

# Q-Midi: A MidiShare Interface for the Q Programming Language

Albert Gräf

March 23, 2003

## Abstract

This paper is about Q-Midi, an interface for developing MIDI applications in the Q programming language. Q is a modern functional language based on term rewriting; this means that a Q program is just a collection of equations which are used as rewriting rules to simplify expressions. Q-Midi represents MIDI events as symbolic data which makes it easy to formulate functional programs to manipulate and process MIDI sequences on a high level of abstraction. Realtime programming and the manipulation of MIDI files are also supported. Q-Midi is based on Grame's MidiShare, a C library for portable MIDI programming. The interface works on Linux, Mac OS X and Windows systems. Therefore Q-Midi provides an interesting tool for developing portable computer music applications in a high-level functional programming language.

## 1 Introduction

Despite its limitations, the MIDI (“Musical Instruments Digital Interface”) format [8] is the established portable and application-independent way of representing symbolic musical data. Computer music applications typically take a sequence of MIDI events, either from a MIDI file or directly from the computer's MIDI interface, as their input, analyze it and then possibly transform it to an output sequence. MIDI input might even be processed in realtime, meaning that the program directly responds to events entered by the user on a connected keyboard. Coding MIDI applications can be an awkward task because programming interfaces vary considerably between different operating systems, and the programmer often has to deal with obnoxious technical detail to realize even the simplest kinds of applications. Fortunately, there now are MIDI programming libraries like Grame's MidiShare [2] which simplify the development process by providing a portability layer between the application and the operating system, as well as additional facilities like event queues and timers for realizing realtime applications.

But MIDI application development can be facilitated even further by employing a high-level functional programming language. To these ends we have developed a MidiShare interface, called Q-Midi, for the equational programming language Q [3]. Q-Midi provides the necessary facilities to access the MIDI interface and MIDI files in the Q language, including realtime processing. Using Q-Midi, one can implement typical MIDI processing tasks, such as the filtering, analysis, transformation and construction of MIDI sequences, with comparatively little effort. Employing Q's multithreading capabilities it is possible to make parallel background processing tasks work in concert. Last but not least, the Q interpreter allows

the programmer to experiment with the MIDI interface and quickly test out MIDI processing functions in a convenient interactive environment.

The paper is organized as follows. We first give an introduction to the Q programming language. Then we describe the Q-Midi interface and discuss some important aspects of the interface, in particular, setting up and connecting MidiShare clients, the representation of MIDI messages and events, MIDI input/output and timing, and the handling of MIDI files. Finally we cover some typical applications, namely realtime event manipulation, sequencing and algorithmic composition.

## 2 The Q language

Q is rather different from conventional programming languages, so we briefly review its most important features here. If you are already familiar with ML or Haskell, it might be enough to skim through this section since Q has many similarities with these languages. A full description of the language can be found in [3]. If you wish to actually try out the examples you can find the Q interpreter for download on the author's website.<sup>1</sup>

Q is a fairly simple language which is based entirely on the notion of *term rewriting*. A Q program, called a *script*, is merely a collection of *equations* which are used as *rewriting rules* to reduce expressions to *normal form*. If we leave away the jargon, this simply means that an expression is evaluated in a symbolic fashion by repeatedly applying equations. The resulting expression, to which no more equations are applicable, is said to be in “normal form” and constitutes a “value” in the Q language.

Despite its conceptual minimalism, Q is *not* a toy language. First, term rewriting is a powerful computational model which allows you to formulate your programs in an abstract, declarative style. Second, the Q interpreter implements term rewriting very efficiently so that it can compete in execution speed with other interpreted languages. Third, as we point out below, standard programming elements such as branches and loops can also be conveniently expressed in a purely equational style. Last but not least, Q comes with a comprehensive standard library, which provides the usual general operations and system functions programmers have come to expect from modern programming languages, including full multithreading support.

### 2.1 Expressions

Fig. 1 summarizes the different types of simple and compound expressions in the Q language. We remark that Q integers are “bignums”, implemented with the GNU multiprecision package, and thus can get as large as you like (and RAM allows). Floating point values are the usual 64 bit “double precision” ones. Strings are null-terminated, as in C. The lexical syntax of symbols is similar to Prolog; thus variables are distinguished from function symbols in that they start with an uppercase letter. E.g., `foo` denotes a function symbol, while `BAR` is a variable. Using this convention, symbol declarations can mostly be avoided, and in fact Q is a language without mandatory declarations.

As fig. 1 shows, Q has two different data structures for encoding sequences of expressions, namely *lists* and *tuples*. Internally, lists are encoded using right recursive structures (using pointers) which makes it easy to insert and delete elements, whereas tuples are represented

---

<sup>1</sup>See <http://www.musikwissenschaft.uni-mainz.de/~ag/q>.

| Simple expressions    |                           | Compound expressions |  |
|-----------------------|---------------------------|----------------------|--|
| Type                  | Examples                  | Type                 | Examples                                 |
| Integer               | 12345678<br>0xa0<br>-033  | List                 | []<br>[a]<br>[a,b,c]                     |
| Floating point number | 0.<br>-1.0<br>1.2345E-78  | Tuple                | ()<br>(a)<br>(a,b,c)                     |
| String                | ""<br>"abc"<br>"Hello!\n" | Function application | sin 0.5<br>max X (sin Y)<br>(*2) (sin X) |
| Symbol                | foo<br>BAR                | Operator application | -X<br>X+Y<br>X or Y                      |

Figure 1: Expression types.

as contiguous sequences in memory (like “arrays” in conventional programming languages) which allows compact storage and fast access to individual tuple members. Otherwise the two data structures are very similar. Q provides a variety of built-in operations on lists and tuples, such as determining the length, indexing and concatenation.

The most important construct in the Q language is the *function application* which is simply denoted by juxtaposition. Thus, e.g., `sin X` denotes the application of the (built-in) sine function to an argument `X`. Applications to multiple arguments is achieved using nested single-argument applications. For instance, `max X Y` is an application of the standard library “maximum” function `max` to two arguments `X` and `Y`, which is actually interpreted as `(max X) Y`. That is, applying `max` to `X` yields another function `max X` which, when applied to the argument `Y`, computes the maximum of `X` and `Y`. This style of writing function applications is also known as “Currying”, after the American logician Haskell B. Curry, and is ubiquitous in modern functional programming languages.

For the most common kinds of arithmetic, relational and logical operations the Q language provides the usual infix or prefix operator symbols, such as `+` and `*`. A table of the most important operators, listed in order of decreasing precedence, is shown in fig. 2. All binary operators are left-associative, except the exponentiation/index operators which are right-associative, and the relational operators which are non-associative. Function application takes precedence over all these operators. Default precedences can be overridden by grouping expressions with parentheses as usual.

Currying also applies to operator applications which are actually nothing but “syntactic sugar” for function applications. As in Haskell and ML, by enclosing an operator symbol like `+` in parentheses, we turn the operator into an ordinary prefix function; e.g., `(+) 1 2` is exactly the same as `1+2`. So, for instance, `(* 2)` denotes the function which doubles its argument. Q also supports the use of so-called *operator sections*, in which a binary operator is specified with only its left or right operand. Thus, e.g., `(+1)` is a function which increments its argument by 1, and `(1/)` denotes the reciprocal function which computes the inverse of its argument.

| Group                       | Operators | Meaning                           | Example                 |
|-----------------------------|-----------|-----------------------------------|-------------------------|
| Exponentiation/<br>indexing | $\wedge$  | exponentiation                    | $X\wedge Y$             |
|                             | !         | indexing                          | $X!Y$                   |
| Unary prefix<br>operators   | -         | unary minus                       | $-X$                    |
|                             | #         | length/size                       | $\#X$                   |
|                             | not       | logical negation                  | not X                   |
| Multiplication<br>operators | *         | multiplication                    | $X*Y$                   |
|                             | /         | division                          | $X/Y$                   |
|                             | div       | integer division                  | $X \text{ div } Y$      |
|                             | mod       | remainder of integer division     | $X \text{ mod } Y$      |
|                             | and       | logical/bitwise conjunction       | $X \text{ and } Y$      |
|                             | and then  | short-circuit logical conjunction | $X \text{ and then } Y$ |
| Addition<br>operators       | +         | addition                          | $X+Y$                   |
|                             | -         | subtraction                       | $X-Y$                   |
|                             | ++        | concatenation                     | $X++Y$                  |
|                             | or        | logical/bitwise disjunction       | $X \text{ or } Y$       |
|                             | or else   | short-circuit logical disjunction | $X \text{ or else } Y$  |
| Relational<br>operators     | <         | less than                         | $X<Y$                   |
|                             | >         | greater than                      | $X>Y$                   |
|                             | <=        | less than or equal                | $X<=Y$                  |
|                             | >=        | greater than or equal             | $X>=Y$                  |
|                             | =         | equal                             | $X=Y$                   |
|                             | <>        | not equal                         | $X<>Y$                  |
|                             | in        | in relation                       | $X \text{ in } Y$       |
| Sequence operator           |           | execute expressions in sequence   | $X  Y$                  |

Figure 2: Operators.

## 2.2 Equations and expression evaluation

In the Q language, expressions are evaluated from left to right, innermost expressions first, using the equations of the script supplied by the programmer. This is also known as “applicative order” or “call by value” evaluation, since a function and its argument are evaluated before the function is applied. (Q also provides means to declare functions with “call by name” evaluation order which is useful to realize “lazy” data structures and operations; but we will not go into that here.) The Q interpreter includes a collection of certain built-in operations, like basic arithmetic, logical operations etc. These operations can be thought of as being implemented by a large collection of “built-in equations”. The built-in rules always take precedence.

A user-defined equation consists of a left-hand side and a right-hand side expression, delimited with the symbol ‘=’ and terminated with a semicolon. For instance, the following equation defines a function `sqr` which squares its argument by multiplying it with itself:

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

(We remark here that the Q language has a free format, i.e., whitespace is generally ignored, except if it serves as a delimiter between adjacent symbols. Thus equations can be formatted in any desired manner. A script may also include comments which have the same syntax as in C/C++.)

Equations are always applied from left to right. Thus the above definition tells the interpreter that whenever it sees an expression of the form `sqr X`, which denotes the application of the function `sqr` to an argument `X`, it should be replaced with `X*X`. For instance, `sqr`

5 reduces to 5\*5, which in turn reduces to 25 employing the built-in definition of the ‘\*’ operator.

An equation may also contain a *condition* part of the form `if X` where *X* can be any expression which evaluates to a truth value (`true` or `false`). For instance, the following definition introduces a function `odd` which checks whether its (integer) argument is an odd number:

```
odd X          = true if X mod 2 = 1;
                = false otherwise;
```

Note that the above definition actually consists of *two* equations for the same left-hand side `odd X`. In general, if an equation does not specify a left-hand side, it is assumed to have the same left-hand side as the previous equation. If more than one equation is applicable to a given expression, the equations are tried in the order in which they occur in a script. Hence, using the above definition, the interpreter will first check the condition `X mod 2 = 1` on the first equation. This expression must evaluate to a truth value, otherwise the interpreter will generate a runtime error. If it evaluates to `true`, the first equation will be applied, yielding `true` as the value of `odd X`. Otherwise the second equation is applied, yielding `false`. We indicated that this is the “default case” of the definition with the keyword `otherwise`; this is nothing but syntactic sugar, but it tends to improve the readability of definitions like the one above.

Of course, the `odd` function could also be defined more concisely as follows:

```
odd X          = (X mod 2 = 1);
```

Note that in this case the comparison on the right-hand side has to be parenthesized, to resolve any ambiguities, as ‘=’ is used both as the equality operator and to delimit the two sides of an equation.

Equations can also be used to define *recursive* functions, i.e., functions which are defined in terms of themselves. In modern functional languages recursion is actually used to realize all kinds of repetitive control structures; we will have to say more about this in the following section. As a simple example, consider the *faculty* function which computes the product of all positive integers up to a given integer *N*:

```
fac N          = N*fac (N-1) if N>0;
                = 1 otherwise;
```

Finally, the left-hand side of an equation may also contain constants or structured arguments. For instance, as in Prolog we may use `[]` to denote the empty list and `[X|Xs]` to denote a list with head element *X* and list of remaining elements *Xs*. Using these constructs we can implement Lisp-style “`car`” and “`cdr`” operations as follows:<sup>2</sup>

```
car [X|Xs]     = X;
cdr [X|Xs]     = Xs;
```

As another reminiscence of Prolog, we can indicate argument values we do not care about, like the *Xs* value in the above definition of `car`, using the *anonymous variable*, denoted ‘\_’.

---

<sup>2</sup>These operations are also provided by the standard library, but they are named `hd` and `tl` instead.

```

car [X|_]      = X;
cdr [_|Xs]     = Xs;

```

The anonymous variable can only occur on the left-hand side of an equation, and may stand for a different value for each occurrence. In contrast, named variables, when repeated, stand for the same value. For instance, the following function can be used to remove adjacent duplicates from a list:

```

uniq []       = [];
uniq [X,X|Xs] = uniq [X|Xs];
uniq [X|Xs]   = [X|uniq Xs];

```

## 2.3 Iteration

Even today, most programmers grow up with imperative programming languages, and consequently they first feel at a complete loss when they have to do without predefined branch and loop control structures and mutable variables. Therefore in the following we show that it is in fact rather simple to realize typical looping constructs in a purely functional style, employing nothing but conditional and recursive equations. The key technique here is “tail-recursive” programming, which characterizes a special type of recursion which can be executed in constant memory space.

Let us take the Fibonacci function as a simple example. This function maps 0 to 0, 1 to 1, and then each integer  $n > 1$  to the sum of the Fibonacci values for  $n - 2$  and  $n - 1$ . A straightforward recursive definition of the function in Q looks as follows:

```

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

```

A good programmer immediately notices that this definition, albeit correct, is grossly inefficient: it takes an exponential number of computation steps which is unacceptable for larger inputs. Therefore the naive definition is usually rewritten to an iterative form, which keeps track of the two previous values of the function. In a conventional programming language like Pascal this algorithm could be implemented as follows:

```

function fib(N : integer);
var A, B, C: integer;
begin
  A := 0; B := 1;
  while N>0 do begin
    C := A+B; A := B; B := C;
    N := N-1;
  end;
  fib := A;
end;

```

But how do we accomplish the same in Q? There are no looping constructs and in fact we do not even have an assignment operation! The answer is that we can turn the local variables A and B, which store the last two values of the function, into parameters of a second, “auxiliary” function which performs the iteration. This can be done as follows:

```

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

```

This definition not only has the desired effect to cut down the running time to linear. It is also tail-recursive, that is, the recursive invocation of the `fib_loop` function is the last operation executed during an applicative order evaluation of the right-hand side of the second equation. The Q interpreter automatically optimizes such definitions and executes them in constant stack space. Thus the above function is indeed executed just as efficiently as the corresponding imperative program (up to constant factors).

Similar techniques are applicable to all kinds of iterative algorithms which can be implemented on a machine with a fixed set of registers [1]. You can even define your own generic control structures this way. It takes some time to get used to this style of programming, but once mastered it is very effective and modular.

## 2.4 Variable definitions

Following the standard terminology in mathematical logic, a variable occurring on the left-hand side of an equation is called a *bound* variable. Other variables occurring on the right-hand side of an equation are called *free*. In the Q language, free variables can be assigned a value by means of a *global variable definition* of the form `def L = R`, where *L* and *R* are arbitrary expressions. Note that `def` is a keyword, not a function. For instance, consider the following equation:

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

Since *C* occurs free on the right-hand side of the equation we have that `foo 9` reduces to the normal form `C*9`. But if we assign a value to the variable *C* as follows,

```
def C           = 5;
foo X           = C*X;
```

then `foo 9` evaluates to `5*9=45`.

The left-hand side of a variable definition can also be a structured expression to be matched against the right-hand side value. In this case all variables of the left-hand side will be set to the corresponding component values in the right-hand side. For instance,

```
def (X,Y)       = (16.3805*5, .05);
```

will set *X* to `16.3805*5=81.9025` and *Y* to `0.05`.

The Q language also provides *local variable definitions* which pertain to individual equations. These definitions are introduced with the keyword `where` at the end of an equation. For the sake of a practical example, let us define a function to solve quadratic equations in the normalized form  $x^2 + px + q = 0$ . The solutions to this equation are given by

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

where the value  $D = p^2/4 - q$  under the square root, the so-called *discriminant*, determines whether the equation has any real solutions. Thus our `solve` function first computes the discriminant and assigns it to the local variable `D`. It then checks whether this value is non-negative and, if so, returns the pair of solutions. All this can be accomplished with a single equation, as follows:

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

An equation may contain multiple conditions and/or **where** clauses. For instance, if we want to avoid the repeated calculation of the square root of the discriminant on the right-hand side of the above equation, we may assign that value to a second local variable `E`:

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

## 2.5 Data types

We have already seen that the `Q` language supports different types of expression values, such as truth values, integers, floating point numbers, strings, lists and tuples. These data types, as well as a few others, are built into the language. New data types can be introduced by means of a *type declaration*. The elements of such user-defined data types are (constant) function symbols or applications. These types are also called “algebraic” because they correspond to the (free) term algebra for the signature of “constructor” symbols given in the type declaration. For instance, the built-in type of truth values can be thought of as being predefined as follows (the `const` keyword in this declaration indicates that the members of the type, `true` and `false`, are to be considered as constant symbols):

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

As another example, here is how we can define a type consisting of the days of the week:

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

A type like the one above (as well as the built-in `Bool` type), whose members are all constant symbols without parameters, is also called an *enumeration type*. `Q` provides special support for enumeration types. In particular, the members of an enumeration type can be compared with `<` and `>`.

In general, a member of a user-defined type can also take the form of a function application. For instance, the following declaration introduces a type `BinTree` along with two constructor symbols: a parameterless (i.e., constant) symbol `nil` and the symbol `bin` which takes three arguments: a value `X`, and two subtrees `T1` and `T2`. Such a data type can be used to represent binary search trees.

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

A binary tree insertion operation, which keeps the tree sorted, could then be implemented as follows:

```

insert nil Y           = bin Y nil nil;
insert (bin X T1 T2) Y = bin X (insert T1 Y) T2 if X>Y;
                      = bin X T1 (insert T2 Y) if X<Y;
                      = bin Y T1 T2 otherwise;

```

Types can also be derived from each other, using single inheritance, which provides for object-oriented programming techniques a la Smalltalk or Java; see [3] for details.

Like Lisp and Prolog, but in difference to ML and Haskell, Q is a language with *dynamic typing*, that is, a variable can hold values of any type. In order to check which equations and built-in definitions are applicable to an expression, the interpreter keeps track of the actual type at runtime. It is possible to restrict an equation to variables of a given type, employing a *type guard* of the form *VAR:Type* on the left-hand side. For instance, if we want to ensure that the `sqr` function is only applied to numbers (integers or floating point values), we can invoke the built-in `Num` type as a guard on `sqr`'s argument:

```

sqr X:Num           = X*X;

```

## 2.6 Multithreading

Q also provides the necessary operations for realizing multithreaded scripts, i.e., scripts which run multiple evaluations concurrently. The usual thread management functions and synchronization facilities (mutexes, conditions and semaphores) from the POSIX threads library are also supported. For instance, when evaluating the `main` function from the following script, two threads will be started in parallel which each produce output on the terminal at random time intervals.

```

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

```

Note that in the above example `random` is a built-in function which returns a pseudo-random 32 bit integer, and `sleep` pauses for the specified interval in seconds (a random interval in the range between 0 and 1 in this case). The `||` operator in the first equation allows several expressions to be executed in sequence, so the `task` function first sleeps for a random time, then prints something on the terminal, and finally invokes itself again. Two different calls of this function are executed in parallel by the third equation, which uses the `thread` function to evaluate an expression in a new thread.

## 2.7 Memory Management and Exception Handling

Like most functional languages, Q provides automatic memory management. Using a reference counting scheme, expressions are garbage-collected as soon as they become inaccessible in the course of a computation. Other cleanup tasks on data structures involving side effects, such as closing files, are also performed automatically. The interpreter's memory management is completely dynamic; thus the heap and the evaluation stack are resized automatically as necessary. However, it is possible to restrict the amount of memory and stack space available to a program.

Because of its rewriting semantics, Q is mostly an exception-free language. For instance, an “error condition” like “division by zero” or other forms of argument mismatch will usually not generate a runtime error, but simply means that the corresponding expression is in normal form. Nevertheless, there are a few “hard” error conditions like memory overflow for which the interpreter generates an exception. Sometimes it is also useful to have a user program raise and handle exceptions; for this purpose the Q language provides the built-in `catch` and `throw` functions.

## 2.8 Scripts and modules

The basic compilation unit in the Q language is the “script”, which is simply a (possibly empty) sequence of declarations and definitions in a single source file. In order to make a given set of definitions available for use in the interpreter, one just collects the relevant declarations, variable definitions and equations in a single script which is then submitted to the interpreter for execution.

If we put all definitions into a single script, that’s all there is to it. However, one often wants to break down a larger script into a collection of smaller units, commonly called “modules”, which can be managed separately. In the Q language there is no distinction between the main script and the other modules, all these are just ordinary script files. To gain access to the function, variable and type symbols provided by another module we have to use an *import declaration*. For instance, to import the operations defined in the `midi.q` module, which provides the basic operations of the Q-Midi interface, we employ the following declaration:

```
import midi;
```

Note that if a script includes some functions, variables or types which are to be used by other scripts, the corresponding symbols have to be declared explicitly as “public”. For instance:

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

Each script has its separate namespace. Name collisions can be resolved either with “qualified” identifiers of the form `script::ident`, or by declaring aliases; see [3] for details. Undeclared symbols are implicitly declared as “private”, which means that they are accessible only in the script which defines them. Sometimes it is also necessary to explicitly declare a symbol as private, because a public symbol of the same name is already defined elsewhere.

The Q interpreter comes with a collection of predefined scripts called the *standard library*. These modules are always imported automatically. The standard library provides additional functions for list processing, an implementation of the lambda calculus, “container” data structures such as dictionaries and sets, interfaces to graphics and system functions, and much more; see [3] for details.

The Q language also has an elaborate C interface which allows external object modules to be loaded at runtime. This is useful to interface to existing software libraries, in order to create extensions for various problem domains. In fact, an important part of Q’s standard library (the system interface) is actually implemented in C. Other extensions are available, such as interfaces to John W. Eaton’s GNU Octave, IBM’s Data Explorer and John Ousterhout’s Tcl/Tk. The Q-Midi module discussed in this paper is implemented using Q’s C interface as well.

## 2.9 Running scripts with the interpreter

In order to run a script with the Q interpreter, one simply invokes the interpreter on the corresponding script file. For instance, having saved our definition of the `fac` function in the file `fac.q`, we run the interpreter using the shell command `q fac`. The interpreter then displays its sign-on message and leaves us at its command prompt, so we can start typing in the expressions to be evaluated:

```
==> fac 20
2432902008176640000
```

The interpreter can also be invoked without any script, in which case only the built-in and standard library operations are available. To exit the interpreter, one either enters the end-of-file character at the beginning of the command line or invokes the built-in `quit` function. The interpreter also provides a number of special commands which can be used, e.g., to edit and restart the current script, run another script, import additional modules, define and undefine global variables, or invoke a symbolic debugger. The complete reference manual is available with the `help` command, which invokes the GNU info reader. This command also accepts a keyword as argument. For instance, a complete list of all available commands can be obtained as follows:

```
==> help commands
```

Command line editing is provided via the GNU readline library. Moreover, the Q distribution includes an Emacs mode for editing Q scripts and running the interpreter within Emacs; see [3] for details. For MS Windows systems there also is a program called “Qpad”, an interactive graphical application for editing and running Q scripts.

While the interpreter has been designed to facilitate interactive usage, Q scripts can also be configured as standalone applications. To these ends, you can invoke a script and have it execute some `main` function immediately with the shell command `q -c main script`. Under Unix and Linux you can also employ the usual “shebang notation”. For instance, if you include the following as the first line in the main script and make the script file executable, the script can be run from the shell like any other program.

```
#!/usr/bin/q -c main
```

## 3 The Q-Midi interface

Grame’s MidiShare [2] has been under development since 1987. Originally being implemented on Apple Macintosh and Atari computers, the latest releases of the system also run on modern platforms such as Linux, Mac OS X and Windows. MidiShare is thus available for all major current desktop platforms, which makes it an ideal basis for portable MIDI interface modules in high-level languages. Work on the Q-Midi module started in 2002; the present version was finished in March 2003 and includes all essential MidiShare functionality, including MidiShare client setup and driver management, MIDI timing and input/output functionality, and MIDI file access. Hence Q-Midi provides a complete programming interface for developing MIDI applications in the Q programming language, which is available for Linux, OS X and Windows systems.

In the following we only describe the main features of the interface. A complete description of all operations can be found in the `midi.q` script. To work with the interface in a Q script, the script must import this module using an appropriate import declaration as described in section 2.8. Besides the `midi.q` script, Q-Midi also includes a second script `mididev.q` which implements a portable “device table”, to accommodate some incompatibilities between the current versions of MidiShare under Linux and Windows. Thus a Q-Midi application typically starts with an import declaration like the following:

```
import midi, mididev;
```

### 3.1 MIDI message type

For convenience, Q-Midi encodes MIDI messages not as raw byte sequences, but as members of an algebraic data type `MidiMsg` which is declared at the beginning of the `midi.q` script (see fig. 3). This facilitates the manipulation of MIDI messages using equations. For instance, “note on” messages are encoded using a constructor `note_on` which has three arguments: a channel number (0-15), a MIDI note number (0-127) and a velocity value (0-127). Thus a function for transposing a note on message by a given (positive or negative) amount of semitones can be implemented simply as follows:

```
transp N (note_on C P V)      = note_on C (P+N) V;
```

The `MidiMsg` type supports all standard MIDI messages (including the “meta” messages which are only used in MIDI files) and some MidiShare-specific extensions like the `note` message which allows to specify a note with a given duration. (This kind of message is automatically translated to a corresponding pair of note on and off messages when output to a MIDI device.) Besides the `MidiMsg` type itself, the module also provides various functions for determining the category of a message (`is_voice`, `is_meta`, etc.), and to check for specific messages (`is_note_on`, `is_note_off`, ...). These are useful, e.g., for filtering lists of MIDI messages.

### 3.2 Setting up clients and connections

MidiShare takes a somewhat unusual approach in that application programs do not input from or output to MIDI devices directly. Instead, an application program registers one or more MidiShare *clients* and connects them to each other.<sup>3</sup> Thus the program actually establishes a graph of client connections through which all MIDI messages flow. Physical MIDI devices are represented by special predefined clients as well, which act as sources and sinks in the client graph. An application may receive and transmit MIDI messages on any of its registered clients. This approach is very flexible, because it allows different parts of the application to be connected to each other dynamically, with MidiShare handling all the data flow automatically and transparently.

Following this approach, a Q-Midi application first has to register one or more MidiShare clients with the `midi_open` function which takes the desired client name (a string) as its single argument, and returns a client *reference* number. The reference number can then be used in

---

<sup>3</sup>Note that the “clients” are actually called “applications” in the MidiShare documentation. We chose the term “clients” here because we use “application” to refer to the entire application program which may define multiple MidiShare clients.

```

public type MidiMsg = const

  /* voice messages */
  note_on CHAN /* 0..15 */ PITCH /* 0..127 */ VEL /* 0..127 */,
  note_off CHAN /* 0..15 */ PITCH /* 0..127 */ VEL /* 0..127 */,
  key_press CHAN /* 0..15 */ PITCH /* 0..127 */ VAL /* 0..127 */,
  ctrl_change CHAN /* 0..15 */ CTRL /* 0..127 */ VAL /* 0..127 */,
  prog_change CHAN /* 0..15 */ PROG /* 0..127 */,
  chan_press CHAN /* 0..15 */ VAL /* 0..127 */,
  pitch_wheel CHAN /* 0..15 */ LSB /* 0..127 */ MSB /* 0..127 */,

  /* system common */
  sysex BYTES /* byte sequence, encoded as a list */,
  quarter_frame TYPE /* 0..7 */ COUNT /* 0..15 */,
  song_pos LSB /* 0..127 */ MSB /* 0..127 */,
  song_sel NUM /* 0..127 */,
  tune,

  /* system realtime */
  clock,
  start,
  stop,
  continue,
  active_sense,
  reset,

  /* meta-events */
  seq_num NUM /* 16 bit */,
  text TEXT /* string */,
  copyright TEXT,
  seq_name TEXT,
  instr_name TEXT,
  lyric TEXT,
  marker TEXT,
  cue_point TEXT,
  chan_prefix CHAN /* 0..15 */,
  port_prefix PORT /* 0..127 */,
  end_track,
  tempo TEMPO /* 24 bit */,
  smpte_offset SECS /* 20 bit */ FRAMES /* 12 bits */,
  time_sign NUM DENOM CLICK QUARTER_DEF,
  /* NUM = numerator, DENOM = log2 of denominator, CLICK = MIDI clocks per
  metronome click (24 clocks = 1 quarter note), QUARTER_DEF = number of
  32ths in a quarter note */
  key_sign SIGN /* -7=7 flats..+7=7 sharps */ KEY /* 0=major, 1=minor */,
  specific BYTES /* byte sequence */,

  /* MidiShare-specific */
  note CHAN /* 0..15 */ PITCH /* 0..127 */ VEL /* 0..127 */ DUR /* 15 bit */,
  ctrl14b_change CHAN /* 0..15 */ CTRL /* 0..31 */ VAL /* 14 bit */,
  non_reg_param CHAN /* 0..15 */ PARAM /* 14 bit */ VAL /* 14 bit */,
  reg_param CHAN /* 0..15 */ PARAM /* 14 bit */ VAL /* 14 bit */,
  midi_stream BYTES /* byte sequence */,

  /* unsupported (internal) MidiShare messages */
  unknown;

```

Figure 3: The MidiMsg type.

establishing connections between the clients with the `midi_connect` function, and is also used to refer to a client when performing MIDI input and output. Therefore the client number has to be stored somewhere, typically in a global variable. The connections can be defined at initialization time as well. A typical setup for a simple application, which only uses a single client which is connected to the MIDI input and output devices, might then look as follows:

```
def REF = midi_open "My application",
  _ = midi_connect IN REF || midi_connect REF OUT;
```

Here we used a definition involving the anonymous variable to establish the connections, since we do not care for the return values of the `midi_connect` calls. Also note that the `IN` and `OUT` variables are assumed to give the reference numbers of the clients representing the MIDI input and output devices. We discuss how to determine these clients in the following section.

### 3.3 MIDI devices

Unfortunately, currently (as of MidiShare 1.86) there is an incompatibility between MidiShare's handling of MIDI devices on Linux and other operating systems. On Windows, as well as OS X, all MIDI devices are accessed via the predefined MidiShare client with reference number 0. Different devices are distinguished by means of logical *port* numbers which can be associated freely with the physical MIDI devices available on the system. Q-Midi provides the necessary operations to determine which MIDI device drivers and ports are available, and to inspect and change the port assignments. These facilities have not yet been implemented in the Linux version. Instead, the Linux distribution of MidiShare provides different driver clients, which are started as separate programs, for each type of interface. The port numbers are irrelevant here and can always be set to zero.

This incompatibility is expected to be rectified in a future Linux version of MidiShare. For the time being, Q-Midi provides the `mididev.q` script which implements a table of portable device descriptions, to facilitate the development of Q-Midi applications which work on all supported platforms. The `mididev.q` script provides transparent access to the Linux driver clients (starting them up at program initialization time if necessary) and a table of standard device descriptions. The device table, available through the global variable `MIDIDEV`, is implemented as a list; each element of the list is a triple specifying a symbolic name for the device (as a string) and the corresponding MidiShare client and port numbers. Under Windows and OS X, the device table is initialized from MidiShare's actual port assignments. Under Linux, three standard device entries are provided: `MIDIDEV!0` (the external MIDI interface), `MIDIDEV!1` (an internal software synthesizer) and `MIDIDEV!2` (the network). To make applications portable to all platforms, we define the first three ports on OS X and Windows in a manner which matches these specifications. Using `mididev.q`, we can then set up the MIDI devices in a Q-Midi script, e.g., as follows:

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

These definitions, which must precede the lines for setting up the connections discussed in the preceding section, assign the client number for the external MIDI interface to the `IN` variable, and the client and port numbers for the internal MIDI synthesizer to the variables `OUT` and `PORT`. (If *both* input and output is to go to the external MIDI interface then `MIDIDEV!1` must be replaced with `MIDIDEV!0` in the second definition.)

### 3.4 MIDI messaging and timing

Once the clients have been initialized, as explained in the previous sections, Q-Midi is ready to perform MIDI input and output. For this purpose, two functions are used, `midi_get` and `midi_send`. Two additional routines are provided for timing purposes: `midi_time` and `midi_wait`.

MidiShare maintains an internal 32 bit time value which counts off the time in milliseconds which elapsed since MidiShare was last activated. The current value of this counter can be accessed with the parameterless `midi_time` function. A client can also wait for a specific time to arrive with the `midi_wait` function, which suspends the program until the given absolute time and then returns that time value. (The time argument of `midi_wait` is taken modulo `0x100000000` automatically, so you do not have to worry about wrapover at the end of the time range.) Applications typically use this function to perform scheduling of MIDI events.

A call of the form `midi_send REF PORT MSG` is used to send a MIDI message `MSG` (of type `MidiMsg`) immediately, where `REF` denotes the sending client and `PORT` the destination port, as given by the `MIDIDEV` table. Instead of a simple `MSG` value, one can also specify a `(TIME,MSG)` pair to send a message at the specified time value (which is again taken modulo `0x100000000`). The message will then be sent by MidiShare when the prescribed time arrives, i.e., when the current `midi_time` becomes equal to `TIME`.

Incoming MIDI messages are received with the `midi_get` function which is invoked with the reference number of a client as its single argument, and returns a tuple `(REF,PORT,TIME,MSG)` denoting the reference number of the client the message was received from, the port of the message (which will be the source port if the message was received from a device client, and the destination port specified with `midi_send` if the message was received from a regular client), the time when the message was received, and the message (of type `MidiMsg`) itself. Unlike its C counterpart in the MidiShare library, this function is blocking, i.e., it waits for a message to become available. MidiShare also maintains an input queue so that incoming messages are not lost while the application program is busy doing other processing. One can check beforehand whether a message is currently available with the `midi_avail` function, and empty a client's input queue with the `midi_flush` function.

Note that when implementing a loop for processing MIDI events in realtime (such as a sequencer) it is often essential that timing is performed very accurately and MIDI events are always processed immediately. This can only be guaranteed when the thread executing the realtime loop runs at a high priority, which can be achieved with the `setsched` standard library function. Corresponding examples can be found in section 4.

### 3.5 Message filtering

These functions allow you to determine the messages a client wants to receive with `midi_get` (see above). By default, all ports, channels and message types are accepted. The functions `midi_accept_port`, `midi_accept_chan` and `midi_accept_type` can be used to restrict the set of messages which are received by a client, on the basis of the source/destination port, channel, and type of a message, respectively. Message types are specified using the corresponding constructor symbol. For instance, including the following initializations in a Q-Midi script will cause all `clock` and `active_sense` messages (which are emitted by some MIDI devices in regular time intervals for synchronization purposes) to be filtered out from the client `REF`, so that the application never sees them:

```
def _ = do (flip (midi_accept_type REF) false) [clock,active_sense];
```

Some additional functions are provided to check the current filter states and it is also possible to turn off all filtering again with the `midi_accept_all` function.

### 3.6 MIDI files

Q-Midi also provides a collection of functions to let you read and write MIDI files. MIDI files are represented using the `MidiFile` type. Objects of this type are returned by the functions `midi_file_open`, `midi_file_append` and `midi_file_create`, which open a file for reading, append to an existing file and create a new file, respectively. Each of these functions takes the name of the file to be opened as its (first) argument. A `MidiFile` object can be closed with the `midi_file_close` function (this operation is also performed automatically when a `MidiFile` object is garbage-collected).

MIDI files can have one of three formats, 0 = single track, 1 = multiple tracks/single song, 2 = multiple tracks/multiple songs. By definition, a format 0 file consists of a single track, into which all MIDI events go. A format 1 file (the most common format used by sequencers) contains several tracks forming a single piece; the tempo map of the piece is to be found in the first track. A format 2 file has several independent tracks which are to be played separately; in this case each track contains its own tempo map.

The MIDI file header information also describes how many tracks the file contains and which “division” the timestamps of events in the file are based upon. The division can be either a number `PPQN` denoting pulses per quarter note, or a pair `(FPS,TICKS)` specifying an SMPTE division consisting of a number of frames per seconds (24, 25, 29 or 30) together with the number of subdivisions of each frame.

The MIDI file header information can be determined with the functions `midi_file_format`, `midi_file_division` and `midi_file_num_tracks`. Moreover, the “current” time of the file (time of the last event read or written) can be retrieved with the `midi_file_time` function, and you can check whether the file has been opened for reading or writing with the `midi_file_mode` function.

For both reading and writing, the events in each track are represented as `(TIME,MSG)` pairs, where `TIME` is the time of the event and `MSG` is the MIDI message of type `MidiMsg`. Tracks are represented as event lists. Note that the time values are always absolute (zero-based) timestamps, whereas they are actually stored as “deltas” (time differences w.r.t. the previous event) in a MIDI file. Moreover, the timestamps are measured in “ticks”, i.e., in the unit specified by the division of the file. Hence one has to use the division value and the tempo map of the piece to transform these values to milliseconds, before one can play back the sequence. (We show how to do this in section 4.)

The tracks in a MIDI file opened for reading can be accessed in sequence, either by reading whole tracks at once with `midi_file_read_track`, or by explicitly opening a track with `midi_open_track` and then reading single events with `midi_file_read`. When you are finished with a track you can close it with the `midi_file_close_track` function (this also happens automatically when reading past the end of a track). Moreover, when a MIDI file has been opened for reading, you can position the file pointer at an arbitrary track with `midi_file_choose_track`, which takes a zero-based track index as argument.

Creating a new or appending to an existing MIDI file works analogously. A new track is written at the end of the file with the `midi_file_write_track` function. You can also

use `midi_file_new_track`, `midi_file_write` and `midi_file_close_track` to write a track event by event. Note that random track access is only supported while reading a file. When writing to a MIDI file, new tracks can only be added at the end of the file.

## 4 Examples

The examples in this section are not particularly advanced, but not trivial either. Their sole purpose is to give an idea of how programming with Q-Midi looks like. They also demonstrate the use of some of Q's functional programming capabilities which allow many algorithms to be expressed in a very concise way. (We do not claim, however, that the algorithms discussed below are the prettiest or most efficient possible, nor that they cover all intricacies of practical applications.)

### 4.1 Realtime processing

Processing MIDI messages in realtime means that the program responds to each incoming message immediately. This can be done with a loop which invokes a given function on each message. The loop is terminated when a `stop` message is received from the MIDI keyboard. For instance, the following script can be used to transpose note messages in realtime:

```
import midi, mididev;

/* MIDI interface */

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!1,
    REF = midi_open "Transpose",
    _ = midi_connect IN REF || midi_connect REF OUT;

private recv, send;
recv          = midi_get REF;
send          = midi_send REF PORT;

/* generic input loop */

process F          = midi_flush REF || loop F recv;

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

/* transpose notes and aftertouch; echo other messages */

transp N (note_on C P V)      = send (note_on C (P+N) V);
transp N (note_off C P V)    = send (note_off C (P+N) V);
transp N (key_press C P V)    = send (key_press C (P+N) V);
transp N MSG                  = send MSG otherwise;

/* main function */
```

```
public transpose N;
```

```
transpose N = process (transp N);
```

## 4.2 Realtime issues

As already mentioned, on some systems it may be necessary to run realtime loops such as the one above in a high priority thread, in order to get accurate timing and ensure immediate responses to incoming MIDI messages. The necessary modifications to accomplish this are a little tedious, but quite straightforward. The following function may be used to evaluate an expression under realtime scheduling. (This function is declared as a “special form” to defer evaluation of the argument expression until the realtime priority has been set.)

```
public special realtime X;
realtime X = setsched TH 0 0 || Y
  where TH = this_thread, Y = setsched TH POL PRIO || X;
```

Note that the `setsched` function is used to set the scheduling policy and priority of the current thread. Then the given expression is evaluated and the scheduling policy is reset to the default before the result of the evaluated expression is returned. A reasonable value for the scheduling policy `POL` is `SCHED_RR` (“round robin” scheduling). The maximum allowed thread priority `PRIO` is system-dependent. We can determine these values at startup time as follows:

```
def POL = SCHED_RR, PRIO = maxprio;

testprio PRIO = setsched this_thread 0 0 || true
  where () = setsched this_thread POL PRIO;
= false otherwise;

maxprio = until (neg testprio) (+1) 1 - 1;
```

We can put these definitions in a separate script, say `realtime.q`, which is imported in the programs which need them. A call to `realtime` is then inserted wherever a part of our application has to be executed in realtime. For instance, we would modify the main function of the previous transposition program as follows:

```
transpose N = realtime (process (transp N));
```

## 4.3 Sequencing

A particularly important type of realtime application is MIDI sequencing, i.e., recording and playing back MIDI sequences. For this purpose, we can represent the sequences as lists of MIDI events, where each event is encoded as a pair consisting of an absolute time value and a MIDI message. The list is ordered w.r.t. the timestamps. For instance, the beginning of a piece starting with an arpeggiated C major chord on channel 0 might look as follows:

```
[(0,note_on 0 60 64),(50,note_on 0 64 64),(100,note_on 0 67 64)]
```

Recording sequences can be done with a loop similar to the one in the preceding section. However, here we do not transform the events, but echo them unchanged. Moreover, the loop is invoked with an additional list argument (initially the empty list) in which we collect the received messages. This list is returned when the recording is terminated with a `stop` message.

```
import midi, mididev, realtime;

/* MIDI interface */

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!1,
    REF = midi_open "Record",
    _ = midi_connect IN REF || midi_connect REF OUT;

private recv, send;
recv          = midi_get REF;
send         = midi_send REF PORT;

/* recording loop */

recloop SEQ (_,_,_,stop) = SEQ;
recloop SEQ (_,_,T,MSG) = send MSG ||
                           recloop (append SEQ (T,MSG)) recv
                           otherwise;

/* main function */

public record;

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

Playing back a recorded sequence can be done with the following algorithm which uses the `midi_wait` function to wait until the next event in the sequence is due. Here we have to distinguish between the timestamps of events in the sequence and the actual time at which the playback is performed. The playback loop keeps track of both the sequence time of the last event emitted, and the actual time when the event was output. As long as the sequence time of the current event matches the sequence time of the previous one, we simply send the event and proceed with the rest of the list. Otherwise we compute the real time at which the current event is due and wait until that time arrives.

```
import midi, mididev, realtime;

/* MIDI interface */

def (_,IN,_) = MIDIDEV!0, (_,OUT,PORT) = MIDIDEV!1,
    REF = midi_open "Play",
    _ = midi_connect IN REF || midi_connect REF OUT;
```

```

private recv, send;
recv          = midi_get REF;
send          = midi_send REF PORT;

/* playback loop */

playloop S T [] = ();
playloop S T [(S,MSG)|SEQ] = send 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);

/* compute time differences, taking into account possible wrapover */

def MAX = 0x100000000;

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

/* main function */

public play SEQ;

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

```

In a practical application one would also have to combine recording and playback. This is not much more complicated than a combination of the two programs above, but we also have to synchronize the time values between recording and playback. Moreover, Q's multithreading functions would be used to execute the recording and the playback loop in parallel.

#### 4.4 Loading a MIDI file

In most applications we will also have to deal with MIDI files. As an example, in this section we discuss a simple method for loading the tracks of a type 0 or 1 MIDI file, mix down the tracks to a single sequence of MIDI events, and convert the timestamps to milliseconds. The resulting sequence is then ready to be processed with the `play` function from the preceding section.

Let us first discuss how to mix two sequences, each of which is already ordered by timestamps, to a single ordered sequence. This can be done with a method which is the basic ingredient of the well-known mergesort algorithm. The basic idea behind this method is that in each step the next event of the output sequence comes from the input sequence which has the smallest timestamp at its beginning. We then remove this event from the input sequence and add it to the output sequence. This process is repeated until both input sequences have become empty. The algorithm can be expressed in Q as a recursive function as follows:

```

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;

```

The second main ingredient of our program is a method to convert the timestamps in the mixed sequence to milliseconds, given the division of the file. In the case of SMPTE timestamps this is merely a matter of multiplying the timestamps with an appropriate factor:

```

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

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

```

Here we have employed one of Q's generic list functions, `map`, which applies a function to every member of a list.

For musical timestamps, given in pulses per quarter note, this becomes a bit trickier, since we also have to take into account the tempo of the piece which may change over time. We thus have to keep track of the current tempo which is given by the `tempo` meta events in the sequence. Note that the `tempo` message specifies the tempo in microseconds per quarter note. If no `tempo` message is given, the tempo defaults to 120 BPM, which equals 500,000 microseconds per quarter. Hence we can do the conversion as follows:

```

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

convert_ppqn _ _ []      = [];

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

update_tempo (tempo TEMPO) _ = TEMPO;
update_tempo _ TEMPO        = TEMPO otherwise;

```

Note that the `update_tempo` function is used to update the current tempo value when we encountered a `tempo` message. With the `(TEMPO,S,T)` argument of the `convert_ppqn` function we keep track of this value as well as the current sequence time (in ticks) and the corresponding physical time (in milliseconds).

The rest of the program can now be implemented using the basic MIDI file functions discussed in section 3.6. We also employ another generic list function, `foldl`, which iterates a binary function over a list, starting from a given initial value. This is used to iterate the `mix` function discussed above over all tracks in the file.

```

/* the main function */

```

```

public load NAME;

```

```

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

/* read the tracks (produces a list of sequences) */

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

load_track F _                           = midi_file_read_track F;

```

## 4.5 Algorithmic composition

The final example we consider comes from the realm of *algorithmic composition*. It is a kind of pattern-based “sequencing technique” which was proposed by the German composer and music theorist Johann Philipp Kirnberger in the 1750s, long before anything resembling a computer existed. The *Kirnberger polonaises* are a kind of the “musical games of dice” which became very popular in Europe during the second half of the eighteenth century. Kirnberger probably was not the original inventor of these games, but apparently the first one to publish a paper about this technique. More information about the Kirnberger polonaises and a facsimile of Kirnberger’s original publication can be found in [5].

A Kirnberger polonaise is made up of two parts *A* and *B* consisting of 6 and 8 bars, respectively. The piece then follows the scheme *AABA'*, where *A'* denotes a repetition of the last 4 bars of part *A*. The main part of Kirnberger’s paper from 1757 is a score containing the bars from which the polonaises are put together, in which the bars are numbered consecutively. A table specifies the indices of bars which can be used for each of the bars in part *A* and *B* of the piece. Here we only consider the basic form of Kirnberger’s game in which a single die is used. Hence the table we use specifies six bar numbers for each bar in part *A* and *B*, i.e.,  $6 \times (6 + 8) = 84$  table entries in total. The table is shown in fig. 4.

Putting together a Kirnberger polonaise now proceeds as follows. We throw the die once for each of the bars in part *A* and *B*, i.e., 14 times in total. For each throw we write down the bar number in the corresponding row of the table, i.e., if the die shows value  $j \in \{1, \dots, 6\}$  in the  $i$ th throw ( $i = 1, \dots, 14$ ) we take the bar number in column  $j$  of the  $i$ th row of the table. For instance, if the die yields the sequence 2-5-3-5-5-6 4-1-6-2-2-4-2-5 then we obtain the bars 10-56-60-79-48-64 for part *A* and bars 43-11-7-5-27-49-15-75 for part *B* of the piece.

| Part A |    |    |    |    |    |    | Part B |    |    |    |    |    |    |
|--------|----|----|----|----|----|----|--------|----|----|----|----|----|----|
|        | 1  | 2  | 3  | 4  | 5  | 6  |        | 1  | 2  | 3  | 4  | 5  | 6  |
| 1      | 70 | 10 | 42 | 62 | 44 | 72 | 7      | 80 | 20 | 82 | 43 | 78 | 69 |
| 2      | 34 | 24 | 6  | 8  | 56 | 30 | 8      | 11 | 77 | 3  | 41 | 84 | 63 |
| 3      | 68 | 50 | 60 | 36 | 40 | 4  | 9      | 59 | 65 | 9  | 45 | 29 | 7  |
| 4      | 18 | 46 | 2  | 12 | 79 | 28 | 10     | 35 | 5  | 83 | 17 | 76 | 47 |
| 5      | 32 | 14 | 52 | 16 | 48 | 22 | 11     | 74 | 27 | 67 | 37 | 61 | 19 |
| 6      | 58 | 26 | 66 | 38 | 54 | 64 | 12     | 13 | 71 | 1  | 49 | 57 | 31 |
|        |    |    |    |    |    |    | 13     | 21 | 15 | 53 | 73 | 51 | 81 |
|        |    |    |    |    |    |    | 14     | 33 | 39 | 25 | 23 | 75 | 55 |

Figure 4: Kirnberger’s bar table.

Translating this method to a Q script is fairly straightforward. In the following we assume that the individual bars are stored in MIDI files `01.mid`, `02.mid`, ..., `84.mid` in a subdirectory called `midi`. Using the `load` function from the previous section we can then read in a list with the individual bars as follows:

```
def M = map load (glob "midi/*.mid");
```

Here we employed the standard library function `glob` which returns a list of all filenames matching the given pattern. Next we define a second global variable `T` which contains Kirnberger's table, encoded as a list of lists. Note that we actually decrement the bar numbers by 1 (this is accomplished with the `map (map pred)` construct), since they will be used as indices into the list `M` of MIDI bars (in Q list indices are zero-based).

```
def T = map (map pred)
  [// part A
    [ 70, 10, 42, 62, 44, 72],
    [ 34, 24, 6, 8, 56, 30],
    [ 68, 50, 60, 36, 40, 4],
    [ 18, 46, 2, 12, 79, 28],
    [ 32, 14, 52, 16, 48, 22],
    [ 58, 26, 66, 38, 54, 64],
  // part B
    [ 80, 20, 82, 43, 78, 69],
    [ 11, 77, 3, 41, 84, 63],
    [ 59, 65, 9, 45, 29, 7],
    [ 35, 5, 83, 17, 76, 47],
    [ 74, 27, 67, 37, 61, 19],
    [ 13, 71, 1, 49, 57, 31],
    [ 21, 15, 53, 73, 51, 81],
    [ 33, 39, 25, 23, 75, 55]];
```

Now let us consider how to “throw the dice”. For this purpose we use Q's built-in `random` function, which yields a pseudo random 32 bit integer, and map this integer to the range from 0 to 5. (This is again because the values will be used as indices into a list, the rows of the table `T` in this case.)

```
dice N = listof die (I in nums 1 N);
die = random div 1000 mod 6;
```

This definition shows another predefined list function at work, namely `listof` which allows lists of values to be constructed using “list comprehensions”, similar to the way in which sets are described in mathematics. In this case a list is constructed from `N` different random values.

The central part of our algorithm is the following function which puts together a Kirnberger polonaise for a given list of die values. This is achieved by mapping the index operator to pairs of corresponding table rows and die values with the `zipwith` function. We then extract parts `A` and `B` of the piece and construct part `A'` by dropping the first two bars from part `A`. Finally, the parts are concatenated with the list concatenation operator `++` and all

bars are combined to a single piece by iterating the `seq` function over the resulting list of MIDI bars.

```
polonaise D                = foldl seq [] (A++A++B++A1)
  where P = map (M!) (zipwith (!) T D),
        A = take 6 P, B = drop 6 P, A1 = drop 2 A;
```

The function `seq` combines two MIDI fragments in such a manner that they are played in succession. To these ends, we have to translate the absolute timestamps in the second sequence accordingly. This is achieved by shifting the timestamps in the second sequence by the amount given by the timestamp of the last event in the first sequence.

```
seq S1 S2                  = S2 if null S1;
                           = S1++map (shift DT) S2 where (DT,_) = last S1;
shift DT (T,MSG)           = (T+DT,MSG);
```

Voilà! The main function now simply generates a piece using the `polonaise` function and plays it back with the `play` function from section 4.3:

```
kirnberger                 = play (polonaise (dice 14));
```

## 5 Conclusion

The development of complex computer music applications will probably never be a simple business. Nevertheless, as we tried to point out in this paper, MIDI programming can be facilitated considerably by a high-level functional programming language. Q-Midi provides an advanced interface for this kind of programming, in which MIDI messages are represented as abstract elements of an algebraic data type. To our knowledge, none of the other “modern style” functional languages has a usable MIDI interface yet, so Q-Midi probably is the first attempt at providing such a facility. (For Haskell there is a library called “Haskore” [4] for dealing with musical structures which can be mapped to MIDI files, but it does not provide direct MIDI input/output.)

Of course, there are other projects which apply functional programming for developing computer music applications. For instance, Rick Taube at the CCRMA is the author of “Common Music”, a comprehensive algorithmic composition package written in Lisp [9]. It must be mentioned, however, that this software currently does not provide the degree of MIDI interoperability required to implement realtime applications. Another interesting approach is taken by Grame’s “Elody” system [6], which provides a graphical programming language based on the lambda calculus for formulating compositional rules from which musical pieces can be generated and performed, also in realtime. As such it is a specialized functional language, designed to facilitate the development of specific types of computer music applications, but no general MIDI programming environment.

So we think that Q-Midi provides a viable alternative for those who would like to explore MIDI programming in a non-imperative programming language. Future work on Q-Midi will be concerned with higher-level abstractions needed for developing advanced computer music applications and porting important computer music algorithms to Q. Moreover, we are currently investigating alternatives for equipping Q with a digital audio interface, which is the

second major cornerstone of a comprehensive computer music system. In particular, we are planning to provide an interface between Q and James McCartney’s well-known “SuperCollider” sound synthesis software [7].

## References

- [1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996. See <http://mitpress.mit.edu/sicp>.
- [2] D. Fober, S. Letz, and Y. Orlarey. MidiShare joins the open sources softwares. In *Proceedings of the International Computer Music Conference*, pages 311–313, International Computer Music Association, 1999. See also <http://www.grame.fr/MidiShare>.
- [3] A. Gräf. The Q Programming Language. *Musikinformatik & Medientechnik* 6, Johannes Gutenberg University, Mainz, 3rd edition, 2002.
- [4] P. Hudak. Haskore Music Tutorial. Yale University, Department of Computer Science, 2000. See <http://www.haskell.org/haskore>.
- [5] H. Kupper. Der allezeit fertige Polonoisen- und Menuettencomponist von Johann Philipp Kirnberger. Mit einer Einleitung von Hubert Kupper. *Musikinformatik & Medientechnik* 15, Johannes Gutenberg University, Mainz, 1995.
- [6] S. Letz, D. Fober, and Y. Orlarey. Realtime composition in Elody. In *Proceedings of the International Computer Music Conference*, International Computer Music Association, 2000. See also <http://www.grame.fr/Elody>.
- [7] J. McCartney. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002. See also <http://www.audiosynth.com>.
- [8] The Complete MIDI 1.0 Detailed Specification. MIDI Manufacturers Association, 2001. See <http://www.midi.org>.
- [9] H. Taube. Common Music: A music composition language in Common Lisp and CLOS. Technical Report STAN-M-63, Center for Computer Research in Music and Acoustics, Stanford University, 1989. See also <http://ccrma-www.stanford.edu/software>.