel `(REF,PORT,TIME,MSG)` mit den folgenden Angaben:

- `REF`: Die Referenznummer des Clients, von dem das Ereignis empfangen wurde. Im vorliegenden Beispiel ist dies die Nummer des Treiber-Clients, also identisch mit dem Wert der `IN`-Variable.

- `PORT`: Die Portnummer des Ereignisses. Unter Linux ist diese stets 0. Unter Windows wird hier die Nummer des MidiShare-Ports geliefert, so dass man unterscheiden kann, von welchem Gerät das Ereignis erzeugt wurde.
- `TIME`: Die „MidiShare-Zeit" des Ereignisses in Millisekunden. Wie wir später noch genauer sehen werden, hat MidiShare einen internen Zähler, in dem die aktuelle Zeit seit der letzten Aktivierung von MidiShare gespeichert wird. Damit kann MidiShare auf Millisekunden genau die Zeit bestimmen, zu der eine Nachricht von der MIDI-Schnittstelle empfangen wurde. Der von `midi_get` gelieferte „Zeitstempel" sagt uns also, wann genau das empfangene MIDI-Ereignis stattfand.
- `MSG`: Die empfangene MIDI-Nachricht. Diese wird, wie wir in Kapitel 5 noch genauer diskutieren, als ein Element des Q-Datentyps `MidiMsg` dargestellt. Z.B. wird eine Note-On-Nachricht als ein Ausdruck der Form `note_on C P V` kodiert, wobei `C` der MIDI-Kanal („Channel"), `P` die Tonhöhe („Pitch") und `V` die Anschlagstärke („Velocity") ist.

Im vorliegenden Beispiel wurde also auf MIDI-Kanal 0 das Mittel-C (MIDI-Note Nummer 60) mit der Dynamik 117 angeschlagen und nach 3068670-3068410=260 Millisekunden wieder losgelassen.

Die `midi_get`-Funktion liefert eingehende MIDI-Ereignisse in dem Moment, in dem sie empfangen werden, so dass die Verarbeitung der Ereignisse in „Echtzeit" möglich ist.[2] Damit Nachrichten nicht verlorengehen, während das Programm mit anderen Dingen beschäftigt ist, werden eingehende Nachrichten, die nicht sofort abgerufen werden, in einem *Eingabepuffer* abgelegt, und zwar in der Reihenfolge, in der sie eingehen. Die Nachrichten können dann zu einem späteren Zeitpunkt in der Reihenfolge ihres Eintreffens abgerufen werden. Der Eingabepuffer arbeitet also nach dem Prinzip einer „Warteschlange", weswegen man ihn auch als „Eingabe-Schlange" („input queue") bezeichnet. Jeder registrierte Client verfügt über seinen eigenen Eingabepuffer, in dem alle Nachrichten abgelegt werden, die ihn über die mit `midi_connect` geschalteten eingehenden Verbindungen erreichen. Manchmal ist es notwendig, diesen Eingabepuffer zu löschen. Dazu verwendet man die Funktion `midi_flush`:

```
==> midi_flush REF
()
```

Nach Anwendung dieser Operation ist der Eingabepuffer leer; ein darauf folgendes `midi_get` wird also darauf warten, dass neue Ereignisse in den Puffer eingespeist werden.

MIDI-Ausgabe

Kommen wir nun zur MIDI-Ausgabe. Diese erfolgt über die Funktion `midi_send`, die mit den folgenden Parametern aufgerufen wird:

- `REF`: Die Referenznummer des Clients, über den die Nachricht gesendet wird.
- `PORT`: Der Ziel-Port der Nachricht. Unter Linux kann dieser stets auf 0 gesetzt werden; unter Windows wird mit dieser Nummer das Ausgabegerät adressiert. In einem portablen Programm,

2 Der Terminus „Echtzeit" ist hier „cum grano salis" zu verstehen. Weder Windows noch Linux sind wirkliche Echtzeit-Systeme, die *immer* innerhalb vorgegebener Zeitgrenzen arbeiten. Tatsächlich erfolgt also die Verarbeitung nicht wirklich „sofort", sondern „sobald wie möglich" innerhalb der Grenzen der Hardware und des Betriebssystems unter Berücksichtigung z.B. der momentanen Prozessorlast. Bei heutigen PC-Systemen, deren Prozessortakt mindestens einige hundert MHz beträgt, ist dies aber für den Zweck der MIDI-Programmierung normalerweise völlig ausreichend.

das sowohl unter Linux als auch Windows laufen soll, verwendet man hier den aus der `MIDIDEV`-Tabelle ermittelte Wert für die Portnummer (vgl. `bsp08.q`).

- `MSG`: Die zu übertragende Nachricht, als Element des `MidiMsg`-Datentyps kodiert.

Will man also z.B. ein Mittel-C (Note Nummer 60 auf Kanal 0 mit maximaler Lautstärke) auf dem angeschlossenen Synthesizer ausgeben und danach wieder abschalten, so wird `midi_send` wie folgt aufgerufen:

```
==> midi_send REF PORT (note_on 0 60 127)
()

==> midi_send REF PORT (note_on 0 60 0)
()
```

Man beachte, dass in diesem Fall die Ausgabe der MIDI-Nachrichten sofort, also in „Echtzeit" erfolgt. Man kann stattdessen auch einen Zeitpunkt angeben, zu dem die Ausgabe erfolgen soll. Dazu wird als drittes Argument von `midi_send` ein Paar `(T,MSG)` verwendet, wobei `T` der gewünschte Zeitpunkt und `MSG` wieder die auszugebende MIDI-Nachricht ist. Zur Errechnung des Zeitpunkts ist die Funktion `midi_time` nützlich, die den momentanen Zeitwert liefert. Um z.B. das Mittel-C sofort anzuschlagen und nach einer halben Sekunde (= 500 Millisekunden) wieder loszulassen, verwendet man `midi_send` wie folgt:

```
==> midi_send REF PORT (midi_time,note_on 0 60 127) || midi_send REF PORT ↵
(midi_time+500,note_on 0 60 0)
()
```

Tip: Zur Kodierung von Noten-Ereignissen gibt es auch eine besondere MidiShare-spezifische Erweiterung, die `note`-Nachricht, bei der man die Dauer der Note direkt angeben kann. Hierbei handelt es sich nicht um eine „echte" MIDI-Nachricht; `note`-Nachrichten werden von MidiShare bei der Ausgabe automatisch in zwei separate MIDI-Ereignisse (ein „Note-On" gefolgt von einem „Note-Off" nach der angegebenen Dauer) umgesetzt. Man kann also das Ergebnis des letzten Beispiels (Mittel-C für eine halbe Sekunde) auch einfacher wie folgt erreichen:

```
==> midi_send REF PORT (note 0 60 127 500)
()
```

Die `note`-Nachrichten erleichtern die direkte Transkription einer Partitur nach MIDI. Beim Einlesen von MIDI-Nachrichten von einem MIDI-Gerät werden aber stets „echte" `note_on`- und `note_off`-Nachrichten geliefert, hier findet also keine automatische Umsetzung statt.

4.5 Filterfunktionen

Einige MIDI-Keyboards erzeugen von sich aus MIDI-Ereignisse in regelmäßigen Abständen, z.B. `active_sense`- oder `clock`-Nachrichten, die für spezielle Anwendungen, insbesondere für die Synchronisierung mehrerer MIDI-Geräte, nützlich sind. Oft werden diese Nachrichten aber gar nicht benötigt, oder sind sogar lästig, etwa wenn man interaktiv MIDI-Ereignisse mit der `midi_get`-Funktion abfragen möchte. Leider erlauben viele Keyboards nicht, die automatische Erzeugung dieser Nachrichten abzuschalten. Die Q-Midi-Schnittstelle stellt daher die Funktion

`midi_accept_type` bereit, mit der man bestimmte Typen von Nachrichten ganz aus der MIDI-Eingabe herausfiltern kann. Diese Funktion wird mit drei Argumenten aufgerufen, der Nummer des Client, dem Konstruktor-Symbol der Nachricht und einem Wahrheitswert (`true`: Nachricht wird *nicht* gefiltert; `false`: Nachricht wird gefiltert). Um z.B. alle `active_sense`-Nachrichten herauszufiltern, geht man wie folgt vor:

```
==> midi_get REF                // Gerät liefert active_sense-Nachrichten
(1,0,754870,active_sense)

==> midi_get REF
(1,0,755110,active_sense)

==> midi_accept_type REF active_sense false   // Aktivieren des Filters
()

==> midi_flush REF              // Leeren des Eingabepuffers
()

==> midi_get REF                // active_sense wird nun gefiltert
(1,0,780740,note_on 0 53 115)

==> midi_get REF
(1,0,780870,note_on 0 53 0)
```

Neben `midi_accept_type` gibt es auch noch weitere Filterfunktionen, mit denen MIDI-Ereignisse eines bestimmten MidiShare-Ports oder eines vorgegebenen MIDI-Kanals gefiltert werden können. Man kann auch sämtliche Filterfunktionen auf einen Schlag mit der Funktion `midi_accept_all` ausschalten. Weitere Informationen dazu finden sich in der Q-Midi-Dokumentation.

5 MIDI-Nachrichten und -Ereignisse

Wie bereits erwähnt wurde, werden MIDI-Nachrichten von Q-Midi nicht direkt als Byte-Sequenzen kodiert, sondern als Elemente eines speziellen Datentyps MidiMsg. Dies erleichtert die Interpretation und Bearbeitung von MIDI-Nachrichten in einem Programm. Z.B. wird eine „Note On"-Nachricht durch einen konstanten Ausdruck der Form note_on CHAN PITCH VEL dargestellt. Die Deklaration des MidiMsg-Datentyps findet sich ziehmlich am Beginn des midi.q-Skripts. Im folgenden gehen wir kurz auf die wichtigsten Typen von MIDI-Nachrichten und deren Darstellung als MidiMsg-Datenelemente ein.

5.1 Nachrichten-Kategorien

Wie wir bereits in Abschnitt 1.3 gesehen hatten, unterscheidet man bei MIDI verschiedene Kategorien von Nachrichten. Diese Unterteilung der Nachrichten wird auch in der Q-Midi-Schnittstelle beachtet. Q-Midi unterstützt sämtliche gängigen, vom MIDI-Standard vorgesehenen Nachrichten-Kategorien:

- *Voice-Nachrichten:* Hierbei handelt es sich um die Nachrichten, die für einen bestimmten MIDI-Kanal bestimmt sind. Dazu zählen z.B. note_on und note_off, Controller- (ctrl_change) und Instrument-Nachrichten (prog_change).
- *System-Common-Nachrichten:* Hierzu zählen alle System-Nachrichten, die *nicht* in Echtzeit verarbeitet werden, d.h., deren Verarbeitung eine gewisse Zeit in Anspruch nehmen kann. Das gebräuchlichste Beispiel sind die sysex-Nachrichten, mit denen Hardware-spezifische Steuerfunktionen aufgerufen werden. Außerdem findet man in dieser Kategorie Nachrichten wie quarter_frame und song_pos, die zur Synchronisierung der MIDI-Wiedergabe mit einer externen Sound-Quelle dienen, und andere Nachrichten wie tune zur Steuerung des angeschlossenen Synthesizers.
- *System-Realtime-Nachrichten:* Im Unterschied zur System-Common-Kategorie werden diese Nachrichten in Echtzeit, also sofort, von MIDI-Geräten verarbeitet. Realtime-Nachrichten sind ausnahmslos sehr kurz; sie dienen zur Synchronisierung der laufenden MIDI-Wiedergabe (active_sense und clock) und zur Kontrolle der MIDI-Wiedergabe (start, stop, continue,reset).
- *Meta-Nachrichten:* Diese Nachrichten findet man nur in MIDI-Dateien. Sie dienen dort zur Speicherung bestimmter Meta-Informationen wie z.B. Tonart- und Metrum-Angaben, Spur-Bezeichnungen, Liedtexte, Marker usw. Mit diesen Nachrichten werden wir uns erst in Kapitel 8 beschäftigen.

Neben den oben angeführten Nachrichten-Kategorien unterstützt Q-Midi auch noch weitere, MidiShare-spezifische Nachrichten-Typen. Dazu gehören die bereits erwähnte note-Nachricht, verschiedene Nachrichten zur Steuerung spezieller Controller-Funktionen, und die midi_stream-Nachricht, mit der man eine beliebige „rohe" Byte-Folge an ein MIDI-Gerät senden kann. Alle diese Nachrichten werden bei der Ausgabe von MidiShare automatisch in Standard-MIDI-Nachrichten umgesetzt. Im folgenden behandeln wir (mit Ausnahme der note-Nachricht) nur die Standard-MIDI-Nachrichten.

5.2 Noten

Die wichtigsten MIDI-Nachrichten sind naturgemäß jene, mit denen Noten kodiert werden, also die Standard-MIDI-Nachrichten note_on und note_off. Daneben gibt es noch die MidiShare-spezifische note-Nachricht, mit der eine Note mit einer bestimmten Dauer ausgegeben werden kann.

`note_on CHAN PITCH VEL`	„Note-On“ auf Kanal `CHAN` (0-15) mit Tonhöhe `PITCH` (0-127) und Dynamik `VEL` (0-127)
`note_off CHAN PITCH VEL`	„Note-Off“(Parameter wie bei `note_on`)
`note CHAN PITCH VEL DUR`	Note mit `CHAN`, `PITCH`, `VEL` wie bei `note_on`, `DUR` = Dauer der Note (in Millisekunden)

5.3 Instrumentierung und Controller-Nachrichten

Zur Festlegung des Instrumenten-Klangs und zur Steuerung verschiedener Controller-Funktionen werden die Voice-Nachrichten `prog_change` und `ctrl_change` verwendet.

`prog_change CHAN PROG`	„Program Change“ auf Kanal `CHAN` (0-15) für Instrument Nummer `PROG` (0-127)
`ctrl_change CHAN CTRL VAL`	„Control Change“ auf Kanal `CHAN` (0-15) für Controller Nummer `CTRL` (0-127), Wert `VAL` (0-127)

Eine Aufstellung aller Standard-Controller-Nummern findet man auf Jeff Glatts Website [http://www.borg.com/~jglatt/]. Einige Controller haben zwei verschiedene Controller-Nummern für die Grob- und Feineinstellung, so z.B. 0 und 32 („Bank Select Coarse“ und „Fine“, mit denen man auf vielen neueren Synthesizern verschiedene Sätze mit jeweils 128 Instumenten-Klängen ansprechen kann), 1 und 33 („Modulation Wheel Coarse/Fine“, mit denen normalerweise die Stärke eines Vibrato-Effekts gesteuert wird), und 7 und 39 („Volume Coarse/Fine“). Weitere wichtige Controller sind 91 und 93, mit denen man auf vielen Geräten die Stärke des Reverb- und Chorus-Effekts einstellen kann. Darüberhinaus gibt es noch einige spezielle Controller-Nummern (98-101 und 38), mit denen man bis zu 32768 weitere Parameter, die so genannten „RPNs“ („Registered Parameter Numbers“) und „NRPNs“ („Non-Registered Parameter Numbers“) setzen kann. Die Bedeutung der NRPNs ist Geräte-abhängig, während die Funktion der RPNs von der MMA einheitlich festgelegt sind. Zu den wichtigsten RPNs zählen z.B. die Parameter „Pitch Bend Range“ (Einstellung der Sensitivität des Tonhöhenrades) sowie „Master Coarse“ und „Master Fine Tuning“, mit denen die Grundstimmung des Synthesizers geändert werden kann.

5.4 Weitere Voice-Nachrichten

Mit den folgenden Nachrichten lassen sich weitere Kanal-spezifische Parameter einstellen.

`key_press CHAN PITCH VAL`	„Key Pressure“ auf Kanal `CHAN` (0-15) für Tonhöhe `PITCH` (0-127), Wert `VAL` (0-127)
`chan_press CHAN VAL`	„Channel Pressure“ auf Kanal `CHAN` (0-15), Wert `VAL` (0-127)
`pitch_wheel CHAN LSB MSB`	„Pitch Wheel“ auf Kanal `CHAN` (0-15), Wert `LSB` und `MSB` (jeweils 0-127)

Mit der `pitch_wheel`-Nachricht wird die Tonhöhe aller Noten des angegebenen Kanals um den gleichen (Cent-)Betrag geändert. Die Einstellung wird mit zwei 7-Bit-Werten vorgenommen, `MSB` für die Grob- und `LSB` für die Feineinstellung. Der Gesamtwert ergibt sich aus der Kombination

beider Werte, also als LSB+128*LSB. Die Mittelstellung ist stets LSB=0x0, MSB=0x40, entsprechend einem Gesamtwert von 0x2000; kleinere Werte verändern die Tonhöhe nach unten, größere nach oben. Die maximale Tonhöhenänderung sollte gemäß der General-MIDI-Spezifikation standardmäßig einen Ganzton nach unten und oben umfassen; bei älteren Synthesizern findet man aber unterschiedliche Default-Einstellungen. Außerdem lässt sich die Sensitivität des Tonhöhen-Rades auch durch Änderung des entsprechenden „RPN"-Parameters einstellen (s.o.).

5.5 System-Common-Nachrichten

Mit den System-Common-Nachrichten werden verschiedene, meist Geräte-abhängige Funktionen gesteuert. Wir behandeln hier nur die wichtigste System-Common-Nachricht, sysex.

sysex BYTES	„System Exclusive"-Nachricht, BYTES = Liste der zu sendenden Byte-Werte (jeder Wert im Bereich 0-127)

Eine sysex-Nachricht setzt sich aus einer (7-Bit-)Byte-Folge beliebiger Länge zusammen. Der Inhalt der Byte-Folge hängt vom jeweiligen Geräte-Hersteller und der aufzurufenden Funktion ab, und beginnt stets mit einer Hersteller-spezifischen Kennung (z.B. 0x43 für Yamaha XG-Synthesizer). Auf diese Weise kann ein Synthesizer die für Geräte eines anderen Herstellers bestimmten Nachrichten erkennen und einfach ignorieren. Eine Aufstellung der zulässigen sysex-Nachrichten findet man normalerweise im Handbuch des Synthesizers. Bei Yamaha-Synthesizern kann man z.B. mit Nachrichten der Form sysex [0x43,0x10,0x4c,AH,AM,AL,D+0x40] die Stimmung des Synthesizers ändern. Dabei adressiert AH, AH, AL einen Eintrag der Stimm-Tabelle (entsprechend einer einzelnen Note innerhalb der MIDI-Oktave, für einen gegebenen MIDI-Kanal), und D bezeichnet die gewünschte Abweichung von der gleichtemperierten Stimmung in Cents.

5.6 System-Realtime-Nachrichten

Die System-Realtime-Nachrichten dienen hauptsächlich zur Synchronisierung verschiedener MIDI-Geräte und -Applikationen. Nützlich sind insbesondere die „Sequencer"-Nachrichten start, stop und continue, mit denen auch die Funktion eines Sequencer-Programms von außen (z.B. von einem MIDI-Keyboard aus) gesteuert werden kann.

active_sense	Einige Synthesizer senden kontinuierlich diese Nachricht, um damit kundzutun, dass sie noch „da" sind.
clock	Die „Clock"-Nachricht wird verwendet, um das Wiedergabe-Tempo verschiedener MIDI-Geräte zu synchronisieren.
start	Starten der MIDI-Wiedergabe
stop	Stoppen der MIDI-Wiedergabe
continue	Fortsetzen der MIDI-Wiedergabe
reset	Zurücksetzen des Synthesizers

Die genaue Wirkung einer reset-Nachricht hängt vom jeweiligen Gerät ab; normalerweise sollte sich ein Synthesizer beim Empfangen einer solchen Nachricht sofort in seine

„Ausgangskonfiguration“ zurückversetzen, was immer vom jeweiligen Gerätehersteller darunter verstanden wird.

5.7 Kodierung von MIDI-Ereignissen

Wie wir in Kapitel 4 gesehen haben, liefert die `midi_get`-Funktion nicht nur eine MIDI-Nachricht, sondern auch einige zusätzliche Informationen, nämlich eine Client-Referenz- und eine Port-Nummer sowie einen „Zeitstempel“, der angibt, wann die Nachricht empfangen wurde. Wenn wir MIDI-Nachrichten aufzeichnen wollen, so ist der Zeitstempel natürlich sehr wichtig, da er angibt, in welcher zeitlichen Abfolge die Nachrichten eingegangen sind. Man speichert daher in einer MIDI-Sequenz normalerweise die MIDI-Nachrichten stets zusammen mit den Zeitwerten. Zu diesem Zweck wird der Zeitstempel üblicherweise mit der entsprechenden MIDI-Nachricht zu einem Paar der Form `(TIME,MSG)` zusammengefasst. Solche Zeit/Nachrichten-Paare werden auch als *MIDI-Ereignisse* bezeichnet. Man beachte, dass also die letzten beiden Komponenten eines von `midi_get` zurückgegebenen Tupels ein MIDI-Ereignis bilden.

6 Echtzeit-Verarbeitung

Unter Echtzeit-Verarbeitung versteht man den Empfang, die Analyse und ggf. die Tranformation und Ausgabe von MIDI-Ereignissen in Echtzeit, d.h., das Programm wartet auf Eingaben vom MIDI-Eingabegerät und verarbeitet diese bei Empfang sofort. Echtzeit-Verarbeitung kommt also bei allen Anwendungen zum Einsatz, mit denen MIDI-Ereignisse unmittelbar z.B. während des Spiels eines Keyboards oder beim Abspielen einer MIDI-Datei weiterverarbeitet werden. Typische Beispiele für solche Anwendungen sind Begleitautomatiken und Akkord-Analyse-Programme.

Ein MIDI-Echtzeit-Programm besteht üblicherweise aus einer Schleife, in der immer wieder die folgenden Schritte abgearbeitet werden:

1. **Eingabe:** Lies ein Ereignis von der MIDI-Eingabe.
2. **Analyse:** Stelle fest, um welches Ereignis es sich handelt und analysiere ggf. die Parameter des Ereignisses.
3. **Ausgabe:** Gib die Ergebnisse der Analyse z.B. auf dem Terminal aus, und/oder erstelle aus der Eingabe ein oder mehrere neue MIDI-Ereignisse und sende diese an die MIDI-Ausgabe.

Schritt 1 ist normalerweise immer gleich, während die weiteren Schritte von der jeweiligen Verarbeitungsfunktion abhängen. **Wichtig:** Damit ein solches Programm tatsächlich in Echtzeit funktionieren kann, dürfen die Verarbeitungsschritte nicht zu aufwendig sein. Die Verarbeitung jedes eingehenden Ereignisses sollte also in einer sehr kurzen Zeit (typischerweise weniger als 1 Millisekunde) abgeschlossen sein, damit keine wahrnehmbaren Latenzen entstehen. Dies ist für Q-Skripts normalerweise gewährleistet, wenn die Berechnung der Verarbeitungsfunktion nicht mehr als einige hundert Reduktionen erfordert. (Der genaue Wert hängt natürlich von der Leistung des Prozessors ab. Im Zweifelsfall sollte man die Funktion `midi_time` verwenden, um die tatsächlichen Latenzzeiten abzuschätzen.)

6.1 Grundform eines Echtzeit-Programms

Ein simples Beispiel eines Echtzeit-Programms hatten wir bereits in `bsp01.q` kennengelernt. Im folgenden behandeln wir nun, wie ein solches Programm im Detail realisiert wird. Zunächst benötigen wir natürlich wie üblich den Import der Q-Midi-Skripts und die Infrastruktur der MIDI-Ein- und Ausgabe. Diese übernehmen wir aus Kapitel 4 (vgl. `bsp08.q`):

```
import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, REF = midi_open "bsp09",
  _ = midi_connect IN REF || midi_connect REF OUT;
```

Als nächstes müssen wir uns über die Organisation des Programms Gedanken machen. Unser Programm hat allgemein die Form einer Schleife, in der eine Verarbeitungsfunktion auf jede eingelesene MIDI-Nachricht angewendet wird. Diese kann als eine endrekursive Funktion realisiert werden (vgl. Abschnitt 3.6), der die anzuwendende Verarbeitungsfunktion als Parameter übergeben wird. Außerdem möchten wir das Programm auch beenden können. Wir müssen also eine Eingabemöglichkeit vorsehen, über die dem Programm mitgeteilt wird, dass die Schleife verlassen werden soll. Viele heutige Synthesizer verfügen zu diesem Zweck über eine „Stop“-Taste, mit der eine MIDI-„Stop“-Nachricht an den PC gesendet wird. Wir brauchen also nur innerhalb der Schleife zu überprüfen, ob eine „Stop“-Nachricht empfangen wurde. Dies ist am einfachsten möglich, wenn wir die erste MIDI-Nachricht bereits vorher abrufen und als Argument an die Schleife übergeben. Die Schleifen-Funktion führt dann die folgenden beiden Schritte aus:

1. Überprüfen auf „Stop“-Nachricht. Falls eine „Stop“-Nachricht empfangen wurde, Ende.
2. Ansonsten Verarbeitung der Nachricht mit der übergebenen Funktion. Einlesen des nächsten Ereignisses und zurück zum Beginn der Schleife (Schritt 1).

Die beiden Schritte lassen sich auf einfache Weise in Form von zwei Gleichungen realisieren:

```
loop F (_,_,_,stop)  = ();
loop F (_,_,_,MSG)   = F MSG || loop F (midi_get REF) otherwise;
```

Diese generische Fassung der Eingabeschleife kann auf beliebige Verarbeitungsfunktionen angewendet werden, die ausschließlich auf dem eingehenden MIDI-Ereignis operieren. (Komplexere Verarbeitungsfunktionen „mit Gedächtnis“ werden im folgenden Abschnitt behandelt.) Betrachten wir als einfaches Beispiel einer Verarbeitungsfunktion die Transposition um eine gegebene Zahl von Halbtonschritten. Hier müssen wir also in allen Nachrichten, die Notennummern enthalten, die Notennummer um den angegebenen Betrag verschieben, was mit einer einfachen Addition zu bewerkstelligen ist. Andere Nachrichten werden unverändert wieder ausgegeben. Zur Unterscheidung der relevanten Nachrichten-Typen benötigen wir vier Gleichungen, wie folgt:

```
transp N (note_on C P V)       = midi_send REF PORT (note_on C (P+N) V);
transp N (note_off C P V)      = midi_send REF PORT (note_off C (P+N) V);
transp N (key_press C P V)     = midi_send REF PORT (key_press C (P+N) V);
transp N MSG                   = midi_send REF PORT MSG otherwise;
```

Zum Schluss fügen wir noch eine Hauptfunktion `transpose` hinzu, mit der die Eingabegeschleife gestartet wird. Voilà! Fertig ist unser Transpositions-Programm:

```
/* bsp09.q: Einfache MIDI-Echtzeit-Anwendung (Transposition) */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, REF = midi_open "bsp09",
  _ = midi_connect IN REF || midi_connect REF OUT;

/* generische Eingabeschleife */

loop F (_,_,_,stop)            = ();
loop F (_,_,_,MSG)             = F MSG || loop F (midi_get REF) otherwise;

/* Transpositions-Verarbeitungsfunktion */

transp N (note_on C P V)       = midi_send REF PORT (note_on C (P+N) V);
transp N (note_off C P V)      = midi_send REF PORT (note_off C (P+N) V);
transp N (key_press C P V)     = midi_send REF PORT (key_press C (P+N) V);
transp N MSG                   = midi_send REF PORT MSG otherwise;

/* Hauptfunktion */

private transpose N;

transpose N                    = midi_flush REF ||
                                 loop (transp N) (midi_get REF);
```

Hinweise:

1. Die Hauptfunktion wird hier explizit als „private" deklariert, da auch die Standard-Bibliothek eine `transpose`-Funktion enthält und wir eine Namenskollision vermeiden wollen.
2. Vor Aufruf der Schleife wird in `transpose` zunächst der Eingabepuffer mit `midi_flush` geleert, um evtl. vor Aufruf der Funktion eingegebene Noten zu löschen.
3. Wenn Sie das Programm mit einem externen Synthesizer testen, erklingen normalerweise sowohl die eingegebenen als auch die transponierten Noten. Um dies zu vermeiden, können Sie die lokale Wiedergabe des Synthesizers abschalten, wodurch nur über die MIDI-Schnittstelle empfangene MIDI-Nachrichten klingen. Falls dies nicht möglich ist, verwenden Sie statt `MIDIDEV!0` für die MIDI-Ausgabe den internen Synthesizer `MIDIDEV!1` und drehen Sie die Lautstärke des externen Synthesizers herunter.
4. Wenn Ihr Synthesizer nicht über eine „Stop"-Taste verfügt, so kann die Schleife nur durch Abbruch der Berechnung unterbrochen werden. Eine sauberere Methode ist in diesem Fall die Ausführung der Transpositions-Funktion „im Hintergrund" (so genannter „Thread"). Dazu wird der Aufruf der `transpose`-Funktion als Argument der Standardbibliotheks-Funktion `thread` übergeben und das Ergebnis einer Variablen zugewiesen. Die `transpose`-Funktion läuft nun im Hintergrund und die Kommandozeile bleibt weiter verfügbar. Wir können daher über einen weiteren „Kontroll"-Client eine `stop`-Nachricht an die Schleife senden, um diese zu beenden:

```
==> def TASK = thread (transpose 5)

==> def CTRL = midi_open "ctrl"; midi_connect CTRL REF
()

==> midi_send CTRL PORT stop
()
```

6.2 Echtzeit-Programme mit Gedächtnis

Die im vorigen Abschnitt besprochene Eingabeschleife ist nur für Verarbeitungsfunktionen geeignet, die ausschließlich auf der eingegebenen Nachricht operieren. Damit lassen sich bereits viele einfache Anwendungen realisieren. Für kompliziertere Verarbeitungsfunktionen muss die Eingabeschleife aber so abgewandelt werden, dass sie die Mitführung eines aktuellen Zustands (eine Art „Gedächtnis") ermöglicht. Ein typisches Beispiel sind Harmonie-Analysatoren, die zur Bestimmung des aktuellen Akkords die Information benötigen, welche Noten momentan klingen, d.h. bereits an- aber noch nicht abgeschaltet wurden. Der Einfachheit halber klammern wir hier das Problem der Akkord-Erkennung aus und betrachten eine einfachere Aufgabenstellung, nämlich die Protokollierung der momentan klingenden Noten auf dem Terminal. Diese soll jedesmal dann erfolgen, wenn sich der aktuelle Akkord ändert, d.h., bei jedem Empfang einer „Note-On"- oder „Off"-Nachricht.

Unsere Verarbeitungsfunktion benötigt hier zusätzlich zum jeweils empfangenen MIDI-Ereignis auch die Menge der zuletzt klingenden Noten. Die neue Noten-Menge ergibt sich dann aus der vorherigen, indem die Note eines „Note-On" der Menge hinzugefügt, und die Note eines „Note-Off" aus ihr entfernt wird. Schließlich muss die Verarbeitungsfunktion auch noch die aktuelle Menge auf dem Terminal ausgeben. Zur Realisierung der Notenmenge kann man den Datentyp `Set` aus der Standard-Bibliothek verwenden. Dieser bietet zwei Funktionen `insert` und `delete`, mit denen Elemente hinzugefügt und entfernt werden können. Unsere Verarbeitungsfunktion lässt sich mit diesen Hilfsmitteln wie folgt implementieren:

```
c NOTES (note_on C P V)        = print NOTES || NOTES
                                   where NOTES = insert NOTES P if V>0;
                               = print NOTES || NOTES
                                   where NOTES = delete NOTES P otherwise;
c NOTES (note_off C P V)       = print NOTES || NOTES
                                   where NOTES = delete NOTES P;
c NOTES MSG                    = NOTES otherwise;
```

Man beachte, dass die Funktion `c` ein zusätzliches erstes Argument hat, die aktuelle Noten-Menge `NOTES`, und als Ergebnis die neue Noten-Menge zurückliefert. Die Ausgabefunktion `print` kann für den Anfang so realisiert werden, dass wir einfach die momentante Noten-Menge als Liste ausgeben (die Standardbibliotheks-Funktion `list` konvertiert dabei einen `Set`-Ausdruck in eine Liste):

```
print NOTES                    = printf "%s\n" (str (list NOTES));
```

Es fehlt noch eine modifizierte Version der Eingabeschleife. Diese hat die zusätzliche Aufgabe, die aktuelle Noten-Menge mitzuführen und als Argument der Verarbeitungsfunktion zuzuführen, was sich mit einem zusätzlichen Parameter `STATE` wie folgt erreichen lässt:

```
loop F STATE (_,_,_,stop)      = ();
loop F STATE (_,_,_,MSG)       = loop F (F STATE MSG) (midi_get REF)
                                   otherwise;
```

Wir fügen nun noch die Hauptfunktion `chord` hinzu, die die Eingabeschleife mit der Verarbeitungsfunktion `c` und dem Initialwert `emptyset` für die leere Noten-Menge aufruft:

```
chord                          = midi_flush REF ||
                                 loop c emptyset (midi_get REF);
```

Ein Beispiel für die Ausgabe des Programms, wenn ein C-Dur Akkord und dann eine „Stop"-Nach - richt eingegeben wird:

```
==> chord
[60]
[60,64]
[60,64,67]
[60,64]
[60]
[]
()
```

Das Programm lässt sich noch verbessern, indem wir statt Notennummern symbolische Notennamen ausgeben. Wir können dazu die folgende kleine `note_name`-Funktion verwenden, die zu einer Notennummer den entsprechenden MIDI-Notennamen zurückgibt. Die Namen der zwölf Noten einer Oktave (inklusive Vorzeichen) werden aus einer Tabelle ermittelt und um die MIDI-Oktav-Nummer (0-10) ergänzt:

```
def NAMES = ("C","C#","D","Eb","E","F","F#","G","G#","A","Bb","B");

note_name P                    = sprintf "%s%d"
                                   (NAMES!(P mod 12), P div 12);
```

Wir gestalten die `print`-Funktion unter Zuhilfenahme der Standardbibliotheks-Funktion `join` so um, dass nun eine von mit Bindestrichen unterteilte Folge von Notennamen ausgegeben wird:

```
print NOTES                    = printf "%s\n"
                                   (join " - " (map note_name (list NOTES)));
```

Das fertige Programm:

```
/* bsp10.q: MIDI-Echtzeit-Anwendung mit Gedächtnis (Akkord-Protokoll) */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, REF = midi_open "bsp10",
  _ = midi_connect IN REF || midi_connect REF OUT;

/* generische Eingabeschleife mit Zustandsparameter */

loop F STATE (_,_,_,stop)      = ();
loop F STATE (_,_,_,MSG)       = loop F (F STATE MSG) (midi_get REF)
                                     otherwise;

/* Verarbeitungsfunktion: Protokollieren des momentan klingenden Akkords */

c NOTES (note_on C P V)        = print NOTES || NOTES
                                     where NOTES = insert NOTES P if V>0;
                               = print NOTES || NOTES
                                     where NOTES = delete NOTES P otherwise;
c NOTES (note_off C P V)       = print NOTES || NOTES
                                     where NOTES = delete NOTES P;
c NOTES MSG                    = NOTES otherwise;

/* Ausgabe des momentanen Akkords */

print NOTES                    = printf "%s\n"
                                   (join " - " (map note_name (list NOTES)));

def NAMES = ("C","C#","D","Eb","E","F","F#","G","G#","A","Bb","B");

note_name P                    = sprintf "%-2s%d"
                                   (NAMES!(P mod 12), P div 12);

/* Hauptfunktion */

chord                          = midi_flush REF ||
                                   loop c emptyset (midi_get REF);
```

Ein Beispiel für die Ausgabe des fertigen Programms:

```
==> chord
C5
C5 - E5
C5 - E5 - G5
C5 - E5
C5

()
```

7 Sequencing

Eine der wichtigsten Grundfunktionen vieler MIDI-Programme ist das so genannte *Sequencing*, d.h. die Aufzeichnung und Wiedergabe von MIDI-Ereignissen, die z.B. über das angeschlossene Keyboard eingegeben werden. Zur Aufzeichnung speichern wir eingehende MIDI-Ereignisse in einer Liste, die auch *MIDI-Sequenz* genannt wird. Eine MIDI-Sequenz hat also die folgende Gestalt:

```
[(T1,MSG1),(T2,MSG2),(T3,MSG3),...]
```

Dabei sind `T1`, `T2`, ... die Zeitwerte und `MSG1`, `MSG2`, ... die zugehörigen MIDI-Nachrichten. Der Einfachheit halber setzen wir im folgenden stets voraus, dass es sich bei den Zeitstempeln immer um *absolute* Werte handelt, die aufsteigend angeordnet sind. Dies erleichtert die Wiedergabe einer Sequenz, da die Ereignisse in genau der Reihenfolge ausgegeben werden können, in der sie in der Liste vorliegen. Die Zeit zwischen zwei benachbarten Ereignissen einer Sequenz ist dann durch die Differenz der entsprechenden Zeitstempel gegeben. Dem Anfangszeitpunkt der Sequenz messen wir hier keine Bedeutung bei; bei der Wiedergabe wird das erste Ereignis also stets sofort ausgegeben.

Im folgenden behandeln wir zunächst die Aufzeichnung einer Sequenz und dann deren Wiedergabe. Schließlich beschäftigen wir uns auch noch damit, wie man beide Funktionen miteinander kombiniert, um während der Wiedergabe einer Sequenz gleichzeitig eine neue Sequenz aufnehmen zu können, und gehen kurz auf die notwendigen Erweiterungen der Algorithmen ein, um mit mehrspurigen Sequenzen arbeiten zu können.

7.1 Aufnahme

Das Aufzeichnen einer MIDI-Sequenz lässt sich recht einfach mit einer MIDI-Eingabe-Schleife mit Gedächtnis realisieren, die wir bereits im vorangegangenen Kapitel kennengelernt hatten. Der Zustands-Parameter der Schleife speichert hier die gesamte Folge der MIDI-Ereignisse, die seit Aufruf der Funktion eingelesen wurden. Wir verwenden dazu eine Liste, in die wir die MIDI-Ereignisse einfügen, sobald diese eintreffen.

```
/* bsp11.q: Aufzeichnung einer MIDI-Sequenz */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, REF = midi_open "bsp11",
  _ = midi_connect IN REF || midi_connect REF OUT;

/* Herausfiltern unerwünschter Ereignisse */

def _ = midi_accept_type REF active_sense false ||
  midi_accept_type REF clock false;

/* Eingabeschleife */

recloop L (_,_,_,stop)          = reverse L;
recloop L (_,_,T,MSG)           = midi_send REF PORT MSG ||
                                  recloop [(T,MSG)|L] (midi_get REF)
                                    otherwise;

/* Hauptfunktion */

public record;

record                          = midi_flush REF || recloop [] (midi_get REF);
```

Man beachte, dass die Hauptfunktion `record` hier als „public“ vereinbart wurde. Damit können wir durch Importieren von `bsp11.q` auf diese Funktion auch in anderen Skripts zurückgreifen. Zur Eingabeschleife selbst (Funktion `recloop`) sind drei Dinge anzumerken:

1. Im Unterschied zu den Echtzeitfunktionen in Kapitel 6 werden hier die kompletten Ereignisse, d.h. Nachrichten *und* Zeitstempel verarbeitet. Wir benötigen die Zeitstempel ja, um die Folge nachher korrekt wiedergeben zu können.
2. Die Sequenz wird hier zunächst in umgekehrter Reihenfolge aufgebaut, da die Ereignisse stets am Beginn der Liste eingefügt werden. Am Ende der Schleife wird die Ergebnisliste dann mit der Standardbibliotheks-Funktion `reverse` umgekehrt. Dies ist effizienter als das Anfügen der Ereignisse jeweils am Ende der Liste. Das Anfügen eines Elements benötigt nämlich eine zur Länge der Liste proportionale Rechenzeit, während das Einfügen am Beginn in konstanter Rechenzeit erledigt wird.
3. Wir gehen hier davon aus, dass die lokale Wiedergabe des MIDI-Eingabegeräts *ausgeschaltet* ist. Daher wird eine empfangene MIDI-Nachricht sogleich wieder ausgegeben. (Dies funktioniert auch, wenn das Eingabegerät ein Keyboard ohne eigene Wiedergabe ist; Sie müssen in diesem Fall nur den internen Synthesizer `MIDIDEV!1` statt `MIDIDEV!0` als Ausgabegerät verwenden.) Falls bei Ihrem MIDI-Synthesizer die lokale Wiedergabe eingeschaltet ist, können Sie den Aufruf von `midi_send` auskommentieren. **Achtung:** Einige Synthesizer ohne separate „MIDI Thru“-Schnittstelle verfügen über eine so genannte „Merge“-Funktion o.ä., die dafür sorgt, dass vom Gerät empfangene MIDI-Nachrichten automatisch wieder über „MIDI Out“ an den PC zurückgesendet werden. Diese Funktion sollte unbedingt ausgeschaltet sein, um Rückkopplungseffekte zu vermeiden!

Als Beispiel für die Verwendung der `record`-Funktion haben wir hier ein C-Dur-Arpeggio aufgezeichnet:

```
==> record
[(25130,note_on 0 60 74),(25780,note_on 0 64 60), (26430,note_on 0 67 ↵
73),(27230,note_on 0 67 0),(27240,note_on 0 64 0),(27240,note_on 0 60 0)]
```

7.2 Wiedergabe

Eine simple Methode zur Wiedergabe einer MIDI-Sequenz hatten wir bereits kennengelernt. Wir können nämlich bei der Ausgabe einer MIDI-Nachricht mit `midi_send` auch die Zeit angeben, zu der die Ausgabe erfolgen soll. Zum Beispiel kann die oben aufgezeichnete Sequenz durch Verschieben des Anfangspunkts auf die momentane MidiShare-Zeit wie folgt wiedergegeben werden:

```
==> do (midi_send REF PORT) [(midi_time,note_on 0 60 74),(midi_time+650,↵
note_on 0 64 60),(midi_time+1300,note_on 0 67 73),(midi_time+2100,↵
note_on 0 67 0),(midi_time+2110,note_on 0 64 0),(midi_time+2110,↵
note_on 0 60 0)]
()
```

Zur Wiedergabe einer umfangreichen MIDI-Sequenz ist diese Methode aber wenig geeignet. Zum einen kann der interne MidiShare-Speicher überlaufen, da die Ereignisse bis zu ihrer Ausgabe zwischengespeichert werden müssen. Zum anderen erlaubt die Methode es nicht, eine laufende Ausgabe zu unterbrechen. Das richtige Verfahren zur Wiedergabe einer MIDI-Sequenz besteht daher darin, die Ereignisse einzeln auszugeben, und zwar erst dann, wenn sie „fällig“ sind. Auf diese

Weise wird der interne MidiShare-Speicher entlastet und die Wiedergabe kann jederzeit beendet werden (z.B. beim Empfang einer `stop`-Nachricht).

Um die Zeit bis zur Fälligkeit eines Ereignisses abzuwarten, verwendet man die Funktion `midi_wait`. Die Funktion wird mit zwei Argumenten, der Referenznummer eines MidiShare-Clients und einem MidiShare-Zeitwert aufgerufen, wartet die Zeitspanne bis zum Eintreffen der gegebenen Zeit ab, und kehrt dann sofort zur aufrufenden Funktion zurück. Der Rückgabewert ist die aktuelle Zeit bei Beendigung von `midi_wait`, die normalerweise mit dem übergebenen Zeit-Parameter identisch ist. Man beachte, dass `midi_wait` im Unterschied zur Systemfunktion `sleep` die Angabe eines *absoluten* Zeitwerts erwartet. Soll also für eine bestimmte Zeitspanne gewartet werden, so muss zu diesem Wert (in Millisekunden) die aktuelle MidiShare-Zeit, `midi_time`, addiert werden. Zum Beispiel erhält man wie folgt eine Pause von einer halben Sekunde:

```
==> midi_wait REF (midi_time+500)
3492140
```

Bei der Wiedergabe einer Sequenz unterscheiden wir zwischen der momentanen *Sequenzzeit* `S` und der tatsächlichen *Echtzeit* `T`. Während der Wiedergabe müssen wir laufend aus den Zeitdifferenzen innerhalb der Sequenz die Echtzeit des jeweils nächsten Ereignisses berechnen. Das erste Ereignis der Sequenz liefert den Startwert der Sequenzzeit `S`, die momentane MidiShare-Zeit den Startwert von `T`. Die Ereignisse werden dann wie folgt in einer Schleife abgearbeitet:

1. Falls das erste Ereignis der Sequenz den Zeitstempel `S` hat, gib das Ereignis aus. (Gegebenfalls kann das Ereignis hier auch protokolliert werden.) Weiter mit der Restliste und Schritt 1.
2. Berechne die Zeitdifferenz `D` zwischen `S` und dem Zeitstempel `S1` des ersten Ereignisses der Liste. Warte bis `T1=T+D`. Dann weiter mit `S=S1`, `T=T1` und Schritt 1.

Die obige Beschreibung lässt sich in folgende endrekursive Funktionsdefinition umsetzen:

```
playloop S T []      = [];
playloop S T [(S,MSG)|SEQ]
                     = midi_send REF PORT MSG || print (T,MSG) ||
                       playloop S T SEQ;
playloop S T SEQ     = playloop S1 T1 SEQ
                         where [(S1,_)|_] = SEQ, D = time_diff S S1,
                           T1 = midi_wait REF (T+D);
```

Die `playloop`-Funktion kann nun von der Hauptfunktion `play` wie folgt aufgerufen werden, wobei wir nur noch die Startwerte für die Sequenz- und Echtzeit festlegen müssen:

```
play []              = [];
play SEQ             = playloop S T SEQ
                         where [(S,_)|_] = SEQ, T = midi_time;
```

Man beachte, dass wir in beiden Definitionen auch den Sonderfall einer leeren Sequenz behandeln müssen. Eine weitere geringfügige Komplikation entsteht bei der Berechnung der Zeitdifferenzen innerhalb von `playloop`: Die interne MidiShare-Uhr ist nämlich ein 32-Bit-Zähler, der nach dem Erreichen des Maximums `0xffffffff` wieder auf `0` zurückspringt. Daher müssen wir alle Zeit-

werte modulo 0x100000000 nehmen und negative Differenzen in positive umrechnen. Dies erledigt die Funktion time_diff:

```
time_diff T1 T2      = ifelse (D>=0) D (D+0x100000000)
                            where D = T2 mod 0x100000000 - T1 mod 0x100000000;
```

Zum Beispiel:

```
==> time_diff 0xfffffff0 0
16
```

Das Protokollieren der Wiedergabe besorgt eine Funktion print, die man z.B. wie folgt realisieren kann:

```
print (T,MSG)        = printf "%10d: %s\n" (T,str MSG);
```

Unser Programm ist nun fast fertig. Wir müssen nur noch eine Abfrage einfügen, um die Wiedergabeschleife durch Eingabe einer stop-Nachricht unterbrechen zu können. Hierbei ist zu beachten, dass wir dazu nicht einfach midi_get aufrufen können, sonst würde die Schleife bis zum Eintreffen eines MIDI-Ereignisses „hängen". Wir können aber mit der Q-Midi-Funktion midi_avail vorher überprüfen, ob ein Ereignis vorhanden ist, dieses im positiven Falle einlesen und überprüfen, ob es sich um ein stop-Ereignis handelt. Dazu ist die Definition von playloop wie folgt zu ergänzen:

```
playloop _ _ SEQ     = SEQ if midi_avail REF and then (midi_get REF!3=stop);
```

Diese Zeile ist vor den anderen Gleichungen für playloop einzufügen, damit die Überprüfung zu Beginn jedes Schleifendurchlaufs stattfindet. Man beachte die Verwendung des logischen Kurzschluss-Operators and then, die hier essentiell ist. Die Abfrage der MIDI-Nachricht mit midi_get darf ja nur dann stattfinden, wenn tatsächlich schon eine Nachricht im Eingabepuffer ist. Außerdem beachte man, dass bei Empfang einer stop-Nachricht der Rest der Sequenz zurückgegeben wird. Auf diese Weise kann die Wiedergabe fortgesetzt werden, indem man den Rückgabewert von play einfach wieder als Argument eines nachfolgenden play-Aufrufs verwendet. Schließlich fügen wir nun noch wie bei der record-Funktion im vorangegangenen Abschnitt einen Aufruf von midi_flush am Beginn der Hauptfunktion ein, damit der Eingabepuffer bei Beginn der Wiedergabe geleert wird. Das fertige Programm ist nun wie folgt:

```
/* bsp12.q: Wiedergabe einer MIDI-Sequenz */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0, REF = midi_open "bsp12",
  _ = midi_connect IN REF || midi_connect REF OUT;

def _ = midi_accept_type REF active_sense false ||
```

```
  midi_accept_type REF clock false;

/* Wiedergabeschleife */

playloop _ _ SEQ      = SEQ if midi_avail REF and then (midi_get REF!3=stop);
playloop S T []       = [];
playloop S T [(S,MSG)|SEQ]
                      = midi_send REF PORT MSG || print (T,MSG) ||
                        playloop S T SEQ;
playloop S T SEQ      = playloop S1 T1 SEQ
                          where [(S1,_)|_] = SEQ, D = time_diff S S1,
                            T1 = midi_wait REF (T+D);

/* Berechnung der Zeitdifferenzen */

time_diff T1 T2       = ifelse (D>=0) D (D+0x100000000)
                          where D = T2 mod 0x100000000 - T1 mod 0x100000000;

/* Protokollieren der Ereignisse */

print (T,MSG)         = printf "%10d: %s\n" (T,str MSG);

/* Hauptfunktion */

public play SEQ;

play []               = [];
play SEQ              = midi_flush REF || playloop S T SEQ
                          where [(S,_)|_] = SEQ, T = midi_time;
```

Zum Testen des Programms können wir nach Starten von `bsp12.q` mit dem `import`-Kommando des Interpreters das Skript `bsp11.q` hinzuladen. Wir zeichnen zunächst eine Folge auf und geben diese dann wieder:

```
==> import bsp11

==> record
[(8244310,note_on 0 60 67),(8244820,note_on 0 64 48),(8245310,note_on 0 67 ↵
43),(8246180,note_on 0 67 0),(8246180,note_on 0 64 0),(8246190,note_on 0 ↵
60 0)]

==> play _
   8254250: note_on 0 60 67
   8254760: note_on 0 64 48
   8255250: note_on 0 67 43
   8256120: note_on 0 67 0
   8256120: note_on 0 64 0
   8256130: note_on 0 60 0
()
```

Es sei an dieser Stelle angemerkt, dass wir aus Gründen der Einfachheit auf die Behandlung einiger Detail-Probleme verzichtet haben und das obige Programm daher noch nicht perfekt ist. Zum einen findet bei jedem Durchlauf der Wiedergabeschleife nur ein Test auf `stop` statt; es wird also jeweils nur ein Ereignis von der MIDI-Eingabe verarbeitet. Damit der Abbruch der Schleife bei dieser Realisierung des Tests ohne große Zeitverzögerung funktioniert, sollten nur `stop`-Ereignisse ein-

gegeben werden. Man kann dies auch dadurch sicherstellen, dass man mittels midi_accept_type alle Ereignisse außer stop aus der Eingabe herausfiltert.

Ein weiterer Mangel des Programms ist, dass bei Unterbrechen der Wiedergabe eventuell noch klingende Noten nicht mehr abgeschaltet werden. Um dies zu vermeiden, kann man ähnlich wie bei bsp10.q in Kapitel 6 in der Eingabeschleife einen zusätzlichen NOTES-Parameter mitführen, in dem die Menge der momentan klingenden Noten gespeichert wird. Bei Abbruch der Wiedergabe müssen dann für alle Elemente von NOTES entsprechende „Note-Off"-Nachrichten gesendet werden. Die dazu notwendigen Erweiterungen des Programms überlassen wir dem Leser zur Übung.

7.3 Gleichzeitige Aufnahme und Wiedergabe

Für die Aufnahme komplizierterer MIDI-Sequenzen ist es nützlich, wenn man die Aufzeichnung in mehreren Durchgängen vornehmen kann. Dazu ist es notwendig, Aufnahme und Wiedergabe miteinander zu kombinieren, wobei die Zeitwerte der neuen Sequenz mit denen der wiedergegebenen Sequenz synchronisiert werden. Eine einfache Methode dafür behandeln wir in diesem Abschnitt. Unser Verfahren verwendet zwei separate MidiShare-Clients, einen für die Wiedergabe und einen für die Aufnahme. Der Aufnahme-Client wird mit der MIDI-Eingabe, der Wiedergabe-Client mit der MIDI-Ausgabe gekoppelt; ferner stellen wir eine Verbindung vom Aufnahme- zum Wiedergabe-Client her, damit eine von der Aufnahmeschleife empfangene stop-Nachricht an den Wiedergabe-Client weitergereicht werden kann:

```
def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0,
  PLAY = midi_open "bsp13 - play", REC = midi_open "bsp13 - rec",
  _ = midi_connect IN REC || midi_connect REC PLAY || midi_connect PLAY OUT;
```

Die Wiedergabefunktion übernehmen wir im wesentlichen unverändert aus bsp12.q – wir müssen nur die Variable REF durch PLAY ersetzen. In der Eingabeschleife sind allerdings umfangreichere Anpassungen notwendig. Erstens führen wir nun genau wie bei der Wiedergabe auch bei der Aufnahme die momentane Sequenz- und Echtzeit mit, um die Zeitstempel der empfangenen MIDI-Ereignisse in Sequenzzeiten umzurechnen. Zweitens muss das „Echo" der empfangenen MIDI-Nachricht nun über den Wiedergabe-Client ausgegeben werden. Drittens schließlich reichen wir eine empfangene stop-Nachricht über den Aufnahme-Client an den Wiedergabe-Client weiter, so dass daraufhin auch die Wiedergabe-Schleife beendet werden kann.

```
recloop S T L (_,_,_,stop)
                    = midi_send REC PORT stop || reverse L;
recloop S T L (_,_,T1,MSG)
                    = midi_send PLAY PORT MSG ||
                      recloop S1 T1 [(S1,MSG)|L] (midi_get REC)
                        where D = time_diff T T1, S1 = S+D;

time_diff T1 T2     = ifelse (D>=0) D (D+0x100000000)
                        where D = T2 mod 0x100000000 - T1 mod 0x100000000;
```

Die Hauptfunktion der Aufnahme, record, wird nun mit einem Parameter, der wiederzugebenden Sequenz aufgerufen. Wir unterscheiden zwei Fälle, je nachdem, ob die Wiedergabe-Sequenz leer ist oder nicht. Im ersten Fall brauchen wir wie in bsp11.q nur die Eingabeschleife aufzurufen; als Startwert für die Sequenzzeit können wir hier einen beliebigen Wert annehmen, z.B. S=0:

```
record []            = midi_flush REC || recloop S T [] (midi_get REC)
  where S = 0, T = midi_time;
```

Im zweiten Fall wird der Startwert der Sequenzzeit auf den Zeitstempel des ersten Ereignisses der Wiedergabe-Sequenz gesetzt. Außerdem muss zusätzlich zur Aufnahme auch die Wiedergabeschleife gestartet werden. Damit Aufnahme und Wiedergabe gleichzeitig ablaufen können, wird erstere als „Thread“ im Hintergrund ausgeführt. Sobald die Wiedergabeschleife beendet ist, ermitteln wir schließlich das Ergebnis des Aufnahme-Threads, d.h. die aufgezeichnete Sequenz; dies erfolgt durch Anwendung der Standardbibliotheks-Funktion `result`, die auf die Beendigung des angegebenen Threads wartet und sein Resultat zurückliefert.

```
record SEQ           = midi_flush PLAY || playloop S T SEQ || result TH
  where [(S,_)|_] = SEQ, T = midi_time,
    TH = midi_flush REC || thread (recloop S T [] (midi_get REC));
```

Damit ist die Definition der `record`-Funktion vollständig. Das neue Programm ist wie folgt:

```
/* bsp13.q: Aufnahme+Wiedergabe von MIDI-Sequenzen */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0,
  PLAY = midi_open "bsp13 - play", REC = midi_open "bsp13 - rec",
  _ = midi_connect IN REC || midi_connect REC PLAY || midi_connect PLAY OUT;

def _ = midi_accept_type REC active_sense false ||
  midi_accept_type REC clock false;

/* Eingabeschleife */

recloop S T L (_,_,_,stop)
                     = midi_send REC PORT stop || reverse L;
recloop S T L (_,_,T1,MSG)
                     = midi_send PLAY PORT MSG ||
                       recloop S1 T1 [(S1,MSG)|L] (midi_get REC)
                         where D = time_diff T T1, S1 = S+D;

time_diff T1 T2      = ifelse (D>=0) D (D+0x100000000)
                         where D = T2 mod 0x100000000 - T1 mod 0x100000000;

/* Wiedergabeschleife */

playloop _ _ SEQ     = SEQ
                         if midi_avail PLAY and then (midi_get PLAY!3=stop);
playloop S T []      = [];
playloop S T [(S,MSG)|SEQ]
                     = midi_send PLAY PORT MSG || print (T,MSG) ||
                       playloop S T SEQ;
playloop S T SEQ     = playloop S1 T1 SEQ
                         where [(S1,_)|_] = SEQ, D = time_diff S S1,
                           T1 = midi_wait PLAY (T+D);

print (T,MSG)        = printf "%10d: %s\n" (T,str MSG);
```

```
/* Aufnahme */

public record SEQ;

record []            = midi_flush REC || recloop S T [] (midi_get REC)
  where S = 0, T = midi_time;

record SEQ           = midi_flush PLAY || playloop S T SEQ || result TH
  where [(S,_)|_] = SEQ, T = midi_time,
    TH = midi_flush REC || thread (recloop S T [] (midi_get REC));

/* Wiedergabe */

public play SEQ;

play []              = [];
play SEQ             = midi_flush PLAY || playloop S T SEQ
                          where [(S,_)|_] = SEQ, T = midi_time;
```

Die Aufnahme zweier Teil-Sequenzen kann nun erfolgen, indem zunächst die `record`-Funktion mit einer leeren Sequenz aufgerufen und das Ergebnis in einer Variablen `SEQ1` gespeichert wird. Danach wird `record` erneut, nun mit `SEQ1` als Wiedergabesequenz, aufgerufen und der zweite Teil der Sequenz in einer neuen Variablen `SEQ2` gespeichert:

```
==> def SEQ1 = record []

==> def SEQ2 = record SEQ1
```

Nun ergibt sich die Frage, wie man beide Sequenzen gleichzeitig abspielt. Dazu müssen wir die Sequenzen zunächst zu einer neuen Sequenz „zusammenmischen", die wieder nach Zeitstempeln aufsteigend sortiert sein muss. Da die beiden Teilsequenzen bereits sortiert sind, kann man dies mit einem einfachen Verfahren bewerkstelligen, das die Grundlage des so genannten „Merge-Sort"-Algorithmus ist. Das Verfahren funktioniert genauso, wie man z.B. zwei vorsortierte Kartenstapel zusammensortiert: Man nimmt stets die kleinste Karte unter den obenliegenden Karten der beiden Stapel und legt diese auf einen dritten Stapel. Nachdem alle Karten von den beiden ursprünglichen Stapeln auf den dritten Stapel abgelegt wurden, ist dieser sortiert. So verfahren wir auch mit unseren beiden Teilsequenzen, nur dass wir nach Zeitstempeln statt nach Kartenwerten sortieren. Der Algorithmus kann in Q z.B. wie folgt implementiert werden:

```
/* Mischen zweier MIDI-Sequenzen */

public mix SEQ1 SEQ2;

mix SEQ1 SEQ2        = SEQ1 if null SEQ2;
                     = SEQ2 if null SEQ1;
                     = [hd SEQ1|mix (tl SEQ1) SEQ2] if T1 <= T2
                          where (T1,_) = hd SEQ1, (T2,_) = hd SEQ2;
                     = [hd SEQ2|mix SEQ1 (tl SEQ2)] otherwise;
```

Unter Zuhilfenahme von `mix` kann man die kombinierte Sequenz nun wie folgt abspielen:

```
==> play (mix SEQ1 SEQ2)
```

7.4 Mehrspurige Sequenzen

Bislang haben wir in diesem Kapitel nur mit einfachen Sequenzen gearbeitet, die sozusagen nur aus einer einzelnen Spur bestehen. In MIDI-Dateien findet man aber normalerweise Sequenzen, die in mehrere Spuren (MIDI-Tracks) aufgeteilt sind. Eine solche Organisation ist recht nützlich, um z.B. die verschiedenen Stimmen eines Musikstücks unterscheiden zu können. Um mehrspurige Sequenzen darzustellen, gibt es in Q im Prinzip zwei verschiedene Möglichkeiten:

1. Man behandelt jede Spur als eine eigene Sequenz. Ein mehrspuriges Stück wird dann als Liste oder Tupel von Sequenzen behandelt.
2. Alle Spuren eines Stückes bilden eine gemeinsame Sequenz. Die Information darüber, zu welcher Spur jedes Ereignis gehört, wird als Nummer im Ereignis gespeichert. Ein Ereignis wird also nun durch ein Tripel `(TRACK,TIME,MSG)` repräsentiert, wobei `TRACK` die Spurnummer des Ereignisses ist.

Die erste Methode erschwert die effiziente Wiedergabe einer Sequenz nicht unerheblich. Wir werden daher die zweite Darstellungsweise verwenden. Diese hat allerdings den Nachteil, dass man nicht mehr einfach durch Indizierung auf eine Spur direkt zugreifen kann, sondern diese bei Bedarf aus der Gesamtsequenz herausfiltern muss (am Ende dieses Abschnitts diskutieren wir eine dafür geeignete Hilfsfunktion).

Betrachten wir nun die Auswirkungen, die die neue Darstellung auf die Funktionen `play` und `record` hat. Für die Funktion `play` ändert sich nur, dass die MIDI-Ereignisse in einer Sequenz nun drei Komponenten haben, von denen die erste (die Spurnummer) bei der Ausgabe an die MIDI-Schnittstelle einfach ignoriert werden kann. Außerdem möchten wir bei der Protokollierung der MIDI-Ereignisse auch die Spurnummer mit ausgeben. Bei der Funktion `record` sind ebenfalls nur einige triviale Änderungen erforderlich. Die naheliegendste Weise zur Erweiterung von `record` besteht darin, dass die Funktion nun mit der gewünschten Spurnummer als Argument aufgerufen wird, der die aufgezeichneten Ereignisse zugeordnet werden sollen.

Auch die Hilfsfunktion `mix` kann leicht an die neue Darstellung angepasst werden. Die für mehrspurige Sequenzen überarbeitete Fassung des Skripts `bsp13.q` ist wie folgt:

```
/* bsp14.q: mehrspurige Aufnahme+Wiedergabe */

import midi, mididev;

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!0,
  PLAY = midi_open "bsp14 - play", REC = midi_open "bsp14 - rec",
  _ = midi_connect IN REC || midi_connect REC PLAY || midi_connect PLAY OUT;

def _ = midi_accept_type REC active_sense false ||
  midi_accept_type REC clock false;

/* Eingabeschleife */

recloop K S T L (_,_,_,stop)
                    = midi_send REC PORT stop || reverse L;
recloop K S T L (_,_,T1,MSG)
                    = midi_send PLAY PORT MSG ||
                      recloop K S1 T1 [(K,S1,MSG)|L] (midi_get REC)
                        where D = time_diff T T1, S1 = S+D;
```

```
time_diff T1 T2      = ifelse (D>=0) D (D+0x100000000)
                         where D = T2 mod 0x100000000 - T1 mod 0x100000000;

/* Wiedergabeschleife */

playloop _ _ SEQ     = SEQ
                         if midi_avail PLAY and then (midi_get PLAY!3=stop);
playloop S T []      = [];
playloop S T [(K,S,MSG)|SEQ]
                     = midi_send PLAY PORT MSG || print (K,T,MSG) ||
                       playloop S T SEQ;
playloop S T SEQ
                     = playloop S1 T1 SEQ
                         where [(_,S1,_)|_] = SEQ, D = time_diff S S1,
                           T1 = midi_wait PLAY (T+D);

print (K,T,MSG)      = printf "%10d: %3d:  %s\n" (T,K,str MSG);

/* Aufnahme */

public record SEQ;

record K []          = midi_flush REC || recloop K S T [] (midi_get REC)
  where S = 0, T = midi_time;

record K SEQ         = midi_flush PLAY || playloop S T SEQ || result TH
  where [(_,S,_)|_] = SEQ, T = midi_time,
    TH = midi_flush REC || thread (recloop K S T [] (midi_get REC));

/* Wiedergabe */

public play SEQ;

play []              = [];
play SEQ             = midi_flush PLAY || playloop S T SEQ
                         where [(_,S,_)|_] = SEQ, T = midi_time;

/* Mischen zweier MIDI-Sequenzen */

public mix SEQ1 SEQ2;

mix SEQ1 SEQ2        = SEQ1 if null SEQ2;
                     = SEQ2 if null SEQ1;
                     = [hd SEQ1|mix (tl SEQ1) SEQ2] if T1 <= T2
                         where (_,T1,_) = hd SEQ1, (_,T2,_) = hd SEQ2;
                     = [hd SEQ2|mix SEQ1 (tl SEQ2)] otherwise;
```

Eine zweispurige Aufnahme erfolgt nun genau wie im vorangegangenen Abschnitt, nur dass neben der Wiedergabe-Sequenz auch noch jeweils die gewünschte Spurnummer angegeben wird, z.B:

```
==> def SEQ1 = record 1 []

==> def SEQ2 = record 2 SEQ1

==> def SEQ = mix SEQ1 SEQ2
```

Zum Abschluss betrachten wir noch die Aufgabe, aus einer mehrspurigen Sequenz eine bestimmte Spur zu extrahieren. Wir definieren dazu die folgende Hilfsfunktion `track`:

```
/* Extraktion einer Spur */

public track K;

track K SEQ          = filter (trackeq K) SEQ;

trackeq K (K,_,_)    = true;
trackeq _ _          = false otherwise;
```

Die Funktion `track` verwendet also die Standardbibliotheks-Funktion `filter` mit einem geeigneten Prädikat `trackeq` zum Herausfiltern aller Ereignisse für eine gegebene Spurnummer. Z.B. kann man nun eine bestimmte Spur einer aufgezeichneten Sequenz `SEQ` wie folgt abspielen:

```
==> play (track 1 SEQ)
```

8 MIDI-Dateien

Wie wir bereits in Kapitel 1 bemerkt haben, können MIDI-Sequenzen, also Folgen von MIDI-Ereignissen, auch in Dateien gespeichert werden. Dazu wird das so genannte *MIDI-Dateiformat* verwendet. Dieses (binäre) Dateiformat ist maschinenunabhängig; MIDI-Dateien können daher auf jedem MIDI-fähigen System abgespielt werden, gleichgültig, auf welchem anderen System sie erstellt wurden. Im Internet findet man umfangreiche Sammlungen von MIDI-Dateien. Für viele Anwendungen kann man daher auf bereits vorhandenes Material zurückgreifen, und muss die MIDI-Sequenzen nicht selbst aufzeichnen. Unsere eigenen Sequenzen können wir in MIDI-Dateien speichern, um sie später wieder einlesen und verwenden zu können. Vorhandene MIDI-Dateien können mit Q-Midi-Anwendungen auch bearbeitet und danach wieder gespeichert werden.

MIDI-Dateien können aus mehreren Teil-Sequenzen bestehen, die man als *Spuren* (Tracks) bezeichnet. Man unterscheidet drei verschiedene Typen von MIDI-Dateien: Typ 0 (enthält immer nur eine Spur), Typ 1 (enthält eine oder mehrere Spuren, die zusammen ein Musikstück bilden) und Typ 2 (enthält eine oder mehrere Spuren, jede Spur stellt ein eigenes Musikstück dar). Der am häufigsten verwendete Typ ist Typ 1.

Im Unterschied zu „gewöhnlichen" MIDI-Sequenzen können MIDI-Dateien auch spezielle „Meta"-Ereignisse enthalten, mit denen zusätzliche Informationen zu einem Stück angegeben werden. Dazu zählen z.B. Tempo, Metrum und Tonart eines Stückes, zusätzliche Instrument- und Spur-Bezeichnungen u.ä. Außerdem werden die Zeitwerte in MIDI-Dateien statt in Millisekunden häufig in *musikalischer Zeit*, d.h. in Unterteilungen von Viertelnoten, angegeben. Zur Wiedergabe von MIDI-Dateien müssen diese Angaben also in Abhängigkeit vom Tempo in *Computer-Zeit* (Millisekunden) umgerechnet werden. Auf die Umrechnung zwischen musikalischer und Computer-Zeit gehen wir im nächsten Abschnitt ein. Danach geben wir einen Überblick über die verschiedenen Typen von Meta-Nachrichten, so wie diese im `MidiMsg`-Datentyp kodiert werden. Schließlich beschäftigen wir uns damit, wie MIDI-Dateien mit Q-Midi eingelesen, gespeichert und bearbeitet werden können.

8.1 Musikalische Zeit vs. Computer-Zeit

Wie wir im letzten Kapitel gesehen haben, werden bei der Aufzeichnung und Wiedergabe einer MIDI-Sequenz die Einsatzzeitpunkte der verschiedenen Ereignisse in physikalischer Zeit gemessen. Wir wollen dieses Zeitmaß, das durch einen internen Zähler wie z.B. `midi_time` repräsentiert wird, als *Computer-Zeit* bezeichnen. Demgegenüber finden wir in den meisten Partituren von Musikstücken überhaupt keine physikalischen Zeitangaben, sondern nur die symbolischen Zeitwerte, die sich aus den Notenwerten ergeben. Dieser Zeitbegriff wird als *musikalische Zeit* bezeichnet. Im Kontext von MIDI wird musikalische Zeit in der symbolischen Einheit „Ticks" (Unterteilungen einer Viertelnote) gemessen. Die Anzahl der Ticks innerhalb einer Viertelnote ist variabel und wird auch als „Auflösung" oder „Pulse je Viertel-Note" („Pulses per Quarter Note", abgekürzt PPQN) bezeichnet. Um Rundungsfehler zu minimieren, verwendet man häufig Werte für PPQN, die Vielfache von 2, 3 und 5 sind, wie z.B. 96, 120, 192, 384 und 768. Bei 96 PPQN entspricht z.B. eine Achtelnote 48 Ticks.

Zur Umrechnung zwischen musikalischer und Computer-Zeit benötigen wir neben PPQN natürlich auch noch das *Tempo*. Musikalisch wird das Tempo üblicherweise in Viertel je Minute (a.k.a. „Schläge je Minute", „Beats per Minute", abgekürzt BPM) angegeben. Aus technischen Gründen verwendet MIDI dagegen Tempoangaben in Mikrosekunden je Viertelnote; dies erlaubt eine bessere Rechengenauigkeiten bei der Umrechnung der Noten- in Zeitwerte. Z.B. entspricht 120 BPM einem MIDI-Tempo von 500.000 µsec/Viertel, also hat eine Achtelnote in diesem Tempo eine Dauer von 250 Millisekunden. Zur Umrechnung zwischen BPM und MIDI-Tempo sowie zwischen Ticks und Millisekunden gelten die folgenden Formeln:

$$\text{TEMPO} = 60.000.000/\text{BPM}$$

$$\text{MSEC} = \text{TICKS}/\text{PPQN}\times\text{TEMPO}/1000$$

In vielen Musikstücken finden wir wechselnde Tempi; in MIDI-Dateien wird das jeweilige Tempo mit der Meta-Nachricht `tempo` angegeben (siehe nächsten Abschnitt). In diesem Fall muss man die Millisekunden-Werte stückweise aus den jeweiligen Tempoabschnitten zusammensetzen. Wird also zunächst eine Viertel-Note in 120 BPM, dann eine Achtel-Note in 100 BPM gespielt, so sind nach den beiden Noten 500+300=800 Millisekunden verstrichen.

8.2 Meta-Nachrichten

Neben gewöhnlichen MIDI-Ereignissen können MIDI-Dateien verschiedene Typen so genannter *Meta-Ereignisse* enthalten, mit denen zusätzliche Informationen über das enthaltene Musikstück kodiert werden. Genau wie ein gewöhnliches MIDI-Ereignis besteht ein Meta-Ereignis aus dem Zeitstempel und der entsprechenden Nachricht. Auch Meta-Ereignisse haben also eine bestimmte Position innerhalb der Sequenz. Insbesondere gelten Meta-Nachrichten zur Kennzeichnung von Tonart (`key_sign`), Metrum (`time_sign`) und Tempo (`tempo`) jeweils ab der Position, an der sie stehen. Diese Typen von Nachrichten findet man in einer MIDI-Datei vom Typ 1 üblicherweise in einem separaten „Tempo-Map"-Track am Beginn der Datei (also in der ersten Spur). Der `MidiMsg`-Datentyp umfasst eine Reihe von Konstruktoren, mit denen alle vom Standard vorgesehenen Meta-Nachrichten dargestellt werden können. Die wichtigsten Typen von Meta-Nachrichten sind in der folgenden Tabelle kurz beschrieben. Eine vollständige Übersicht findet man im `midi.q`-Skript.

`key_sign SIGN KEY`	Bezeichnet die Tonart des Stückes. `SIGN` ist ein Wert zwischen -7 und 7 und gibt die Anzahl der Vorzeichen (7 B's bis 7 Kreuze) an. `KEY` ist 0 für Dur und 1 für Moll. Z.B: `key_sign (-2) 0` = B-Dur.
`time_sign NUM DENOM CLICK QUARTER_DEF`	Bezeichnet das Metrum eines Stückes. `NUM` ist der Zähler und `DENOM` der Zweierlogarithmus des Nenners der Taktart, `CLICK` die Anzahl von MIDI-Clocks (24 Clocks = 1 Viertel) per Metronom-Klick, `QUARTER_DEF` die Anzahl von 32teln in einer Viertelnote. Z.B.: `time_sign 3 2 24 8` = 3/4 mit 1 Metronom-Klick per Viertel und Standard-Viertel-Definition (1 Viertel = 8 32tel). Alle Parameter sind 8 Bit-Werte. Fehlt diese Nachricht, so wird ein Standard-4/4-Metrum angenommen.
`tempo TEMPO`	Gibt das Tempo in Mikrosekunden je Viertel an. Fehlt diese Angabe, so wird ein Default-Wert von 120 BPM (d.h. `tempo 500000`) angenommen.

text TEXT copyright TEXT seq_name TEXT instr_name TEXT lyric TEXT marker TEXT cue_point TEXT	Mit diesen Nachrichten können verschiedene textuelle Informationen in einer MIDI-Datei gespeichert werden. Der TEXT-Parameter ist dabei eine beliebige Zeichenkette. Z.B. werden text und copyright für allgemeine Beschreibungen und Copyright-Informationen und lyric für Liedtexte verwendet. Die seq_name- und instr_name-Nachrichten dienen zur Festlegung eines Sequenz- bzw. Instrument-Namens. Mit der cue_point-Nachricht kann man Einsatz-Punkte für bestimmte „Cues" (z.B. Audio-Dateien) angeben. Die marker-Nachrichten werden in interaktiven Sequencer-Programmen zur Kennzeichnung von Abschnitten innerhalb einer Sequenz verwendet.
end_track	Zeigt das Ende einer MIDI-Spur an. In wohlgeformten MIDI-Dateien *muss* dieses Ereignis am Ende jeder Spur stehen.

8.3 Einlesen von MIDI-Dateien

In Q-Midi werden MIDI-Dateien durch einen speziellen Datentyp MidiFile repräsentiert. Es handelt sich dabei um einen „externen" Datentyp, d.h., die Elemente dieses Typs sind keine „echten" Q-Objekte, sondern in der System-Programmiersprache C implementiert. MidiFile-Objekte werden daher im Interpreter durch das spezielle Symbol <<MidiFile>> angezeigt. Soll von einer MIDI-Datei gelesen werden, so sind dazu die folgenden Schritte notwendig:

1. **Öffnen der MIDI-Datei zum Lesen.** Hierzu ruft man die Funktion midi_file_open mit dem Namen der Datei auf. Die Funktion liefert ein Objekt des MidiFile-Typs zurück, das als Argument für die weiteren Dateioperationen verwendet wird.
2. **Abfrage der Datei-Attribute.** Mit den Funktionen midi_file_format, midi_file_division und midi_file_num_tracks bestimmt man, um welchen Typ von MIDI-Datei (0, 1 oder 2) es sich handelt, in welcher Einheit die Zeitstempel angegeben werden und aus wieviel Spuren die MIDI-Datei besteht.
3. **Einlesen der Spuren.** Um die MIDI-Ereignisse einzeln einzulesen, verwendet man zunächst midi_file_open_track, um die erste Spur zu öffnen. Die MIDI-Ereignisse der Spur werden dann nacheinander mit midi_file_read eingelesen. Schließlich wir die Spur mit midi_file_close_track wieder geschlossen (dies erfolgt automatisch, wenn mit midi_file_read_track über das Ende der Spur hinausgelesen wurde). Diese Prozedur wiederholt man so lange, bis alle Spuren eingelesen wurden. Alternativ dazu kann man auch mit midi_file_read_track eine ganze Spur auf einmal einlesen; das Ergebnis wird dann als Liste zurückgegeben.
4. **Schließen der MIDI-Datei.** Mit der Funktion midi_close wird die MIDI-Datei geschlossen. Diese Operation wird auch automatisch ausgeführt, wenn auf ein MidiFile-Objekt nicht mehr zugegriffen werden kann.

Wichtig: Bei der Verwendung der MIDI-Dateioperationen ist zu beachten, dass diese Operationen aus technischen Gründen nur funktionieren, wenn bereits ein MidiShare-Client mit midi_open registriert wurde.

Ob man besser einzelne MIDI-Ereignisse oder komplette Spuren auf einmal einliest, hängt von der jeweiligen Anwendung ab. Da wir in dieser Einführung immer vollständige Sequenzen auf einmal verarbeiten, bietet sich die zweite Methode an und diese werden wir im folgenden auch verwenden. In jedem Fall werden die MIDI-Ereignisse als `(TIME,MSG)`-Paare kodiert, wobei die Zeitstempel absolute, aufsteigend angeordnete Werte sind. (Tatsächlich werden die Zeitwerte in einer MIDI-Datei als „Deltas" gespeichert, d.h. als Differenzen zwischen aufeinanderfolgenden Ereignissen. Die Umrechnung in absolute Zeitstempel wird automatisch von der MidiShare-Bibliothek vorgenommen.)

Um welche Zeitwerte es sich im Einzelfall handelt, wird mit der `midi_file_division`-Funktion festgestellt. Ist der Rückgabewert eine einzelne Zahl, so gibt diese einen PPQN-Wert an. In diesem Fall sind die Zeitstempel als musikalische Zeitwerte zu verstehen, die mittels der Tempoangaben (`tempo`-Meta-Ereignisse) für die Wiedergabe in Millisekunden umgerechnet werden müssen. Der Rückgabewert von `midi_file_division` kann aber auch ein Paar `(FPS,TICKS)` sein, das ein so genanntes „SMPTE"-Zeitmaß bezeichnet. SMPTE (das wie „Simpty" ausgesprochen wird) steht für „Society of Motion Picture and Television Engineers", die 1967 den unter ihrem Namen bekannten Zeit-Code zur Synchronisierung von Film und Tonspur einführte. In diesem Zeitmaß sind pro Sekunde die durch `FPS` („frames per second") gegebene Anzahl von „Frames" auszugeben, wobei jeder Frame in die gegebenene Anzahl von Ticks unterteilt wird. Das bedeutet, dass jeder Tick `1000/(FPS*TICKS)` Millisekunden entspricht. Der SMPTE-Standard sieht vier mögliche Werte für FPS vor, nämlich 24, 25, 29 oder 30. Z.B. bezeichnet eine SMPTE-Division von `(25,40)` also Zeitstempel in Millisekunden, da 25*40=1000.

Als Beispiel betrachten wir einmal die Datei `prelude3.mid`. Wir starten dazu das `midi.q`-Skript und registrieren einen MidiShare-Client:

```
==> run midi

==> def REF = midi_open "test"
```

Danach können wir die MIDI-Datei wie folgt öffnen:

```
==> def F = midi_file_open "prelude3.mid"
```

Es handelt sich hier um eine Typ 1-Datei mit zwei Spuren und 120 Viertel-Ticks (also musikalische Zeit mit PPQN = 120):

```
==> (midi_file_format F,midi_file_num_tracks F,midi_file_division F)
(1,2,120)
```

Die erste Spur lesen wir wie folgt ein:

```
==> midi_file_read_track F
[(0,port_prefix 0),(0,sysex [65,16,66,18,64,0,127,0,65]),(0,port_prefix ⏎
0),(0,time_sign 3 3 12 8),(0,key_sign 7 0),(0,tempo 545455),(0,marker ⏎
"File Copyright © 1994 by James Kometani.  All rights reserved.  "),⏎
(0,end_track)]
```

Wie man sieht, handelt es sich hier in Übereinstimmung mit dem Typ 1-Format um eine „Tempo-Map"-Spur, die außer der einen `sysex`-Nachricht nur Meta-Ereignisse enthält. Die Signatur-Nachrichten zeigen uns, dass es sich um ein Stück in C#-Dur mit Metrum 3/8 handelt. Das Tempo 545455 entspricht in etwa 110 BPM. Die weiteren MIDI-Ereignisse (1626 an der Zahl) finden sich in der nächsten Spur (wir lassen uns hier nur die ersten 10 Ereignisse anzeigen):

```
==> def SEQ = midi_file_read_track F

==> #SEQ
1626

==> take 10 SEQ
[(0,port_prefix 0),(0,seq_name "Left&Right"),(0,ctrl_change 0 0 0),↵
(0,ctrl_change 0 32 0),(0,prog_change 0 6),(0,ctrl_change 0 7 100),↵
(0,ctrl_change 0 10 64),(0,note_on 0 77 98),(1,note_on 0 49 97),↵
(30,note_on 0 77 0)]
```

Die erste Note des Stücks hat die Nummer 77, in der angegebenen Tonart also ein E#, das für 30 Ticks erklingt. Da PPQN=120, handelt es sich um ein Viertel einer Viertel (30/120=1/4), also eine Sechzehntel-Note, die im gewählten Tempo eine Dauer von etwa 136 Millisekunden (545455/4/1000) hat.

Das Einlesen einer MIDI-Datei mit expliziten Aufrufen der oben beschriebenen Funktionen und die anschließende Umrechnung der Zeitwerte ist ein recht mühseliges Geschäft. Wir definieren uns daher im folgenden eine kleine Hilfsfunktion `load`, mit der wir diese Aufgabe automatisieren können. Wir werden dieser Funktion den Namen der einzulesenden Datei als Parameter übergeben, und die Funktion soll uns dann die komplette, bereits abgemischte Sequenz aller MIDI-Ereignisse der Datei mit in Millisekunden konvertierten Zeitstempeln als Liste zurückgeben. Damit wir auch in der abgemischten Sequenz die einzelnen Spuren auseinanderhalten können, numerieren wir diese durch und verwenden die Mehr-Spur-Darstellung der MIDI-Ereignisse, die wir bereits im vorigen Kapitel eingeführt haben. Die MIDI-Ereignisse werden also als Tripel `(TRACK,TIME,MSG)` kodiert, wobei `TRACK` die jeweilige Spur-Nummer ist.

Wir unterteilen die `load`-Funktion in folgende Arbeitsschritte:

1. Öffnen der Datei.
2. Einlesen und Numerieren der Spuren.
3. Mischen der Spuren.
4. Konvertieren der Zeitstempel.

Die Hauptfunktion sieht dementsprechend wie folgt aus:

```
load NAME            = convert (midi_file_division F)
                         (foldl mix [] (load_tracks F))
                           where F = midi_file_open NAME;
```

Für das Mischen der Spuren verwenden wir die bereits aus `bsp14.q` bekannte Funktion `mix`. Die Gesamtsequenz wird mittels `foldl mix` schrittweise aus den einzelnen Spuren konstruiert. Beginnend mit der leeren Sequenz mischen wir dabei immer das momentane Zwischenergebnis mit der jeweils nächsten Spur zusammen, bis alle Spuren verarbeitet wurden.

Das Einlesen der Tracks können wir wie folgt erledigen. Dabei fügen wir mit Hilfe der Standardbibliotheks-Funktion `cons` am Beginn jeder Nachricht die jeweilige Spur-Nummer (beginnend mit 0) ein.

```
load_tracks F        = map (load_track F)
                        (nums 0 (midi_file_num_tracks F-1));

load_track F K       = map (cons K) (midi_file_read_track F);
```

Die Konvertierung von SMPTE-Zeitstempeln ist ebenfalls nicht sonderlich schwierig:

```
convert (FPS,TICKS) SEQ
                     = map (convert_smpte FPS TICKS) SEQ;

convert_smpte FPS TICKS (K,S,MSG)
                     = (K,round (S/(FPS*TICKS)*1000),MSG);
```

Der einzige etwas trickreiche Teil des Programms ist die Konvertierung musikalischer Zeitstempel auf der Basis der PPQN- und Tempo-Werte. Da das Tempo sich im Lauf der Sequenz ändern kann, benötigen wir hier die momentanen (musikalischen) Sequenz-Zeitwerte `S` und (physikalischen) Millisekunden-Werte `T` sowie den momentanen Tempowert als zusätzliche Zustandsparameter. Zur Umrechnung der Zeitwerte wenden wir den jeweiligen Tempo-Wert auf die Zeitdifferenz zwischen aktuellem und vorhergehenden Ereignis an. Sodann wird, falls eine Tempo-Nachricht verarbeitet wurde, noch der Tempo-Parameter aktualisiert. Man beachte auch, dass als Default-Wert für das Tempo beim Aufruf der `convert_ppqn`-Funktion 500000 (entsprechend 120 BPM) festgelegt wird.

```
convert PPQN SEQ     = convert_ppqn PPQN (500000,0,0) SEQ;

convert_ppqn _ _ []  = [];

convert_ppqn PPQN (TEMPO,S,T) [(K,S1,tempo TEMPO1)|SEQ]
                     = [(K,T1,tempo TEMPO1)|
                         convert_ppqn PPQN (TEMPO1,S1,T1) SEQ]
                          where T1 = T+round (TEMPO/PPQN*(S1-S)/1000);

convert_ppqn PPQN (TEMPO,S,T) [(K,S1,MSG)|SEQ]
                     = [(K,T1,MSG)|convert_ppqn PPQN (TEMPO,S1,T1) SEQ]
                          where T1 = T+round (TEMPO/PPQN*(S1-S)/1000);
```

Fertig! Das komplette Programm in der Übersicht:

```
/* bsp15.q: Einlesen einer MIDI-Datei */

import midi;

def REF = midi_open "bsp15";

public load NAME;
```

```
load NAME            = convert (midi_file_division F)
                       (foldl mix [] (load_tracks F))
                         where F = midi_file_open NAME;

/* Einlesen der Spuren */

load_tracks F        = map (load_track F)
                       (nums 0 (midi_file_num_tracks F-1));

load_track F K       = map (cons K) (midi_file_read_track F);

/* Mischen der Spuren (vgl. bsp14.q) */

mix SEQ1 SEQ2        = SEQ1 if null SEQ2;
                     = SEQ2 if null SEQ1;
                     = [hd SEQ1|mix (tl SEQ1) SEQ2] if T1 <= T2
                         where (_,T1,_) = hd SEQ1, (_,T2,_) = hd SEQ2;
                     = [hd SEQ2|mix SEQ1 (tl SEQ2)] otherwise;

/* Konvertieren der Zeitstempel in Millisekunden */

convert (FPS,TICKS) SEQ
                     = map (convert_smpte FPS TICKS) SEQ;

convert_smpte FPS TICKS (K,S,MSG)
                     = (K,round (S/(FPS*TICKS)*1000),MSG);

convert PPQN SEQ     = convert_ppqn PPQN (500000,0,0) SEQ;

convert_ppqn _ _ []  = [];

convert_ppqn PPQN (TEMPO,S,T) [(K,S1,tempo TEMPO1)|SEQ]
                     = [(K,T1,tempo TEMPO1)|
                        convert_ppqn PPQN (TEMPO1,S1,T1) SEQ]
                         where T1 = T+round (TEMPO/PPQN*(S1-S)/1000);

convert_ppqn PPQN (TEMPO,S,T) [(K,S1,MSG)|SEQ]
                     = [(K,T1,MSG)|convert_ppqn PPQN (TEMPO,S1,T1) SEQ]
                         where T1 = T+round (TEMPO/PPQN*(S1-S)/1000);
```

Wir können nun eine Sequenz wie folgt mit `load` einlesen und dann sofort mit der in `bsp14.q` definierten `play`-Funktion wiedergeben:

```
==> import bsp14

==> def SEQ = load "prelude3.mid"

==> play SEQ
```

Die `load`-Funktion in der oben beschriebenen Form ist allerdings nur für Dateien des Typs 0 oder 1 geeignet. Für Dateien des Typs 2 bildet jede Spur für sich ein Musikstück mit eigenen Tempo-Angaben. In diesem Fall behandelt man jede einzelne Spur so, wie oben gezeigt wurde. Als Ergebnis kann man dann ein Tupel der konvertierten Spuren zurückliefern. Die dazu notwendigen Erweiterungen der `load`-Funktion überlassen wir dem Leser zur Übung.

8.4 Speichern von MIDI-Dateien

Der notwendige Ablauf zum Speichern einer MIDI-Sequenz in einer Datei ist im Prinzip ähnlich wie beim Einlesen:

1. **Öffnen einer MIDI-Datei zum Schreiben.** Dazu verwenden wir die Funktion `midi_file_create` oder `midi_file_append`, je nachdem, ob eine neue Datei erstellt oder einfach nur zusätzliche Spuren an eine vorhandene Datei angehängt werden sollen. Bei der Funktion `midi_file_create` müssen wir neben dem Dateinamen auch das gewünschte Dateiformat (0, 1 oder 2) und die Division (`PPQN` oder `(FPS,TICKS)`) angeben.
2. **Speichern der Spuren.** Mit den Funktionen `midi_file_new_track`, `midi_file_write` und `midi_file_close_track` kann eine Spur Ereignis für Ereignis ausgegeben werden. Alternativ dazu wird eine komplette, als Liste von Ereignissen spezifizierte Spur auf einmal mit `midi_file_write_track` geschrieben. Dies wird solange wiederholt, bis alle Spuren gespeichert sind.
3. **Schließen der Datei.** Mit `midi_file_close` wird die Datei wieder geschlossen. Dies erfolgt auch automatisch, wenn auf das Datei-Objekt nicht mehr zugegriffen werden kann.

Als einfaches Beispiel speichern wir eine aus einem einzelnen Mittel-C bestehende Sequenz in einer Typ 1-Datei mit Millisekunden-Zeitstempeln:

```
==> run midi

==> def REF = midi_open "test"

==> def F = midi_file_create "test.mid" 1 (25,40)

==> midi_file_write_track F [(0,note_on 0 60 127),(500,note_on 0 60 0)]
()
```

Bevor wir die Datei zur Kontrolle wieder einlesen, muss die noch zum Schreiben geöffnete Datei erst geschlossen werden:

```
==> midi_file_close F
()

==> def F = midi_file_open "test.mid"

==> midi_file_read_track F
[(0,note_on 0 60 127),(500,note_on 0 60 0)]
```

Im allgemeinen Fall müssen die Zeitstempel der zu speichernden Sequenz in das Ziel-Format umgerechnet werden (also Millisekunden in die SMPTE- oder PPQN-Zeit der Datei). Außerdem müssen die einzelnen Spuren extrahiert und separat abgespeichert werden. Wir automatisieren diesen Prozess mit der im folgenden definierten Funktion `save`. Wir betrachten hier nur den Fall einer Typ 1-Datei; die notwendigen Anpassungen für Dateien des Typs 0 und 2 überlassen wir wieder dem Leser zur Übung. Unsere `save`-Funktion ist im Prinzip die Umkehrung der `load`-Funktion aus dem letzten Abschnitt. Wir rufen die Funktion mit drei Parametern auf, dem Dateinamen, dem gewünschten Zeitmaß (`(FPS,TICKS)` oder `PPQN`), und der zu speichernden Sequenz. Der Ablauf der Funktion ist wie folgt:

1. Erstellen einer Typ 1-Datei mit dem gewünschten Zeitmaß.
2. Konvertieren der Sequenz in das gewünschte Zeitmaß.
3. Extrahieren und Abspeichern der einzelnen Spuren.

Die Funktion zum Konvertieren der Zeitstempel ist die genaue Umkehrung der Konvertierungsfunktion aus `bsp15.q`. Wir wenden diese Funktion wieder auf die Gesamtsequenz (nicht etwa auf die einzelnen Spuren) an, da die `tempo`-Ereignisse, die wir normalerweise in der ersten Spur einer Typ 1-Datei finden, ja für alle Spuren gelten. Zum Speichern der einzelnen Spuren müssen wir die Spuren dann extrahieren, wozu wir die `track`-Funktion aus Kapitel 7 verwenden, und auch die Spurnummern am Beginn der Ereignisse entfernen; letzteres erledigen wir hier mit der Standardbibliotheks-Funktion `pop`. Die in der Sequenz vorkommenden Spurnummern berechnen wir dabei mit der Funktion `track_nums`, indem wir zunächst die Standardbibliotheks-Funktion `fst` (die das erste Element eines Tupels liefert) auf alle Ereignisse der Sequenz anwenden, die resultierende Liste der Spurnummern mit der Standardbibliotheks-Funktion `set` in eine Menge und dann mit `list` wieder zurück in eine Liste umwandeln. (Die Funktion `list` zur Umwandlung einer Menge in eine Liste hatten wir bereits kennengelernt; die Funktion `set` wandelt umgekehrt eine Liste in eine Menge um. Man kann daher eine Kombination aus `set` und `list` verwenden, um eine Liste zu sortieren und dabei gleichzeitig mehrfach vorkommende Elemente zu eliminieren.)

Das resultierende Programm ist wie folgt:

```
/* bsp16.q: Speichern einer MIDI-Datei */

import midi;

def REF = midi_open "bsp16";

public save NAME DIV SEQ;

save NAME DIV SEQ    = do (save_track F SEQ) (track_nums SEQ)
                         where F:MidiFile = midi_file_create NAME 1 DIV,
                           SEQ = convert DIV SEQ;

/* Bestimmung der Spurnummern */

track_nums SEQ       = list (set (map fst SEQ));

/* Speichern der Spuren */

save_track F SEQ K   = midi_file_write_track F (map pop (track K SEQ));

/* Extrahieren der Spuren (vgl. bsp14.q) */

track K SEQ          = filter (trackeq K) SEQ;

trackeq K (K,_,_)    = true;
trackeq _ _          = false otherwise;

/* Konvertieren der Millisekunden-Zeitstempel */

convert (FPS,TICKS) SEQ
                     = map (convert_smpte FPS TICKS) SEQ;

convert_smpte FPS TICKS (K,T,MSG)
                     = (K,round (T*FPS*TICKS/1000),MSG);
```

```
convert PPQN SEQ     = convert_ppqn PPQN (500000,0,0) SEQ;

convert_ppqn _ _ []  = [];

convert_ppqn PPQN (TEMPO,T,S) [(K,T1,tempo TEMPO1)|SEQ]
                     = [(K,S1,tempo TEMPO1)|
                        convert_ppqn PPQN (TEMPO1,T1,S1) SEQ]
                         where S1 = S+round ((T1-T)*1000*PPQN/TEMPO);

convert_ppqn PPQN (TEMPO,T,S) [(K,T1,MSG)|SEQ]
                     = [(K,S1,MSG)|convert_ppqn PPQN (TEMPO,T1,S1) SEQ]
                         where S1 = S+round ((T1-T)*1000*PPQN/TEMPO);
```

Als Beispiel für die Anwendung der `save`-Funktion speichern wir nochmals eine einfache MIDI-Sequenz, diesmal im musikalischen Zeitmaß mit 96 Ticks je Viertel:[3]

```
==> def SEQ = [(0,0,note_on 0 60 127),(0,500,note_on 0 60 0)]

==> ? save "test.mid" 96 SEQ
()
```

Zur Kontrolle lesen wir die gerade erstellte MIDI-Datei mit unserer `load`-Funktion aus dem letzten Abschnitt wieder ein:

```
==> import bsp15

==> ? load "test.mid"
[(0,0,note_on 0 60 127),(0,500,note_on 0 60 0)]
```

Wie die folgenden Kommandos zeigen, ist die Sequenz innerhalb der Datei tatsächlich im musikalischen Zeitmaß mit PPQN=96 gespeichert:

```
==> def F = midi_file_open "test.mid"

==> midi_file_division F; midi_file_read_track F
96
[(0,note_on 0 60 127),(96,note_on 0 60 0)]
```

Wie man sieht, hat die C-Note in der Datei eine Dauer von 96 Ticks, die bei 96 PPQN der Länge einer Viertelnote, also 500 Millisekunden bei 120 BPM entsprechen. Da die obige Sequenz keine Tempo-Angaben enthält, wird ja das Default-Tempo von 120 BPM angenommen.

3 Man beachte hier das `?`-Kommando am Beginn der Zeile, das dem Interpreter anzeigt, dass ein auszuwertender Ausdruck folgt. Dies ist notwendig, damit der Interpreter unsere `save`-Funktion nicht mit dem speziellen `save`-Kommando verwechselt, das zum Speichern der momentan definierten Variablen-Werte dient. Gleiches gilt auch für die Verwendung von `load` weiter unten.

8.5 Bearbeiten von MIDI-Dateien

Zum Abschluss unserer kleinen Einführung in die MIDI-Programmierung beschäftigen wir uns in diesem Abschnitt mit der Bearbeitung von MIDI-Dateien, wozu wir die Funktionen zum Einlesen und Abspeichern aus den vorangegangenen Abschnitten verwenden. Der grundlegende Ablauf der Bearbeitung ist immer der gleiche:

1. Einlesen einer MIDI-Datei.
2. Bearbeiten der MIDI-Sequenz.
3. Abspeichern der modifizierten Sequenz.

Zum Bearbeiten der Sequenz hat man die in Q vordefinierten Listenfunktionen wie z.B. `map`, `foldl` oder `filter` zur Verfügung. Für die anwendungsspezifischen Bearbeitungsschritte können wir uns dann unsere eigenen Funktionen definieren. Auf diese Weise hat man mit Q-Midi ein sehr flexibles Werkzeug zur automatisierten Bearbeitung von MIDI-Dateien zur Hand, das über die Möglichkeiten der meisten interaktiven Sequencer-Programme weit hinausgeht.

Wir betrachten im folgenden allerdings der Einfachheit halber nur ein elementares Beispiel zur Manipulation einer MIDI-Datei, nämlich die Normalisierung der Dynamik-Parameter von Noten-Ereignissen. Diese Funktion ist z.B. dann praktisch, wenn man mehrere MIDI-Sequenzen auf ein einheitliches Lautstärke-Niveau anheben will. Man erreicht dies, indem man alle Dynamik-Werte mit dem gleichen Faktor multipliziert. Der Faktor wird so gewählt, dass (nach Rundung der resultierenden Werte) der höchste Dynamik-Wert in der Sequenz auf den Maximalwert 127 abgebildet wird. Zur Bestimmung des maximalen Dynamik-Parameters und zur Modifikation der Dynamik-Werte bietet sich der Einsatz der generischen Listenfunktionen `map` und `foldl` an.

Betrachten wir zunächst die Bestimmung des Faktors, um den die Dynamikwerte verstärkt werden müssen. Dieser beträgt `127/MAX`, wobei `MAX` der maximale Dynamik-Wert eines `note_on`-Ereignisses ist. Ist `SEQ` die Eingabe-Sequenz, so können wir `MAX` berechnen als `foldl max 0 (map vel SEL)`, wobei die folgende `vel`-Funktion dazu dient, die Dynamik-Werte aus den Ereignissen der Sequenz zu extrahieren:

```
vel (_,_,note_on _ _ V)         = V;
vel _                           = 0 otherwise;
```

Ist `MAX>0`, so können wir nun folgende Funktion `amp` mit `FACT=127/MAX` auf die einzelnen Ereignisse anwenden, um die Dynamik-Werte auf das gewünschte Niveau anzuheben. (Falls `MAX=0` ist, so enthält die Datei keine echten „Note-On"-Ereignisse und eine Normalisierung ist daher nicht möglich.)

```
amp FACT (K,T,note_on C P V)    = (K,T,note_on C P (round (FACT*V)));
amp FACT EV                     = EV otherwise;
```

Es fehlt jetzt nur noch die Hauptfunktion, die die gewünschte MIDI-Datei einliest, den maximalen `vel`-Wert berechnet, mit `amp` die Dynamik-Werte anhebt, und schließlich das Ergebnis wieder abspeichert. Für die Dateioperationen verwenden wir einfach die Operationen aus den beiden vorhergehenden Beispielen. Das fertige Programm ist wie folgt:

```
/* bsp17.q: Dynamik-Normalisierung einer MIDI-Datei */

import midi, bsp15, bsp16;

def REF = midi_open "bsp17";

public normalize NAME;

normalize NAME                 = save NAME DIV (map (amp (127/MAX)) SEQ)
                                     if MAX>0
                                     where SEQ = load NAME,
                                       MAX = foldl max 0 (map vel SEQ),
                                       DIV = midi_file_division
                                             (midi_file_open NAME);

/* Bestimmung der Dynamikwerte */

vel (_,_,note_on _ _ V)        = V;
vel _                          = 0 otherwise;

/* Verstärkung der Dynamikwerte */

amp FACT (K,T,note_on C P V)   = (K,T,note_on C P (round (FACT*V)));
amp FACT EV                    = EV otherwise;
```

Das folgende Beispiel zeigt die Anwendung von `normalize` auf unsere Beispiel-Datei `prelude3.mid` (da die Datei von `normalize` überschrieben wird, speichern Sie bitte eine Kopie der Original-Datei an einem sicheren Platz). Schauen wir uns zunächst einmal an, welche Dynamik-Werte in der Original-Datei vorkommen. Die Berechnung der verschiedenen Werte kann man wie im vorangegangenen Abschnitt mit den Funktionen `set` und `list` erledigen, die wir hier auf die `vel`-Werte der Ereignisse in der Sequenz anwenden:

```
==> list (set (map vel (load "prelude3.mid")))
[0,95,96,97,98,99,100,101,102,103,104,105]
```

Die Anwendung von `normalize`:

```
==> normalize "prelude3.mid"
()
```

Welche Dynamik-Werte finden wir nun in `prelude3.mid`?

```
==> list (set (map vel (load "prelude3.mid")))
[0,115,116,117,119,120,121,122,123,125,126,127]
```

Wie man sieht, wurden die Werte wie gewünscht so angehoben, dass das Maximum nun 127 ist.

Will man übrigens nicht nur eine, sondern sämtliche MIDI-Dateien in einem Verzeichnis normalisieren, so kann man dazu die Standardbibliotheks-Funktion `glob` verwenden, die aus einem Dateinamen-Muster die Liste der tatsächlichen Dateinamen berechnet. Z.B.:

```
==> do normalize (glob "*.mid")
```

Wir haben hier nur eine recht einfache Anwendung skizziert. Auf ähnliche Weise kann man den Dynamik-Bereich einer MIDI-Sequenz auch komprimieren oder expandieren, ähnlich wie man dies mit Audio-Dateien tut. Durch Verwendung entsprechender Bearbeitungs-Funktionen kann man auch MIDI-Kanalnummern der Voice-Ereignisse ändern, Controller-Werte transformieren, die Zeitstempel variieren, usw. Da man mit Q eine universelle Programmiersprache zur Hand hat, lassen sich mit Q-Midi und den in diesem Kapitel besprochenen Hilfsfunktionen alle Transformationen von MIDI-Dateien durchführen, die überhaupt „berechenbar“ sind, wobei die generischen Listenfunktionen von Q ein wesentliches Hilfsmittel darstellen, mit dem man viele Programme auf einfache Weise realisieren kann.

Anhang

A Installationshinweise

Um die in dieser Einführung vorgestellten Beispiele selbst am PC nachvollziehen zu können, benötigen Sie eine Installation der Q-Programmierumgebung und der Q-Midi-Schnittstelle auf Ihrem Computer. Im folgenden wird kurz erklärt, welche Software dazu installiert werden muss und welche weiteren Konfigurationsschritte nötig sind.

Voraussetzungen: Sie benötigen einen PC mit Betriebssystem Linux oder Windows, Soundkarte mit MIDI-Schnittstelle, sowie ein über die MIDI-Schnittstelle angeschlossenes Keyboard. Die Beispiel-Anwendungen sind so formuliert, dass sowohl die MIDI-Eingabe als auch die Ausgabe über die externe MIDI-Schnittstelle erfolgt. Wenn Ihr Keyboard ein reines Eingabegerät ist, können Sie die Ausgabe alternativ auch über den internen MIDI-Synthesizer der Soundkarte (bzw. einen Software-Synthesizer) vornehmen. Dazu ersetzen Sie in den Beispiel-Programmen jeweils die Gerätebezeichnung `MIDIDEV!0` bei der Definition des Ausgabe-Clients durch `MIDIDEV!1`; vgl. Abschnitt 4.2.

Download der Software: Die Q-Programmierumgebung und Q-Midi-Schnittstelle steht zum freien Download auf der „Q-Homepage“ bereit: http://www.musikwissenschaft.uni-mainz.de/~ag/q. Dort finden Sie auch die Lizenzbedingungen sowie Links zum Download weiterer benötigter Software wie z.B. MidiShare [http://www.grame.fr/MidiShare/] und Tcl/Tk [http://www.tcl.tk].

Die folgenden Installationshinweise beziehen sich auf die zur Zeit der Drucklegung gültigen Versionsnummern; ggf. ersetzen Sie diese durch die aktuellen Versionen.

Linux-Installation: Zur Installation der Linux-RPM-Pakete benötigen Sie den „RedHat Package Manager“, der im Lieferumfang aller gängigen Linux-Distributionen enthalten ist; unter SuSE Linux kann die Installation auch mit YaST2 erfolgen.

Zunächst benötigen Sie eine Installation des MidiShare-Kernel-Moduls und der entsprechenden Bibliotheken und Utilities auf Ihrem Linux-System. Der Original-Quellcode von MidiShare ist unter der URL http://www.grame.fr/MidiShare/Install/Download.html erhältlich. Folgen Sie den Installationsanweisungen in diesem Paket, um MidiShare zu compilieren und auf Ihrem System zu installieren. Falls Sie über ein SuSE Linux 8.1-System verfügen, so können Sie stattdessen auch einfach das Paket midishare-1.86-3.i386.rpm von der Q-Homepage installieren.

Anschließend installieren Sie die Pakete q-4.1.3-1.i386.rpm und q-midi-1.8.2-1.i386.rpm von der Q-Homepage. (Natürlich können Sie auch hier statt der RPM-Pakete direkt „vom Quellcode“ installieren; laden Sie dazu die entsprechenden Quellcode-Pakete mit der Dateinamen-Endung `tar.gz` von der Q-Homepage herunter.)

Für den Betrieb des Q-Midi-Players benötigen Sie außerdem Tcl/Tk. Dieses ist in allen neueren Linux-Distributionen enthalten, Sie brauchen also nur sicherzustellen, dass Sie die entsprechenden Pakete von Ihrer Linux-CD installiert haben.

Beachten Sie nach der Installation der Pakete auch bitte die weiteren Hinweise in den Dateien `etc/README` und `etc/README-Midi` im Q-Verzeichnis (`/usr/share/q`). Dort wird zum Beispiel beschrieben, wie Sie den XEmacs-Editor für die Bearbeitung von Q-Skripts konfigurieren, und wie Sie den Software-Synthesizer „iiwusynth“ auf Ihrem System installieren, der mit Q-Midi als interner Synthesizer unter Linux verwendet werden kann.

Windows-Installation: Für die Installation der Windows-MSI-Pakete wird der „Microsoft System Installer“ benötigt; die Installation wird dann jeweils mit einem Doppelklick auf die entsprechenden MSI-Dateien gestartet. Der Windows Installer ist Bestandteil aller neueren Windows-Systeme und kann bei Bedarf auch unter folgender URL heruntergeladen werden:

http://www.microsoft.com/msdownload/platformsdk/instmsi.htm.

Zur Installation von Q und Q-Midi werden die Pakete Qpad-4.1.3GER.msi und Q-Midi-1.8.2.msi benötigt. Das erste Paket umfasst eine komplette Windows-Entwicklungs-Umgebung für Q, „Qpad" genannt. Das zweite Paket enthält neben dem Q-Midi-Modul auch alle zum Betrieb von MidiShare benötigten Dateien. Folgen Sie den Anweisungen der Installationsprogramme, um die Software auf Ihrem Rechner (z.B.) im Verzeichnis `C:\Programme\Qpad` zu installieren.

Für den Betrieb des Q-Midi-Players wird außerdem Tcl/Tk benötigt. Eine frei verfügbare Windows-Version von Tcl/Tk ist bei ActiveState [http://www.activestate.com] erhältlich unter der URL http://aspn.activestate.com/ASPN/Downloads/ActiveTcl/.

Wichtig: Bevor Sie Q-Midi unter Windows verwenden können, müssen Sie mit dem `msDrivers`-Programm im Qpad-Verzeichnis die MidiShare-„Ports" geeignet konfigurieren. Genauere Informationen zum `msDrivers`-Programm finden Sie nach der Installation des Q-Midi-Pakets im Unterverzeichnis `MidiShare Docs` des Qpad-Verzeichnisses. Stellen Sie die Port-Definitionen mit `msDrivers` so ein, dass Port 0 für die Ein- und Ausgabe auf der externen MIDI-Schnittstelle, und Port 1 für die Ausgabe auf dem internen Synthesizer der Soundkarte konfiguriert ist. Nach erfolgreichem Abschluss der `msDrivers`-Konfiguration sollte das Qpad-Verzeichnis eine Datei `msMMSystem.ini` enthalten. Kopieren Sie diese zusammen mit den weiteren `*.ini`-Dateien in Ihr `WINDOWS`- (bzw. `WINNT`-) Verzeichnis. Ihr System sollte nun für den Einsatz von Q-Midi vorbereitet sein.

Test der Installation: Um Ihre Q-Midi-Installation zu testen, können Sie das mitgelieferte Beispielprogramm `midi_examp.q` verwenden; Sie finden dieses unter `examples/midi` im Q- bzw. Qpad-Verzeichnis. Verwenden Sie die `record`-Funktion, um eine Sequenz aufzuzeichnen. Zur Beendigung der Aufnahme geben Sie einen Zeilenvorschub ein. Danach können Sie die aufgezeichnete Sequenz mit der `play`-Funktion wiedergeben.

```
==> def SEQ = record
Recording. Press <CR> to stop.
<CR>

==> play SEQ
Playing. Press <CR> to stop.
```

Wenn Ihre Q-Midi-Schnittstelle korrekt konfiguriert ist, sollte die aufgezeichnete MIDI-Sequenz nun auf dem Ausgabegerät erklingen. *Hinweis:* Falls statt eines angeschlossenen Synthesizers der interne Synthesizer (bzw. „iiwusynth" unter Linux) für die Wiedergabe verwendet werden soll, so müssen Sie zunächst am Beginn des Skripts in der Definition des Ausgabegeräts `MIDIDEV!0` durch `MIDIDEV!1` ersetzen (Änderung durch Fettdruck hervorgehoben):

```
def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!1;
```

Tip: Sollte Q-Midi unter Windows trotz korrekter Konfiguration keine Ausgabe erzeugen, so prüfen Sie bitte, ob möglicherweise bereits eine andere MIDI-Anwendung geöffnet ist, die auf die selbe MIDI-Schnittstelle zugreift. Windows hat die unangenehme Eigenschaft, dass ein MIDI-Ein- oder Ausgabegerät normalerweise nur von einem Programm gleichzeitig benutzt werden kann. Enthält das Verzeichnis, in dem ein Q-Midi-Programm ausgeführt wird, nach Starten des Programms eine `*.log`-Datei, so finden Sie darin u.U. eine Fehlermeldung, die die Ursache des Problems beschreibt. Zur genaueren Diagnose von Problemen mit der MIDI-Eingabe und -Ausgabe unter

Windows empfiehlt sich der Einsatz der „MidiOx“-Software und so genannter „Loopback-Devices“ wie z.B. „MidiYoke“, siehe http://www.midiox.com.

Beispielprogramme: Die Beispielprogramme in dieser Einführung finden Sie als „Zip“-Datei auf der Homepage des Bereichs Musikinformatik, unter der URL http://www.musikwissenschaft.uni-mainz.de/Musikinformatik/, in der „Download“-Sektion.