The Q Programming Language

Version 4.5
14 October 2003

by Albert Gräf
Johannes Gutenberg-University Mainz

This document describes version 4.5 of the Q programming language and system.

This manual has been published in the series *Musikinformatik und Medientechnik*, Bereich Musikinformatik, Musikwissenschaftliches Institut, Johannes Gutenberg-Universität Mainz

P.O. Box 3980
55099 Mainz
Germany
ISSN 0941-0309

1 Introduction

Q stands for "equational", so Q, in a nutshell, is a programming language which lets you "program by equations". You specify a system of equations which the interpreter uses as "rewrite rules" to reduce expressions to "normal form". This allows you to formulate your programs in a high-level, concise and declarative style. Q's development started out in the 1990s, motivated by the author's research on term pattern matching [Gräf 1991] and inspired by the pioneering work on equational programming by Michael O'Donnell and others [O'Donnell 1985], as an attempt to show that term rewriting would provide a feasible basis for a practical programming language. I think that this goal has been achieved, and that the present interpreter is efficient and robust enough for practical purposes. Q has been ported to a variety of operating systems, such as BeOS, FreeBSD, Linux, MacOS X, Solaris and Windows. Porting to other modern UNIX/POSIX environments should be a piece of cake. Thus Q can be used on most modern computer systems, and Q scripts written on one system will usually run on all supported platforms.

The Q language supports a rich variety of built-in types, like arbitrary precision integers, floating point numbers (double precision 64 bit), truth values, strings, lists and files. It also provides primitives for exception handling and multithreaded execution. Q scripts can be broken down into "modules", each with their separate namespace, and Q gives you full control over which symbols (function, variable and type names) are imported and exported by each module. This makes it easy to organize large scripts in a modular fashion. Q also allows you to interface to "external" modules written in the C programming language, which provides a means to access functions in C libraries and employ C's higher processing speed for time-critical tasks.

As a practical programming language, Q comes with "batteries included": A comprehensive standard library, written mostly in Q itself, provides a complex number type, a lot of useful list processing functions (including list comprehensions), a "lazy" (call-by-need) list data structure (a.k.a. "streams"), many common container data structures (dictionaries, sets, etc.), the lambda calculus, and operations for creating PostScript graphics. It also includes an extensive system interface which offers services such as binary and C-style formatted I/O, BSD socket I/O, process management, POSIX multithreading and regular expression matching. Additional extension modules for interfacing to GNU Octave, Tcl/Tk, GGI, IBM's Data Explorer and ODBC-compatible databases make Q a valuable tool for scientific programming and other advanced applications.

In difference to other functional languages, Q is entirely based on the notions of *rewrite rules*, *reductions* and *irreducible expressions* (also known as *normal forms*) pertaining to the *term rewriting* calculus. A Q "program", called a *script*, consists of equations which are treated as rewrite rules and are used to reduce expressions to normal form. The normal form of an expression denotes its value, which can itself be a compound expression. Q has no rigid distinction between "constructor" and "defined" function symbols and it also allows you to evaluate expressions containing "free" variables. Basically, both sides of an equation may involve arbitrary expressions. Therefore Q can also be used as a tool for symbolic expression evaluation. Q's symbolic processing capabilities are best illustrated by the fact that advanced features such as the lambda calculus and list comprehensions are *not* built into the language, but are provided by scripts written in the Q language.

On the surface, Q looks very much like contemporary functional languages such as Miranda or Haskell. In fact, the syntax of the language has largely been inspired by the book *Introduction to Functional Programming* by Richard Bird and Philip Wadler. However, Q is an interpreted language with dynamic typing and eager (leftmost-innermost) evaluation, which is more in line with classical functional languages such as Lisp. For the sake of efficiency, Q scripts are first translated into "bytecode" (an intermediate binary format) which is executed on a virtual stack machine. The interpreter automatically optimizes tail recursion, such that "iterative" algorithms can be realized in constant stack space. Besides the built-in I/O operations, the language is free of side-effects; in particular, Q does not have mutable variables. Q also provides two novel and (IMHO) interesting features: a notion of *special forms* which allows to handle lazy evaluation in an (almost) transparent manner without having to give up the basic eager evaluation strategy; and a notion of *type guards* which provides a means to cope with hierarchies of abstract data types (similar to the notion of classes with single inheritance in object-oriented languages) in the context of a term rewriting language.

Using Q is supposed to be fairly simple: you throw together some equations, start the interpreter and then type in the expressions you wish to evaluate. All this can be done with a few keystrokes, if you use the Emacs Q mode supplied with the package. A graphical user interface for Windows is also available. Of course, you can also run the Q programming utilities from the command line if you prefer that.

The manual is organized as follows. In Chapter 2 [Getting Started], page 5, we start out with a brief and informal introduction to the main features of the language, and a glimpse of how Q scripts look like. Chapter 3 [Lexical Matters], page 19, describes the lexical part of the Q language. In Chapter 4 [Scripts and Modules], page 23, we discuss how declarations, definitions and equations are put together to form a script, and how to break down larger scripts into a collection of smaller modules which can be managed separately. Chapter 5 [Declarations], page 25, discusses how certain entities like types, variables and function symbols can be declared in a Q script. Chapter 6 [Expressions], page 29, treats the syntax of Q expressions, and describes the built-in operators provided by the Q language. Chapter 7 [Equations and Expression Evaluation], page 41, is about equations and variable definitions, and how they are used in the evaluation process. Chapter 8 [Types], page 61, and Chapter 9 [Special Forms], page 69, describe the facilities provided by the Q language for dealing with abstract data types and deferred evaluation. Chapter 10 [Built-In Functions], page 77, and Chapter 11 [The Standard Library], page 93, discuss the built-in and standard library functions of the Q language. Chapter 12 [Clib], page 119, describes Q's "system module" which provides access to some important functions from the C library. The appendix gives additional information about some aspects of the language and its implementation. Appendix A [Q Language Grammar], page 171, contains a summary of the Q language syntax in BNF. Appendix B [Using Q], page 175, provides a description of the programming tools included in the Q programming system, Appendix C [C Language Interface], page 191, describes Q's interface to the C programming language, and Appendix D [Debugging], page 199, is a brief introduction to the symbolic debugger built into the Q interpreter. Finally, Appendix E [Running Scripts in Emacs], page 203, discusses how Q scripts can be edited and run using the Emacs editor.

Acknowledgements

I would like to acknowledge the support and helpful critizism provided by colleagues at Mainz, Bremen and at INRIA Lorraine. In particular, I have to thank Frank Drewes at Bremen who was involved in the initiation of this project, and Klaus Barthelmann at Mainz with whom I had lengthy and enlightening discussions concerning design and implementation issues, and who provided many comments which helped to improve this manual. They both tested various beta versions of the Q interpreter, and contributed sample scripts as well as substantial parts of the standard Q library.

Thanks are also due to the developers of ML and Haskell. While these people have not been involved with Q in any way (most probably they do not even know about it), their work, which has changed the landscape of functional programming, has been a constant source of inspiration for me.

2 Getting Started

A new programming language is best conveyed through some examples, and therefore it is common practice to start a language description with an introductory chapter which treats the most essential language features in a fairly informal manner. This is also the purpose of the present chapter. We first show how some of the "standard features" of the Q programming language can be employed for using the Q interpreter effectively as a sophisticated kind of "desktop calculator". The remainder of this section then adresses the question of how you can extend the Q environment by providing your own definitions in Q scripts, and describes the evaluation process and the treatment of runtime errors in the interpreter.

2.1 Using the Interpreter

To begin with, let us show how the Q interpreter is invoked to evaluate some expressions with the built-in functions provided by the Q language. Having installed the Q programming system, you can simply invoke the interpreter from your shell by typing `q`. The interpreter will then start up, display its sign-on, and leaves you at its prompt:

```
==>
```

This indicates that the interpreter is waiting for you to type an expression. Let's start with a simple one:

```
==> 23
23
```

Sure enough, that's correct. Any integer is a legal value in the Q language, and the common arithmetic operations work on these values as expected. Q integers are "bignums", i.e., their size is only limited by available memory:

```
==> 16753418726345 * 991726534256718265234
16614809890429729930396098173389730
```

"Real" numbers (more exactly, their approximation using 64 bit double precision floating point numbers) are provided as well. Let's try these:

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

The `sqrt` identifier denotes a built-in function which computes the square root of its argument. (A summary of the built-in functions can be found in Chapter 10 [Built-In Functions], page 77.)

What happens if you mistype an expression? Let's see:

```
==> sqrt (16.3805*5)/,05
! Syntax error
>>> sqrt (16.3805*5)/,05
                     ^
```

As you can see, the interpreter not only lets you know that you typed something wrong, but also indicates the position of the error.

At this point, it is time to try out the interpreter's "command history". Using the up and down arrow keys, you can cycle through the expressions you have typed before, edit them, and resubmit a line by hitting the carriage return key. Also note that what you typed is stored in a "history file" when you exit the interpreter, which is reloaded next time the interpreter is invoked. A number of other useful keyboard commands are provided, see section "Command Line Editing" in *The GNU Readline Library*. In particular, you can have the command line editor "complete" function symbols with the `<TAB>` key. E.g., if you type in `sq` and press the `<TAB>` key, you will get the `sqrt` function.

The interpreter also maintains a global variable environment, in which you can store arbitrary expression values. This provides a convenient means to define abbreviations for frequently-used expressions and for storing intermediate results. For instance:

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

==> X
81.9025

==> def f = sqrt

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

In fact, the above definitions and expressions can also be entered on a single line, using commas to separate different definitions in a `def` command, and semicolons to separate different expressions and `def` statements:

```
==> def X = 16.3805*5, f = sqrt; X; f X/.05
81.9025
181.0
```

Another useful feature is the built-in "anonymous" variable '`_`', which is always set to the value of the most recent expression value printed by the interpreter:

```
==> _
181.0

==> 2*_
362.0
```

Sometimes you would also like the interpreter to "forget" about a definition. This can be done by means of an `undef` statement:

```
==> undef X; X
X
```

Besides `def` and `undef`, the interpreter provides a number of other special commands. The most important command for beginners certainly is the `help` command, which displays the online manual using the GNU info reader. You can also run this command with a keyword to be searched in the info file. For instance, to find out about all special commands provided by the interpreter, type the following:

```
==> help commands
```

(Type `q` when you are done reading the info file.)

Other useful commands are `who` which prints a list of the user-defined variables, and `whos` which describes the attributes of a symbol:

```
==> who
f

==> whos f sqrt

f               user-defined variable symbol
                = sqrt

sqrt            builtin function symbol
                sqrt X1
```

You can save user-defined variables and reload them using the `save` and `load` commands:

```
==> save
saving .q_vars

==> def f = 0; f
0

==> load
loading .q_vars

==> f
sqrt
```

Some statistics about the most recent expression evaluation can be printed with the `stats` command:

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

==> stats
0 secs, 3 reductions, 3 cells
```

As indicated, the `stats` command displays the (cpu) time needed to complete an evaluation, the number of "reductions" (a.k.a. basic evaluation steps) performed by the interpreter, and the number of expression "cells" needed during the course of the computation. This information is useful for profiling purposes.

Other commands allow you to edit and run a script directly from the interpreter, and to inspect and set various internal parameters of the interpreter; see Section B.2 [Command Language], page 181.

Let us now return to expression evaluation. The Q interpreter can carry out computations on non-numeric data as well. For instance, strings and lists may be concatenated using the `++` operator:

```
==> "abc"++"xyz"
```

```
"abcxyz"

==> [a,b,c]++[x,y,z]
[a,b,c,x,y,z]
```

To determine the length of a string or list, we can use the unary # operator, and zero-based indexing is implemented with the binary subscript operator !:

```
==> #"abc"; "abc"!1
3
"b"
==> #[a,b,c]; [a,b,c]!1
3
b
```

Analogous operations are also provided for "tuples", which are Q's equivalent of "vectors" or "arrays" in other languages, and are written as expression sequences enclosed in parentheses, like (a,b,c).

Some remarks about the notion of function applications in the Q language are in order. As indicated in the examples above, function application is simply denoted by juxtaposition. Parentheses are only required for setting off nested applications and to change the precedence of operators.

Another important point is that the Q language, like most contemporary functional languages, allows you to turn any built-in operator into a prefix function by enclosing it in parentheses. Thus (+) denotes the function which adds its two arguments, and X+1 can also be written as (+) X 1; in fact, the former expression is nothing but "syntactic sugar" for the latter, see Section 6.4 [Built-In Operators], page 33. Furthermore, function application is in fact an explicit (binary and left-associative) operation, which makes it possible to write down "curried" expressions like (*) 2 which denotes the function which doubles its argument. Q also supports the usual *operator section* notation which allows you to specify a binary operator with only either its left or right operand. For instance, (1/) denotes the reciprocal, (+1) the "increment by 1", and (2*) (again) the doubling function.

It is quite instructive to keep on playing a bit like this, to get a feeling of what the Q interpreter can do. You might also wish to consult Chapter 10 [Built-In Functions], page 77, which discusses the built-in functions, and Appendix B [Using Q], page 175, which shows how the interpreter is invoked and what additional capabilities it offers. To exit the interpreter when you are finished, invoke the built-in quit function:

```
==> quit
```

This function does not return a value, but immediately returns you to the operating system's command shell.

2.2 Using the Standard Library

The standard library is a comprehensive collection of useful functions and data structures which are *not* provided as built-ins, but are implemented by scripts which are mostly written in Q itself. Most of these scripts are included in the "prelude script" prelude.q, which is always loaded by the interpreter, so that the standard library functions are always available. See Chapter 11 [The Standard Library], page 93, for an overview of these operations. You can check which standard library scripts a.k.a. "modules" are actually loaded with the interpreter's `modules` command. With the standard prelude loaded, this command shows the following:

```
==> modules
array           assert          bag             clib*
comp            complex         cond            dict
error           hdict           heap            lambda
list            math            prelude         set
sort            stddecl         stdlib          stdtypes
stream          string          typec
```

As you can see, there are quite a few modules already loaded, and you can use the functions provided by these modules just like any of the built-in operations. The standard library provides a lot of additional functions operating on numbers, strings and lists. For instance, you can take the sum of a list of (integer or floating point) numbers simply as follows:

```
==> sum [1,2,3,4,5]
15
```

In fact, the library provides a rather general operation, `foldl`, which iterates a binary function over a list, starting from a given initial value. Using the `foldl` function, the above example can also be written as follows:

```
==> foldl (+) 0 [1,2,3,4,5]
15
```

(There also is a `foldr` function which works analogously, but combines list members from right to left rather than from left to right.)

To generate a list of numbers, we can use the `nums` function:

```
==> nums 1 5
[1,2,3,4,5]
```

The `nums` function is just a special instance of the general-purpose list-generating function `while`. For instance, we can generate the list of all powers of 2 in the range [1,1000] as follows:

```
==> while (<=1000) (2*) 1
[1,2,4,8,16,32,64,128,256,512]
```

(Recall from the previous section that `(2*)` is the doubling function. And `(<=1000)` is a predicate which checks its argument to be less than or equal to `1000`.)

If we want to generate a list of a given size, we can use `iter` instead. So, for instance, we might compute the value of a finite geometric series as follows:

```
==> sum (iter 4 (/3) 1)
1.48148148148148
```

The map function allows you to apply a function to every member of a list. For instance, the following expression doubles each value in the list [1,2,3,4,5]:

```
==> map (2*) (nums 1 5)
[2,4,6,8,10]
```

Lists can also be filtered with a given predicate:

```
==> filter (>=3) (nums 1 5)
[3,4,5]
```

The scan operation allows us to compute all the sums of initial segments of a list (or accumulate any other binary operation over a list):

```
==> scan (+) 0 (nums 1 5)
[0,1,3,6,10,15]
```

Initial and remaining parts of a list can be computed with take, drop and takewhile, dropwhile:

```
==> take 3 (nums 1 5); drop 3 (nums 1 5)
[1,2,3]
[4,5]

==> takewhile (<=3) (nums 1 5); dropwhile (<=3) (nums 1 5)
[1,2,3]
[4,5]
```

To transpose a list of lists (i.e., map a list of lists to a list of lists of corresponding elements), you can use the transpose function:

```
==> transpose [[1,2,3,4,5],[A,B,C,D,E]]
[[1,A],[2,B],[3,C],[4,D],[5,E]]
```

All the operations described above – and many others – are provided by the stdlib.q script. It is instructive to take a look at this script and see how the operations are defined.

In addition, the list.q script provides yet another list generating function listof for writing down so-called "list comprehensions", which allow you to describe list values in much the same way as sets are commonly specified in mathematics. For instance, you can generate a list of pairs (I,J) with 1<=J<I<=5 simply as follows:

```
==> listof (I,J) (I in nums 1 5, J in nums 1 (I-1))
[(2,1),(3,1),(3,2),(4,1),(4,2),(4,3),(5,1),(5,2),(5,3),(5,4)]
```

We also have a random number generator, which is implemented by the built-in random function. Here is how we can generate a list of 5 pseudo random 32 bit integers:

```
==> listof random (I in nums 1 5)
[1960913167,1769592841,3443410988,2545648850,536988551]
```

To get random floating point values in the range [0,1] instead, we simply divide the results of random by 0xffffffff:

```
==> listof (random/0xffffffff) (I in nums 1 5)
[0.18231268883271,0.240211885944058,0.594435058439717,0.516151836029289,
```

```
0.526110585901446]
```

Lists can be sorted using quicksort (this one comes from `sort.q`):

```
==> qsort (<) _
[0.18231268883271,0.240211885944058,0.516151836029289,0.526110585901446,
0.594435058439717]
```

The standard library also provides “lazy” lists, a.k.a. “streams”, which are defined in `streams.q`. Streams are like lists but can actually be infinite because their elements are only produced “on demand”. Most list operations carry over to streams accordingly. For instance, if we want to create a geometric series like the one generated with `iter` above, but we do not know how many elements will be needed in advance, we can employ the stream iteration function `iterate`:

```
==> iterate (/3) 1
bin 1 (iterate (/3) ((/3) 1))
```

The `bin` stream “constructor” works in the same way as the `[|]` list constructor, but is “special” in that it does *not* evaluate its arguments. To get all members of the series we can apply the `scan` function:

```
==> def S = scan (+) 0 _
```

Now we can extract any number of initial values of the series using the `take` operation, and convert the resulting stream to an ordinary list with the `list` function. For instance, if we are interested in the first five values of the series, we proceed as follows:

```
==> list (take 5 S)
[0,1,1.33333333333333,1.44444444444444,1.48148148148148]
```

Note that the stream `S` is really infinite; if we want, we can extract *any* value in the series:

```
==> S!9999
1.5
```

Let’s see how many iterations are actually required to reach the limit `1.5` with an error of at most `1e-15`:

```
==> #takewhile (>1e-15) (map (1.5-) S)
32
```

This means that the sum of the first 31 series terms is needed to get an accuracy of 15 digits (which is the best that we can hope for with 64 bit floating point values). We can readily verify this using `iter`:

```
==> sum (iter 31 (/3) 1)
1.5
```

Another frequently used operation is `lambda` (contained in `lambda.q`) which allows you to create new function objects “on the fly”. For instance, another way to represent the doubling function is:

```
lambda X (2*X)
```

(Note that this is not quite the same as the prefix function `(2*)`. Lambdas actually evaluate to so-called "combinator" expressions which represent the denoted function objects, see Section 11.7 [Lambda Calculus], page 103. For instance, the value of `lambda X (lambda Y (X*Y))` is the combinator expression `_B _A (*)`.)

The `def` statement can be used to assign lambdas to variables in the interpreter. For instance:

```
==> def double = lambda X (2*X); double 7
14
```

The lambda variable can also be an arbitrary pattern to be matched against the actual parameter, e.g.:

```
==> lambda (X,Y) ((1-X)*Y) (.9,.5)
0.05
```

The `lambda` function is also used to implement the variable bindings in the `listof` operation, see above. Thus variable patterns can also be used inside list comprehensions:

```
==> listof (2*I+J) ((I,J) in [(1,2),(3,4),(5,6)])
[4,10,16]
```

Besides string and list processing functions and general utility functions of the kind sketched out above, the standard library also includes a collection of efficient "container" data structures which are useful in many applications. For instance, so-called *hashed dictionaries* (also known as "hashes" or "associative arrays") are implemented by the `hdict.q` script. The following expressions show how to create a dictionary object from a list, and how to access this dictionary using the index (`!`), `keys` and `vals` operations:

```
==> def D = hdict [(foo,99),(bar,-1),(gnu,foo)]; D!gnu
foo

==> keys D
[foo,bar,gnu]

==> vals D
[99,-1,foo]
```

This completes our little tour through the standard library. To find out more, please refer to Chapter 11 [The Standard Library], page 93, or take a look at the scripts themselves.

2.3 Writing a Script

Now that we have taken the first big hurdle and are confident that we can actually get the Q interpreter to evaluate us some expressions, let us try to define a new function for use in the interpreter. That is, we are going to write our first own Q "program" or "script".

In the Q language, new functions are defined by the means of "equations", in a way similar to function definitions in mathematics. Having exited the Q interpreter, invoke your favourite text editor and enter the following simple script which implements the well-known factorial function:

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

Save the script in the file `fac.q` and then restart the interpreter as follows (here and in the following '`$`' denotes the command shell prompt):

```
$ q fac.q
```

You can also edit the script from within the interpreter (using `vi` or the editor named by the `EDITOR` environment variable) and then restart the interpreter with the new script, as follows:

```
==> edit fac.q

==> run fac.q
```

In any case, now the interpreter "knows" about the definition of `fac` and we can use it like any of the built-in operations:

```
==> fac 10
3628800
```

For instance, what is the number of 30-element subsets of a 100-element set?

```
==> fac 100 div (fac 30*fac 70)
29372339821610944823963760
```

As another example, let us write down Newton's algorithm for computing roots of a function. Type in the following script and save it to the file `newton.q`:

```
newton DX DY F          = until (satis DY F) (improve DX F);
satis DY F X            = abs (F X) < DY;
improve DX F X          = X - F X / derive DX F X;
derive DX F X           = (F (X+DX) - F X) / DX;
```

Restart the interpreter with the `newton.q` script and try the following. Note that the target function here is `lambda Y (Y^3-X)` (see the description of the `lambda` function in the preceding section) which becomes zero when `Y` equals the cube root of `X`.

```
==> def cubrt = lambda X (newton eps eps (lambda Y (Y^3-X)) X)

==> def eps = .0001

==> cubrt 8
2.00000000344216
```

Well, this is not *that* precise, but we can do better:

```
==> def eps = 1e-12

==> cubrt 8
2.0
```

2.4 Definitions

As mentioned in the previous section, in the Q language function definitions take the form of equations which specify how a given expression pattern, the left-hand side of the equation, is transformed into a new expression, the right-hand side of the equation. The left-hand side may introduce "variables" (denoted by capitalized identifiers, as in Prolog) which play the role of formal parameters in conventional programming languages. For instance, the following equation defines a function `sqr` which squares its (numeric) argument:

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

An equation may also include a condition part, as in the definition of the faculty function from the previous section:

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

As indicated above, several equations for the same left-hand side can be factored to the left, omitting repetition of the left-hand side. Furthermore, the `otherwise` keyword may be used to denote the "default" alternative.

The left-hand side of an equation may actually be an arbitrary compound expression pattern to be matched in the expression to be evaluated. For instance, the expressions `[]` and `[X|Xs]` denote, respectively, the empty list and a list starting with head element `X` and continuing with a list `Xs` of remaining elements, just as in Prolog. We can employ these patterns in order to implement our own variation of the `sum` function which computes the sum of a list of numbers as follows:

```
private sum Xs;
sum []                  = 0;
sum [X|Xs]              = X+sum Xs;
```

Thus no explicit operations for "extracting" the members from a compound data object such as a list are required; the components are simply retrieved by "pattern matching". We remark that pattern matching works in variable definitions as well. For instance, in the interpreter you can bind variables to the head element and the remainder of a list using a single definition as follows:

```
==> def [X|Xs] = [1,2,3]; X; Xs
1
[2,3]
```

The `private` declaration in the `sum` example above lets the interpreter know that we introduce a new `sum` symbol. This is necessary because the standard library already contains a definition for `sum`, and we want to work with our own `sum` function here, rather than the one provided in the standard library. When the above script is loaded in the interpreter, `sum` will refer to our "private" version, but we could still access the standard library function using the "qualified" symbol `stdlib::sum`, which explicitly refers to the "namespace" of the `stdlib.q` module in which the standard `sum` function is defined. For more information about namespaces and qualified identifiers see Chapter 4 [Scripts and Modules], page 23.

As you can see in previous examples, in the Q language declarations are usually optional; new function and variable symbols are declared implicitly by their use, and by default the interpreter distinguishes between function and variable symbols by taking a look at the initial letter of an identifier. However, as the preceding example shows, there are some situations in which a symbol *has* to be declared explicitly. More about that in Chapter 5 [Declarations], page 25.

But let us return to our main concern, equations and pattern matching. Pattern matching also works with function applications which may themselves be used as compound, tree-like data structures. For instance, the following definition implements an insertion operation on binary trees, which are represented by the "constructor" symbols `nil` and `bin`:

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

The Q interpreter treats equations as "rewrite rules" which are used to reduce expressions to "normal forms". Evaluation normally uses the standard "leftmost-innermost" rule, also known as "applicative order" or "call by value" evaluation. Using this strategy, expressions are evaluated from left to right, innermost expressions first; thus the arguments of a function application (and also the function object itself) are evaluated before the function is applied to its arguments.

An expression is in normal form if it cannot be evaluated any further by applying matching equations (including some predefined rules which are used to implement the built-in operations, see Chapter 7 [Equations and Expression Evaluation], page 41). A normal form expression simply stands for itself, and represents a "value" in the Q language. The built-in objects of the Q language (such as numbers, strings and lists) are always in normal form, but also any other symbol or function application which does not match a built-in or user-defined equation. This means that Q is essentially an "exception-free" language. That is, "error conditions" such as "division by zero" or a "wrong argument type" do *not* raise any exceptions by default, but simply cause the offending expression to be returned "as is". For instance:

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

You may be disconcerted by the fact that `23/0` actually denotes a legal value in the Q language. However, this is one of Q's key features as a term rewriting programming language, and it adds a great amount of flexibility to the language. Most importantly, it allows expressions, even expressions involving variables, to be evaluated in a symbolic fashion. For instance, you might add rules for algebraic identities to your script and have the interpreter simplify expressions according to these rules (just think of symbolic differentiation and similar applications). On the other hand, the Q language also allows you to refine the definition of any built-in operation and thus it is a simple matter to add an "error rule" like the following to your script:

```
X/0                     = error "Division by zero!";
```

(The `error` function is defined in the standard library script `error.q`, see Section 11.14 [Diagnostics and Error Messages], page 117.) In order to implement more elaborate error handling, you can also use Q's built-in throw and catch functions to raise and respond to exceptions on certain error conditions. This is discussed in Section 10.6 [Exception Handling], page 86. For instance, the following rule will raise an exception on division by zero, instead of terminating the evaluation with an error message:

```
X/0                     = throw "Division by zero!";
```

2.5 Runtime Errors

There are a few error conditions which may cause the Q interpreter to abort the evaluation with a runtime error message. A runtime error arises when the interpreter runs out of memory or stack space, when the user aborts an evaluation with the interrupt key (usually Ctl-C), and when the condition part of an equation does not evaluate to a truth value (true or false). (All these error conditions can also be handled by the executing script, see Section 10.6 [Exception Handling], page 86.)

You can use the break on command to tell the interpreter that it should fire up the debugger when one of the latter two conditions arises. After Ctl-C, you can then resume evaluation in the debugger, see Appendix D [Debugging], page 199. This is not possible if the interpreter stumbles across an invalid condition part; in this case the debugger is invoked merely to report the equation which caused the error, and to inspect the call chain of the offending rule. Hitting the carriage return key will return you to the interpreter's evaluation loop.

For instance, reload the fac.q script from Section 2.3 [Writing a Script], page 12:

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

Now enable the debugger with the break command and type some "garbage" like fac fac:

```
==> break on; fac fac
! Error in conditional
  0>  fac.q, line 1: fac fac  ==>  fac*fac (fac-1) if fac>0
(type ? for help)
:
```

What happened? Well, the debugger informs us that the error occurred when the interpreter tried to apply the first equation,

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

to the expression fac fac. In fact this is no big surprise since we cannot expect the interpreter to know how to compare a symbol, fac, with a number, 0, which it tried when evaluating the condition part of the above equation. So let us return to the interpreter's prompt by hitting the carriage return key:

```
: <CR>

==>
```

The interpreter now waits for us to type the next expression. (For a summary of debugger commands please refer to Appendix D [Debugging], page 199. You can also type `?` or `help` while in the debugger to obtain a list of available commands.)

3 Lexical Matters

The vocabulary of the Q programming language consists of notations for identifiers, integers, floating point numbers, character strings, comments, a few reserved words which may not be used as identifiers, and some special symbols which are used as operators and delimiters. Whitespace (blanks, tabs, newlines, form feeds) serves as a delimiter between adjacent symbols, but is otherwise ignored. Comments are treated like whitespace:

```
/* This is a comment ... */
```

Both Prolog- and BCPL- resp. C++-style line-oriented comments are supported:

```
% this is a comment ...
// C++-style comment ...
```

Furthermore, lines beginning with the `#!` symbol denote a special type of comment which may be processed by the operating system's command shell and the Q programming tools. On UNIX systems, this (odd) feature allows you to execute Q scripts directly from the shell (by specifying the `q` program as a command language processor) and to include compiler and interpreter command line options in a script (see Section B.4 [Running Scripts from the Shell], page 188).

Identifiers are denoted by the usual sequences of letters (including '`_`') and digits, beginning with a letter. Upper- and lowercase is distinct. In the Q language, identifiers are used to denote both *function* and *variable* symbols. As in Prolog, a capitalized identifier (such as `X`, `Xmax` and `XMAX`) indicates a variable symbol; all other identifiers denote function symbols (unless they are declared as "free" variables, see below). In difference to Prolog, the underscore '`_`' counts as a *lowercase* letter, hence `_MAX` is a function symbol, not a variable. However, as an exception to the general rule, the identifier '`_`' does denote a variable symbol, the so-called *anonymous* variable.

Variables actually come in two flavours: *bound* and *free* variables, i.e., variables which also occur on the left-hand side of an equation, and variables which only occur on the right-hand side and/or in the condition part of an equation. Identifiers may also be *declared* as free variables; see Chapter 5 [Declarations], page 25. In this case, they may also start with a lowercase letter. Furthermore, all new symbols created interactively in the interpreter are treated as free variable symbols.

Both function and free variable identifiers may also be *qualified* with a *module* identifier prefix (cf. Chapter 4 [Scripts and Modules], page 23), to specifically denote a symbol of the given module. Formally, the syntax of identifiers is described by the following grammatical rules:

```
identifier              : unqualified-identifier
                        | qualified-identifier
qualified-identifier    : unqualified-identifier '::'
                          unqualified-identifier
unqualified-identifier  : variable-identifier
                        | function-identifier
variable-identifier     : uppercase-letter {letter|digit}
                        | '_'
```

```
function-identifier     : lowercase-letter {letter|digit}
letter                  : uppercase-letter|lowercase-letter
uppercase-letter        : 'A'|...|'Z'
lowercase-letter        : 'a'|...|'z'|'_'
digit                   : '0'|...|'9'
```

(Please refer to Appendix A [Q Language Grammar], page 171, for a description of the BNF grammar notation used throughout this document.)

The reserved words of the Q language are:

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

Signed decimal numeric constants are denoted by sequences of decimal digits and may contain a decimal point and/or a scaling factor. Integers can also be denoted in octal or hexadecimal, using the same syntax as in C:

```
number                  : ['-'] unsigned-number
unsigned-number         : '0' octdigitseq
                        | '0x' hexdigitseq
                        | '0X' hexdigitseq
                        | digitseq ['.' [digitseq]] [scalefact]
                        | [digitseq] '.' digitseq [scalefact]
digitseq                : digit {digit}
octdigitseq             : octdigit {octdigit}
hexdigitseq             : hexdigit {hexdigit}
scalefact               : 'E' ['-'] digitseq
                        | 'e' ['-'] digitseq
digit                   : '0'|...|'9'
octdigit                : '0'|...|'7'
hexdigit                : '0'|...|'9'|'a'|...|'f'|'A'|...|'F'
```

Simple digit sequences without decimal point and scaling factor are treated as integers; if the sequence starts with '0' or '0x'/'0X' then it denotes an integer in octal or hexadecimal base, respectively. Other numbers denote (decimal) floating point values. If a decimal point is present, it must be preceded or followed by at least one digit. Both the scaling factor and the number itself may be prefixed with a minus sign. (Syntactically, the minus sign in front of a number is interpreted as unary minus, cf. Chapter 6 [Expressions], page 29. However, if unary minus occurs in front of a number, it is interpreted as a part of the number and is taken to denote a negative value. See the remarks concerning unary minus in Chapter 6 [Expressions], page 29.) Some examples:

```
0  -187326  0.0  -.05  3.1415e3 -1E-10 0177 0xaf -0XFFFF
```

String constants are written as character sequences enclosed in double quotes:

```
string                  : '"' {char} '"'
char                    : any character but newline and "
```

To include newlines, double quotes and non-printable characters in a string, the following escape sequences may be used:

`\n`	newline
`\r`	carriage return
`\t`	tab
`\b`	backspace
`\f`	form feed
`\"`	double quote
`\\`	backslash

Furthermore, a character may also be denoted in the form `\N`, where `N` is the character number in decimal, hexadecimal or octal (using the same syntax as for unsigned integer values).

A string may be continued across lines by putting the `\` character immediately before the end of the line, which causes the following newline character to be ignored.

Some examples:

`""`	empty string
`"A"`	single character string
`"\27"`	escape character (ASCII 27)
`"\033"`	same in octal
`"\0x1b"`	same in hexadecimal
`"a string"`	multiple character string
`"a \"quoted\" string"`	include double quotes
`"a line\n"`	include newline
`"a very \` `long line\n"`	continue across line end

Finally, as already mentioned, some special symbols are used as operators and delimiters:

```
~ , ; : | || < > = <= >= <> ++ + - * / \ ^ ! # @ ( ) [ ]
```

4 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:

```
script                  : {declaration|definition}
```

In order to make a given set of definitions available for use in the interpreter, you can just collect the relevant declarations, variable definitions and equations in a single script which you can submit to the interpreter for execution. The interpreter in turn invokes the compiler to translate the script into a *bytecode* file which is loaded by the interpreter (see Appendix B [Using Q], page 175). A script may also be empty, in which case it does not provide any additional definitions at all.

If you put all definitions into a single script, that's all there is to it. However, you might wish to break down a larger script into a collection of smaller units, also called *modules*, which can be managed separately and which are linked together by the compiler into a single bytecode file. 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, you must use an *import* or *include* declaration (the difference between these two will be pointed out below):

```
declaration             : 'import' module-spec {',' module-spec} ';'
                        | 'include' module-spec {',' module-spec} ';'
module-spec             : module-name ['as' unqualified-identifier]
module-name             : unqualified-identifier
                        | string
```

For instance, the following declaration imports two modules `foo` and `bar`:

```
import foo, bar;
```

Note that the Q language allows you to declare symbols either as "public" or "private", see Chapter 5 [Declarations], page 25. Outside a module, only the public symbols are accessible when the module is imported in another module. Moreover, import is not "transitive", i.e., importing a module `foo` does *not* automatically give you access to the symbols imported by `foo`, but only to the public symbols declared in `foo` itself. In contrast, the `include` declaration causes all imports and includes of the included module to be "reexported". For instance, let us consider the case that module `foo` includes module `bar`:

```
include bar;
```

Then by importing module `foo` you also gain access to the public symbols of module `bar` (and, recursively, to the modules included by `bar`, etc.). This provides a means to group different modules together in one larger module. For instance, the standard prelude script (cf. Chapter 11 [The Standard Library], page 93) simply includes all the other standard library modules. (It is also possible to selectively reexport symbols from imported modules, using a corresponding public symbol declaration, see Chapter 5 [Declarations], page 25.)

In the Q language, each module has its own separate namespace. This means that two different modules, say `foo1` and `foo2`, may both provide their own public function symbol named `foo`. If both `foo1` and `foo2` are imported in the same script, you can distinguish between the two by using a qualified identifier, using the module name as a prefix, e.g., `foo1::foo` or `foo2::foo`. (Writing simply `foo` in this case will produce an error message because the reference is ambiguous.)

Import and include declarations can occur anywhere in a script (and will be active from this point on up to the end of the script file), but it is common practice to put them near the beginning. As in most other programming languages, module dependencies must always be acyclic. If the compiler finds a cyclic chain of `import` and `include` declarations, such as a module importing itself or a module `foo` importing a module `bar` which in turn imports `foo` again, it produces an error message.

As indicated in the syntax rules, the names of the modules to be imported can either be specified using an unqualified identifier, or a string denoting a full or relative pathname, e.g.:

```
import "/home/ag/q/stuff.q";
```

In this case, you can also omit the '`.q`' suffix, it will be supplied automatically when necessary. Moreover, the module identifier is derived automatically as the basename of the script filename (i.e., `stuff` in the above example); therefore, the basename of a script file should always be a legal Q identifier.

If no absolute path is specified, the interpreter locates script files using its "search path", which usually contains the current directory and other system-dependent or user-defined locations where "library" scripts are kept, see Appendix B [Using Q], page 175, for details.

The '`as`' keyword can be used to explicitly specify an unambiguous module name. This is useful, in particular, if the basename of the module is *not* a valid identifier, and for the purpose of resolving name clashes. For instance:

```
import "my/stdlib.q" as mystdlib;
```

Simply importing `"my/stdlib.q"` under its own name in this case would produce an error message, because the name of the module collides with the standard library module of the same name, and the compiler enforces that all module names are unique.

Talking about the standard library (see Chapter 11 [The Standard Library], page 93), we remark that the public symbols of these modules are always available because the `prelude.q` script, which includes all these modules, is by default always imported in *any* script file (except for the prelude itself, and its includes). When looking up an unqualified symbol in a given script, the compiler first searches for a symbol defined in that script, then for a symbol in an imported or included module, then for a symbol defined by the prelude and its includes, and finally for a built-in symbol. You can instruct the compiler to inhibit the default import of the prelude, by using the `--no-prelude` option, see Appendix B [Using Q], page 175. Moreover, you can also override the standard prelude with your own, if it occurs on the search path *before* the standard prelude, see Section B.3 [Setting up your Environment], page 187.

The namespace available in the interpreter is that of the main script. The interpreter also allows you to dynamically import additional modules in the global scope. Moreoever, in the interpreter it is possible to gain access to *all* public and private symbols of the program (also in modules not directly imported in the main script) using qualified identifiers.

5 Declarations

The Q language allows you to declare function and (free) variable symbols explicitly, by means of the syntactic constructs discussed in this chapter. Symbol declarations are optional; if you introduce a new symbol without declaring it, the compiler declares it for you. However, you will sometimes wish to ensure that a new symbol is created to override a symbol of the same name from an imported module, or you want to attach special attributes to a symbol, and then you have to use an explicit declaration. Syntactically, symbol declarations take the following form:

```
declaration             : prefix headers ';'
                        | [scope] 'type' unqualified-identifier
                          [':' identifier] ['=' sections] ';'
                        | [scope] 'extern' 'type' unqualified-identifier
                          [':' identifier] ';'
                        | [scope] 'type' qualified-identifier
                          ['as' unqualified-identifier] ';'

prefix                  : scope
                        | [scope] modifier {modifier}

scope                   : 'private'|'public'

modifier                : 'const'|'special'|'extern'|'var'

headers                 : header {',' header}

header                  : unqualified-identifier
                          {['~'] variable-identifier}
                        | qualified-identifier
                          {['~'] variable-identifier}
                          'as' unqualified-identifier ';'

sections                : section {'|' section}

section                 : [prefix] headers
```

For instance, the following are all valid symbol declarations:

```
public foo;
private extern bar X;
public special lambda X Y;
special ifelse ~P X Y as myifelse;

const red, green, blue;
var FOO, BAR;

public type BinTree = const nil, bin X T1 T2;
private type Day = const sun, mon, tue, wed, thu, fri, sat;
```

The keywords `private` and `public` specify the *scope* of a symbol. *Public* symbols are accessible outside a script, and can be imported by other modules (see Chapter 4 [Scripts and Modules], page 23). If the keywords `private` and `public` are omitted, the scope defaults to private.

The `special` keyword serves to declare *special forms*, which are described in Chapter 9 [Special Forms], page 69. Function symbols declared with `const` introduce "constant" or "constructor" symbols; the compiler enforces that expressions created with such symbols are not redefined in an equation (cf. Section 7.1 [Equations], page 41). The built-in constants `true`, `false`, `[]` and `()` are already predeclared as `const`. Non-`const` function symbols can also be declared as `extern`, meaning that the corresponding function is actually implemented by a corresponding "external" module, see Appendix C [C Language Interface], page 191.

The `var` keyword allows you to declare "free" variable symbols, which can be assigned a value by means of a *variable definition*, see Section 7.3 [Free Variables], page 46. If you use such a declaration, the variable symbol may also start with a lowercase letter (if it has not already been declared as a function symbol); note that without such a declaration, an identifier starting with a lowercase letter will implicitly be declared as a function symbol. Variable symbols can also be declared as `const`; in this case the variable can only be assigned to *once* and always refers to the same value once it has been defined. The built-in variables `INPUT`, `OUTPUT`, `ERROR` and `ARGS` are predeclared as `const`, see Section B.2 [Command Language], page 181.

As indicated, the scope-modifier prefix of a declaration is followed by a comma-separated list of *headers*. The first identifier in each header states the symbol to be declared. In function symbol declarations the function identifier may be followed by a list of variable identifiers which are used to denote the arguments of the declared function symbol. The variables are effectively treated as comments; only their number (called the *arity* of a function symbol) is recorded to check the consistency of different declarations of the same symbol, and to specify the number of "special" arguments in a `special` declaration. In `special` declarations variables may also be prefixed with '`~`' to declare them as "non-special" arguments; see Chapter 9 [Special Forms], page 69.

The first declaration of an unqualified identifier in a module always introduces a new symbol in the module's namespace. By default (if no explicit declaration precedes the first use of a new, unqualified symbol), the compiler automatically declares it as a `private` nullary function or variable symbol, depending on the case of the initial letter of the identifier (as already mentioned in Chapter 3 [Lexical Matters], page 19, capitalized identifiers are interpreted as variable symbols, others as function symbols).

The Q language admits multiple declarations of the same symbol, and the compiler will verify the consistency of all declarations of a given symbol. That is, if a symbol is declared with different attributes (like `var`, `const` or `special`) or different number of arguments, or if non-special arguments of a special form are declared differently, then the compiler will issue an error message.

As indicated by the syntactic rules, it is also possible to *redeclare* a *qualified* symbol. This requires that the target module either is the current module or has already been imported, and causes the compiler to both cross-check the declaration with the declaration in the imported module and redeclare the symbol within the current namespace. Such a redeclaration serves several different purposes. First, it allows you to ensure that an imported symbol was actually declared with the given attributes. Second, it lets you resolve name clashes by redeclaring the symbol in the current scope where it overrides imports of other modules; if you want, you can also import the symbol under a new name using an '`as`' clause. In these two cases the symbol will normally be redeclared as a private symbol. Third, by redeclaring a symbol as `public`, you cause the symbol to be *reexported* by the current module. This provides an alternative to the `include` declaration, and makes it possible to reexport only selected symbols imported from other modules (possibly under a new name, when using an '`as`' clause). Examples:

```
import gnats;
private gnats::foo X Y; // cross-check declaration of gnats::foo
public gnats::foo X Y as bar; // reexport gnats::foo as bar
```

There is yet another usage of an imported symbol redeclaration, namely the `extern` redeclaration. This is only possible with function symbols and if the redeclared symbol is not already declared `extern` by another module. It instructs the compiler that this module provides an external definition of the symbol which will override equational definitions in other modules, see Appendix C [C Language Interface], page 191, for details.

A *type declaration* introduces a type identifier, an optional supertype for the type, and an optional list of "constructor" symbols for the type. It associates the function symbols in the list with the given type. The symbol list is a `|`-delimited list of individual *sections*. Each section takes the form of an ordinary symbol declaration, consisting of scope/modifier prefix and a comma-delimited list of headers. The prefix is optional; by default, a symbol has the same scope as the type it belongs to. To change scope and attributes of a symbol on the list, use an appropriate scope/modifier prefix. For instance, you can declare a public `BinTree` type with private constructors `nil` and `bin` as follows:

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

Each constructor symbol may only be declared once; otherwise the compiler will issue an error message. Hence it is enforced that a constructor symbol only belongs to a single type, and that the number of arguments of that symbol is determined uniquely.

Type identifiers may begin with either an upper- or lowercase letter (the convention, however, is to use capitalized identifiers). There may only be a single declaration for each type. Type identifiers form a separate symbol category and therefore cannot collide with function or variable symbols. Types are used on the left-hand side of equations to restrict the set of expressions which can be matched by a variable; see Section 7.5 [Type Guards], page 49, for details. The Q language has nine predefined type symbols which distinguish the corresponding types of objects built into the language: `Int`, `Float`, `Num` (which is the supertype for both `Int` and `Float`), `String`, `Char` (which is the subtype of `String` denoting the single-character strings), `File`, `List`, `Tuple` and `Bool`. (Besides this, there are also two built-in types for exceptions, see Section 10.6 [Exception Handling], page 86.)

Like function symbols, types imported from other modules can also be redeclared (possibly under a new name), and reexported, e.g.:

```
public type array::Array as MyArray;
```

In this case, no constructor symbols or supertype are specified. Types can also be declared as `extern`, to indicate that they are realized in a corresponding C module, as described in Appendix C [C Language Interface], page 191. In this case, too, there are no constructor symbols; however, a supertype may be specified. For instance:

```
extern type Bar;
```

In difference to function symbols, an existing type imported from another module cannot be redeclared as `extern`. Therefore an external definition must always be given by the module which originally declares the type.

6 Expressions

The notion of expressions is an intrinsic part of most, if not all, programming languages. However, in most programming languages expressions are merely treated as descriptions of computations to be performed in order to obtain the value of an expression, which usually belongs to some basic type provided by the language. In contrast, in the Q language expressions are objects in their own right which, as we shall see, are manipulated according to computational rules specified by the programmer.

As usual, expressions are built from certain kinds of basic objects. In the Q language, these are integers, floating point numbers, strings, function and variable symbols. These objects are combined into larger, compound expressions by means of applications, lists and tuples. For convenience, the Q language also provides prefix and infix operator symbols for the most common operations; these are actually treated as "syntactic sugar" for applications. The syntax of all these entities is described by the following grammatical rules:

```
expression              : identifier
                        | variable-identifier ':' identifier
                        | number
                        | string
                        | expression expression
                        | unary-op expression
                        | expression binary-op expression
                        | '(' [element-list] ')'
                        | '[' [element-list] ']'
                        | '(' op ')'
                        | '(' expression binary-op ')'
                        | '(' binary-op expression ')'

element-list            : expression-list ['|' expression]

expression-list         : expression {',' expression}

op                      : unary-op|binary-op

unary-op                : '-'|'#'|'not'|'''|'`'|'~'

binary-op               : '^'|'!'|'++'|'+'|'-'|'\'|'*'|'/'|'div'|'mod'
                        | 'and'|'or'|'and' 'then'|'or' 'else'
                        |'<'|'>'|'='|'<='|'>='|'<>'|'in'|'||'
```

6.1 Constants and Variables

The nonterminals `identifier`, `number` and `string` occuring in the syntax rules above refer to the lexical entities introduced in Chapter 3 [Lexical Matters], page 19. Note that the Q language actually distinguishes two different types of identifiers (function and variable identifiers) and two different types of numeric quantities (integers and floating point numbers). Integers are implemented as "bignums" using the GNU multiprecision package (GMP); thus the size of an integer value is only limited by available memory. Floating point values are implemented using 64 bit (i.e., double precision) floating point numbers; on most machines, these should provide nonzero absolute values ranging from 1.7E-308 to 1.7E308 and a precision of 15 decimal digits.

String constants are character sequences internally represented as character arrays permitting constant-time access to the individual characters of a string. There is no a priori length restriction for string constants. In the current implementation, the null character `\0` is reserved as a string terminator and must not be used as an ordinary string character.

Furthermore, the Q language provides four other built-in constants which denote the truth values (`true` and `false`), and the empty list and tuple (`[]` and `()`, to be discussed in Section 6.3 [Lists and Tuples], page 32).

A variable symbol can be "bound" or "free", depending on the context in which it occurs. We say that a variable is *bound* in an equation if it occurs on the left-hand side of the equation. Otherwise, the variable is a *free* variable. In this case, the variable may denote a value (introduced with `def`), or simply stands for itself. In any case, a variable is a legal atomic expression which may stand wherever a constant is allowed.

On the left-hand side of equations, it is possible to declare that a variable is of a given *type* by using the notation:

variable`:`*type*

This requires that you have previously declared *type* as a type identifier, see Chapter 5 [Declarations], page 25. When a type identifier is specified with a variable, the variable will only match values belonging to the given type, cf. Section 7.5 [Type Guards], page 49.

6.2 Applications

Application is probably the most important single construct in the Q language. It allows you to apply one object (the "function") to another one (the "argument"). This construct is used to denote "function calls" such as `sqrt 2` as well as "constructor terms" such as `bin X T1 T2` which encode tree-like data structures. Also, as we will see in Section 6.4 [Built-In Operators], page 33, the built-in operators `+`, `-`, etc. are merely syntactic sugar for applications. Indeed, applications could also be used to represent sequences of objects. However, as this latter kind of objects tends to be used so frequently in Q scripts, the Q language provides special support for sequences by means of the built-in list and tuple constructs discussed in Section 6.3 [Lists and Tuples], page 32.

As in other contemporary functional languages, application is a binary operation written simply as juxtaposition: `X Y` denotes the application of `X` to `Y`, both of which may be arbitrary expressions themselves. Application associates to the left; the construct `X Y1 ... Yn` is equivalent to `(...((X Y1) Y2) ...) Yn`, and denotes the application of `X` to n arguments `Y1`, ..., `Yn`. This style of writing function applications is commonly referred to as *currying*, after the American logician H.B. Curry. We will have to say more about this shortly.

Here are some valid examples of applications:

```
sqrt 2
```

```
sqrt (Y+1)
foo X (bar (Y-1)) Z
```

Note that since application is left-associative, nested applications in arguments must be set off with parentheses. For instance, `foo (bar X)` applies `foo` to `bar X`, whereas `foo bar X` applies `foo` to two arguments `bar` and `X`.

Since currying is so ubiquitous in functional programming, you should be well familiar with it, so let us briefly explain what it is, and what it is good for.

Functions of multiple arguments can generally be defined in two different ways:

- As a function taking a single, structured argument (usually a tuple). This is the traditional, "uncurried" method one commonly encounters in mathematics. For instance, we might define the function `max`, which takes the maximum of two values, as follows:

  ```
  max (X,Y)       = X if X>Y;
                  = Y otherwise;
  ```

- Recursively, as a function which, when given the first argument, yields another function of the remaining arguments. Such functions are called *curried*. For instance, here is a curried definition for the `max` function:

  ```
  max X Y         = X if X>Y;
                  = Y otherwise;
  ```

  Here `max X Y` is to be read as `(max X) Y`, meaning that by applying `max` to the first argument `X`, we obtain another function `max X`, which, when applied to the second argument `Y`, yields the maximum of `X` and `Y`.

The Q language supports both kinds of notations. Choosing between the two is more than just a matter of taste. Besides saving parentheses, curried functions have the chief advantage that they allow us to make use of initial "parts" of a multi-argument function. For instance, given the curried definition of `max` from above, `max 0` can be used to denote the function computing the "nonnegative part" of its argument (which is the argument itself if it is nonnegative, and zero otherwise). This does not work with the uncurried definition since that definition requires us to specify both arguments of `max` in one chunk; instead, we would have to start defining the derived function from scratch.

Uncurried definitions also have their merits. In particular, they allow us to define *variadic* functions, i.e., functions which can handle a varying number of arguments. For instance, the following definition enables us to apply the `max` function to both pairs and triples:

```
max (X,Y)               = X if X>=Y;
                        = Y otherwise;
max (X,Y,Z)             = max (X,max (Y,Z))
```

In fact, in the Q language it is possible to define generic functions which apply to any number of arguments specified as a tuple. For instance, the following version of `max` handles tuples of any size at least 2:

```
max (X,Y)               = X if X>=Y;
                        = Y otherwise;
max (X,Y|Zs)            = max (X,max (Y|Zs));
```

(In the above example, the notation `(...|Zs)` is used to denote a tuple whose "remaining elements" form the tuple `Zs`. This is explained in the following section.)

6.3 Lists and Tuples

Besides strings, the Q language provides two closely related general constructs for representing sequences of objects: lists and tuples.

The constant `[]` denotes the empty list. In general, a list consisting of elements `X1`, `X2`, ..., `Xn` is denoted `[X1,X2,...,Xn]`. For instance, `[a,b,c]` consists of three elements (symbols) `a`, `b` and `c`. It is possible to have nested lists, as in `[a,[b,c]]` which consists of two elements, the symbol `a` and the list `[b,c]`.

As in Prolog, lists are represented in a right-recursive fashion using a binary constructor `[|]` which takes as its arguments the head element of the list and the list of the remaining elements. Thus, `[a,b,c]` is simply a convenient shorthand notation for `[a|[b|[c|[]]]]`. You can also mix both styles of notation; for instance, `[a,b|[c,d]]` is another way to represent the 4-element list `[a,b,c,d]`.

Note that `[a|[b,c]]` is different from `[a,[b,c]]`: the former denotes a three-element list, while the latter is a two-element list whose second member happens to be a list itself. Also note that the `[|]` constructor can in fact be applied to any pair of values (the second value does not have to be a list); e.g., `[a|b]` is a perfectly well-formed expression (although the built-in length, indexing and concatenation operations described in Section 6.4.5 [String/List/Tuple Operators], page 38, will fail on such values).

Tuples work in much the same fashion as lists. The constant `()` denotes the empty tuple, and a tuple consisting of elements `X1`, `X2`, ..., `Xn` is written as `(X1,X2,...,Xn)`, which is equivalent to `(X1|(X2|...|(Xn|())))`, where the notation `(X|Xs)` denotes a tuple consisting of a first element `X` and the tuple `Xs` of the remaining elements.

The Q language also has the notion of a *1-tuple* `(X)=(X|())`. It is important to note that in Q a 1-tuple is really different from its single member. Otherwise, there could be no nested 1-tuples – in fact, due to the right-recursive nature of tuples in Q, there would be no nested tuples at the end of a tuple at all. Therefore Q distinguishes between 1-tuples and their members, and if you want to define a function operating on both 1-tuples and ordinary expressions, you will have to provide equations for both.

Unfortunately, since parentheses are used for two different purposes, namely for expression grouping and for tuples, 1-tuples also give rise to an ugly syntactic ambiguity: should an expression of the form `(X)` denote a 1-tuple or a simple parenthesized expression? For notational convenience, Q adopts the following convention: any "primary" expression (i.e., anything which binds stronger than an application) can be turned into a 1-tuple simply by enclosing it in parentheses. So, for instance, `(99)`, `((-99))`, `('X)`, `(foo)`, `((foo (bar X)))` all denote 1-tuples. Note the extra level of parentheses around compound expressions required to distinguish 1-tuples from ordinary parenthesized expressions. (This also applies to negative numbers like `-99`, which syntactically are applications of unary minus, i.e., a compound expression.) If the above sounds confusing to you, here is a simple rule of thumb: an extra level of parentheses is required whenever the target expression must be parenthesized when occuring as an argument of a function application. Nested 1-tuples can be obtained by adding extra levels of parentheses accordingly. E.g., `((99))` and `(((-99)))` both denote a 1-tuple of a 1-tuple of a number.

(NB: A common pitfall with Q's tuple notation is that one easily gets unwanted 1-tuples if one has the habit of decorating expressions with superflous parentheses. This is undoubtedly one of Q's worst "features", but the author just could not put up with alternative notations like curly braces, or the unwieldy `(X|())` syntax.)

The big difference between lists and tuples in the Q language is that, while lists are always represented as recursive data objects using a binary constructor symbol (just the way that they are written), tuples are actually implemented as "vectors" which are stored as contiguous sequences in memory. (Of course, this only works for "well-formed" tuples; if the "remainder" `Xs` of a tuple `(X|Xs)` is not a tuple, then this tuple can only be represented using an explicit application of the tuple constructor.) Therefore tuples normally use up much less memory than lists of the same size, and they also allow constant-time access to their members. The size of a tuple can be determined in constant time as well. In contrast, the same operations, when applied to a list, require time proportional to the size of the list. On the other hand, lists are more efficient when accessing the remainder part of a list using pattern matching, and when a new element is prepended to a list using the list constructor, which can both be done in constant time. Here a tuple needs time proportional to its size, since the member sequence of the original tuple must be copied when accessing its remainder part or when constructing a new tuple. (This also implies that converting a list to a tuple using the tuple constructor actually takes *quadratic* time and hence is quite slow for larger sequences; as a remedy, a built-in `tuple` function is provided which does the conversion in linear time, see Section 10.4 [Conversion Functions], page 79.)

These tradeoffs should be carefully considered when deciding whether to implement a given sequence as a list or a tuple. Tuples are usually the best choice for implementing fixed sequences requiring fast random access to its individual members, whereas lists provide an efficient means to represent sequences which have to be traversed and manipulated very frequently.

6.4 Built-In Operators

Besides the forms of expressions already discussed above, the Q language also provides a collection of infix and prefix operator symbols for common arithmetic, logical, relational and other operations. A complete list of these symbols is given below, in order of decreasing precedence:

`' ` ~`
: quotation operators (unary)

`^ !`
: exponentiation and subscript

`- # not`
: prefix operators (unary)

`* / div mod and and-then`
: multiplication operators

`++ + - or or-else`
: addition operators

< > = <= >= <> in
  relational operators

|| sequence operator

Most of these symbols have their usual meaning; a closer description follows below. All binary operators are left-associative, with the exception of ^ and ! which associate to the right, and the relational operators which are nonassociative. Application takes precedence over all these operations except the quotation operators; hence `sqrt X^3` denotes `(sqrt X)^3` and *not* `sqrt (X^3)`. The quotation operators have the highest possible precedence, and hence bind stronger than even applications. Parentheses are used to group expressions and overwrite default precedences and associativity as usual. But note that extra parentheses around a *primary* expression (identifier, operator symbol, number, string, list, tuple, quoted or parenthesized expression, i.e., anything which binds stronger than function application) turns the expression into a 1-tuple, see Section 6.3 [Lists and Tuples], page 32. C programmers will also note that the logical operators have the same "wrong" precedence as in Pascal. Thus you should make sure that you always parenthesize relational expressions when combining them with logical connectives.

You should also note that unary minus *must* be parenthesized when appearing in an argument of a function application. For instance, although `foo X -Y` is a perfectly well-formed expression, it denotes `(foo X) - Y` and *not* `foo X (-Y)` as you might have intended by the spacing which is however ignored by the Q compiler. As already noted in Chapter 3 [Lexical Matters], page 19, unary minus in front of a number is special; it causes values such as `-2` to be interpreted as negative numbers rather than denoting an explicit application of the unary minus operator (an explicit application of unary minus can be denoted using the built-in `minus` symbol; see below). The rules for parenthesizing negative numbers are the same as those for unary minus; you have to write `foo X (-2)` instead of `foo X -2` (which denotes `(foo X) - 2`).

In the Q language, expressions involving operators are merely syntactic sugar for applications. By enclosing an operator in parentheses, you can turn it into an ordinary prefix function. For instance, `X+Y` is exactly the same as `(+) X Y`, and `not X` actually denotes the application `(not) X`. The built-in operators simply provide a convenient way for applying these operations to their arguments. Moreover, you can also turn a binary operator into a unary function by specifying either the left or the right operand. E.g., `(1/)` denotes the reciprocal and `(*2)` the doubling function. Such constructs are commonly called *operator sections*. Inside a section, the usual precedence and associativity rules apply. For instance, the `X+3` subterm in `(*(X+3))` *must* be parenthesized because `*` has a higher precedence than `+`, and hence the partial expression `(*X+3)` is not well-formed.

The `-` operator plays a somewhat awkward role in the syntax, since it is used to denote both unary and binary minus. Q adopts the convention that the notation `(-)` always denotes *binary* minus; unary minus may be denoted by the built-in `minus` function. Thus the expression `minus X` applies unary minus to `X`. Note that this construct always denotes an explicit application of the unary minus operation. For instance, `minus 5` denotes the application of unary minus to the integer `5`, while `-5` is a negative integer.

We also remark that the construct `(-X)` is *not* an operator section, but a parenthesized expression involving unary minus. The easiest way to construct an operator section which subtracts a given value from its argument is to formulate the function using the addition operator as in `(+(-X))`.

6.4.1 Quotation Operators

The `'` (quote), `` ` `` (backquote) and `~` (tilde) operators are used to deal with so-called *special forms*. The quote operator quotes an expression as a literal; it is a constructor symbol and hence becomes part of the quoted expression. The backquote and tilde operators are used to "splice" and "force" subterms in an expression. We postpone a discussion of these operations until Chapter 9 [Special Forms], page 69.

6.4.2 Arithmetic Operators

The operators `+`, `-`, `*`, `/` denote the sum, the difference, the product and the quotient of two numeric values, respectively. As already noted, `-` is also used as unary minus. The operators `div` and `mod` denote integer quotient and modulo, respectively; they only apply to integers. The `^` operator denotes exponentiation, as defined by `X^Y = exp (ln X*Y)`; it always returns a floating point value. (If `X<0` then `X^Y` is defined only if `Y` is an integer. `0^0` is left undefined as well, so if you would like to have that `0^0 = 1` then you must add corresponding equations yourself. Also note that the `complex.q` standard library module extends the built-in definition of the exponentiation operator to handle the case that `X<0` with general exponent; see Section 11.12 [Complex Numbers], page 109.)

The argument and corresponding result types of these operations are summarized in the following table (`Int` denotes integer, `Float` floating point, and `Num` numeric (integer or floating point) values):

`+ - *`	`Int Int` → `Int`
	`Int Float` → `Float`
	`Float Int` → `Float`
	`Float Float` → `Float`

`/ ^`	`Num Num` → `Float`
`div mod`	`Int Int` → `Int`
`- (unary)`	`Int` → `Int` `Float` → `Float`

6.4.3 Relational Operators

The operators `<`, `>`, `=`, `<=`, `>=`, `<>` are binary predicates meaning "less", "greater", "equal", "less or equal", "greater or equal" and "not equal", respectively. The built-in definition of these operations only applies to numbers, strings and truth values. All relational operators return truth values (`true`, `false`) as results. Strings are compared lexicographically, on the basis of the local character encoding. Truth values are ordered by `false < true`.

If you would like to compare other types of values than the basic objects mentioned above, normally you will have to provide suitable definitions yourself. For instance, you might wish to extend the equality operation to other built-in and user-defined data structures such as lists, trees, etc., by overloading the = operator accordingly. The following equations implement an equality predicate on lists (the parentheses on the left-hand side are necessary to prevent the equality operation to be interpreted as the equal sign separating left-hand and right-hand side of an equation):

```
([] = [])               = true;
([] = [Y|Ys])           = false;
([X|Xs] = [])           = false;
([X|Xs] = [Y|Ys])       = (X=Y) and then (Xs=Ys);
```

Rules for other comparison operators (<>, <=, etc.) could be added in the same fashion. Actually, the standard library provides corresponding definitions; see Chapter 11 [The Standard Library], page 93.

A special case arises with types consisting of only nullary constructor symbols declared with `const`, so-called *enumeration types*, see Chapter 8 [Types], page 61. The values of such a type can always be compared with the relational operators, using the order in which they are listed in the type declaration. (A special case of this is the order of the built-in truth values.) For instance, assume that the `Day` type is declared as follows:

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

Then the listed constants will be ordered as `sun < mon < ... < sat`.

Besides the equality predicate, the Q language also provides a notion of "syntactic" equality which applies to all kinds of expressions; see Section 7.2 [Non-Linear Equations], page 45, for details.

The Q language provides one other relational operator, the `in` operator, which does not have a predefined meaning and hence can be employed by the programmer for his own purposes. In the standard library, the `in` operator is used to form list and stream comprehensions, see Section 11.8 [List Comprehensions], page 106, and Section 11.9 [Streams], page 106.

6.4.4 Logical and Bit Operators

The logical operations `not`, `and`, `or` denote logical negation, conjunction and disjunction, respectively. These operators take truth values as their arguments. They are defined in a straightforward manner:

```
not true                = false;
not false               = true;

true and true           = true;
true and false          = false;
false and true          = false;
false and false         = false;

true or true            = true;
true or false           = true;
false or true           = true;
false or false          = false;
```

Like most other programming languages, Q also has logical connectives for the *short-circuit evaluation* of logical expressions, which are denoted by the operators `and then` and `or else`. These operations are actually implemented as “special forms” which evaluate their arguments from left to right only as far as required to determine the value of the expression (cf. Chapter 9 [Special Forms], page 69). They are defined by the following built-in equations:

```
true and then X         = X;
false and then X        = false;

false or else X         = X;
true or else X          = true;
```

The first operand is always evaluated. Depending on its value, the second operand may not be evaluated at all. For instance, if `X` evaluates to `false`, then `X and then Y` immediately reduces to `false`, without ever having to evaluate the second argument `Y`. On the other hand, if `X` evaluates to `true`, then `Y` is evaluated and returned as the value of the expression. The `or else` operation works analogously.

One reason for using short-circuit evaluation is efficiency: prevent unnecessary computations when an initial part of a logical expression already determines the value of the entire expression. Furthermore, short-circuit evaluation is sometimes essential to check a condition before evaluating an expression depending on the validity of this condition. For instance:

```
(X <> 0) and then (foo (1/X) > 0)
```

You should also note that, according to the above definitions, `X and then Y` and `X or else Y` are *always* reduced if `X` is a truth value, even if the second operand `Y` does *not* evaluate to a truth value. This may look a bit weird at first, but it is in fact the most reasonable way to implement short-circuit evaluation in the Q language, since Q uses dynamic typing, and hence the type of an expression is only known *after* it has been evaluated. In fact, this “feature” can be quite useful at times. For instance, you can also use `and then` to write down a simple kind of conditional expression like

```
check X and then bar X
```

where `bar X` is the value you wish to compute, but only if the condition `check X` holds.

The Q interpreter also uses the operators `not`, `and` and `or` to denote bitwise logical operations on integers. Thus, `not X` denotes the one’s complement of an integer `X`, and `X and Y` and `X or Y` the bitwise logical conjunction and disjunction of integers `X` and `Y`, respectively. These operations behave as if negative integers were represented in two’s complement (although GMP actually uses a sign-magnitude representation). This means that for each integer `X`, `-X = not X + 1`, or, equivalently, `not X = -X-1`. For instance:

```
==> 17 and not 13
16

==> 17 or not 13
-13
```

```
==> not _
12
```

These results become clear when considering that 17 has bits 0 (=1) and 4 (=16) on (and all other bits off), while not 13 has bits 0 (=1), 2 (=4) and 3 (=8) off (and all other bits on). Thus, writing these numbers as unsigned bit sequences with most significant bits first, where ...1 denotes an infinite sequence of leading 1's, we have:

```
17 and not 13 =
10001 and not 1101 = 10001 and ...10010 = 10000
= 16

17 or not 13 =
10001 or not 1101 = 10001 or ...10010 = ...10011 (= not 1100 = not 12)
= -13
```

Together with the built-in bit shift operations, the bitwise logical operations can also be used to implement "bitsets", i.e., sets of nonnegative integers represented by bit patterns. See Section 10.1 [Arithmetic Functions], page 77, for details.

In case you are wondering about "exclusive or:" While this operation is not provided as a builtin, you can easily define it yourself as follows:

```
xor X Y                 = (X or Y) and not (X and Y);
```

6.4.5 String/List/Tuple Operators

The ++ operator denotes concatenation, # is the length or size operator, and the subscript operator ! is used for indexing. These operations work consistently on strings, lists and tuples. For instance:

```
"abc"++"xy"             ⇒ "abcxy"
[a,b,c]++[x,y]          ⇒ [a,b,c,x,y]
(a,b,c)++(x,y)          ⇒ (a,b,c,x,y)

#"abc"                  ⇒ 3
#[a,b,c]                ⇒ 3
#(a,b,c)                ⇒ 3

"abc"!1                 ⇒ "b"
[a,b,c]!1               ⇒ b
(a,b,c)!1               ⇒ b
```

Note that indexing with the subscript operator starts at zero, s.t. X!0 and X!(#X-1) denote the first and last member of a string, list or tuple, respectively. We also remark that since ! is right-associative, double subscripts may have to be parenthesized. For instance, compare (X!I)!J and X!I!J=X!(I!J).

It should also be noted that all list and tuple operators check that their first argument is a well-formed list or tuple value. However, the second argument of the ++ operator may in fact be any value. List concatenation just replaces the empty sublist marking the end of its first list argument with the second argument, as if it was defined by the following equations:

```
[]++Ys                  = Ys;
[X|Xs]++Ys              = [X|Xs++Ys];
```

Hence we have that, e.g., `[]++1` ⇒ `1` and `[1,2]++3` ⇒ `[1,2|3]`, which may look odd, but is consistent with the above equations. Tuple concatenation works in an analogous manner.

6.4.6 The Sequence Operator

The sequence operator lets you evaluate sequences of expressions in a given order. The value of the sequence is given by the rightmost expression. That is,

```
X || Y                  ⇒ Y.
```

The sole purpose of this construct is to allow operations with side-effects (such as I/O functions) to be carried out sequentially. A typical example is

```
writes "Input: " || reads
```

which prints a prompt on the terminal and then reads in a string. (The built-in functions `writes` and `reads` are described in Chapter 10 [Built-In Functions], page 77.)

7 Equations and Expression Evaluation

> "Computational processes are abstract beings that inhabit computers." [Abelson/Sussman 1985, p.1]

At its core, any programming language faces the programmer with three distinct abstract notions [Abelson/Sussman 1985]: the entities which are to be manipulated, usually referred to as "data", the computational "processes" which carry out those manipulations, and the description of these processes by finite collections of computational rules, commonly called "programs".

As in other functional programming languages, all computations performed by Q scripts are expression evaluations: The user specifies an expression to be evaluated, the interpreter computes the value of that expression and prints it as the result of the computation.

Having described the "data" manipulated by Q scripts, expressions, we now turn to the computational processes carried out on those expressions, expression evaluations, and their description by collections of computational rules, equations.

7.1 Equations

In the Q language, there is in fact no such thing like a "function definition" in conventional languages. Rather, definitions take the form of *equations* which describe how a given expression, the left-hand side of the equation, can be transformed into another one, the corresponding right-hand side. Note the difference between this interpretation and common mathematical usage in which an equation is usually something to be "solved". Here we are actually dealing with "rewriting rules", i.e., the equations are oriented from left to right. The syntax of equations is captured by the following grammar rules:

```
definition              : expression '=' expression qualifiers ';'
                          {'=' expression qualifiers ';'}
qualifiers              : {qualifier}
qualifier               : condition
                        | where-clause
condition               : 'if' expression
                        | 'otherwise'
where-clause            : 'where' expression '=' expression
                          {',' expression '=' expression}
```

Variables occurring on the left-hand side of an equation are taken to be universally quantified. They play the role of *formal parameters* in conventional programming languages, which are "bound" to their corresponding values when the equation is applied to an actual expression value. For instance, suppose we wish to define a function `sqr` which squares its argument. We know that to square something we simply multiply it with itself. The corresponding equation is:

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

Here the left-hand side is simply the application of the function symbol `sqr` to an argument value `X` which stands for any actual value we might apply the `sqr` function to.

An equation may also contain a collection of conditions and "local" variable definitions (these two kinds of supplementary elements are also called *qualifiers* in the Q jargon), and multiple right-hand sides (which are interpreted as a collection of equations for the same left-hand side). An empty condition may be denoted with the keyword `otherwise`, which is a convenient means to indicate the "default case" of a definition. For instance, the following equations define the factorial function:

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

The `otherwise` keyword is nothing but syntactic sugar; it can always be omitted. However, it tends to improve the readability of definitions like the one above.

In difference to function definitions in conventional programming languages, the left-hand side of an equation may contain constant and structured argument *patterns* which are to be *matched* against the actual parameters of a function. For instance, the following equations, which involve constant values on the left-hand side, define the Fibonacci function:

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

As an example for structured argument patterns, operations to extract the "head" (first element, the "car" in Lisp terminology) and "tail" (list of remaining elements, the "cdr") of a list can easily be defined as follows:

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

Note that in the above example we actually only need one of the involved component values. In such a case it is possible to employ the *anonymous variable* '`_`' as a placeholder for any component value we do not care for. For instance:

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

The anonymous variable can only be used on the left-hand side of an equation, i.e., it is illegal to refer to the "value" of the anonymous variable on the right-hand side because in fact this variable *has* no value. (In the interpreter this is different; there the anonymous variable can be used to denote the value of the most recent evaluation result, see Section B.2 [Command Language], page 181.) We also remark that multiple instances of the anonymous variable are matched independently from each other, therefore an equation like

```
foo _ _                 = 0;
```

will be matched for *any* combination of arguments.

As another example, here is the definition of a function `sum` which computes the sum of a list of numbers:

```
sum [X|Xs]              = X+sum Xs;
sum []                  = 0;
```

(The above definition is the Q equivalent for what Lisp programmers call "cdr'ing down a list".)

In the same fashion, we can use tuples and applications to form expression patterns on the left-hand side of an equation. For instance, we can define a binary tree insertion operation as follows:

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

Here the `nil` and `bin` symbols act as *constructors* which are used to build a tree-like data structure. Applications involving such symbols are usually normal form expressions without any "defining" equations. In such a case they can also be declared `const` (cf. Chapter 5 [Declarations], page 25), which means that the compiler will enforce that these symbols cannot be "redefined" by means of an equation:

```
const nil, bin X T1 T2;
```

After this declaration, equations like the following will be invalid:

```
nil                 = ...; // WRONG!
bin X T1 T2         = ...; // WRONG!
```

The same applies to the built-in constant symbols `true` and `false` which are predeclared as `const`, and also to built-in constants, i.e., numbers, strings, files, lists and tuples. The general rule is that no constant value is allowed as the left-hand side, or the leftmost ("function") part of an application which forms the left-hand side of an equation.

Note that since expressions involving operators are nothing but syntactic sugar for applications, they can occur on the left-hand side of an equation as well (with the exception of the `'` operator which is a constructor symbol, see Chapter 9 [Special Forms], page 69). For instance, here is a collection of algebraic simplification rules which cause any expression involving only `+` and `*` to be converted into a "sum of products" form:

```
// distributivity laws:
(X+Y)*Z                 = X*Y+X*Z;
X*(Y+Z)                 = X*Y+X*Z;
// associativity laws:
X+(Y+Z)                 = (X+Y)+Z;
X*(Y*Z)                 = (X*Y)*Z;
```

A word of caution: if the `=` operator occurs at the toplevel of the left- or right-hand side of an equation, it must be parenthesized. This is necessary to prevent the `=` operator to be confused with the `=` symbol used to separate both sides of an equation. (The same also applies to variable definitions.) E.g., you should write something like

```
(X=X)                   = true;
```

instead of

```
X=X                     = true; // WRONG!
```

Although almost any expression can form the left-hand side of an equation, you should be aware that an equation will only be *applicable* if the pattern you specify can actually occur in the course of an expression evaluation. In particular, since the Q interpreter evaluates expressions in a leftmost-innermost fashion, you should make sure that argument patterns match *normal form expressions* (cf. Section 7.6 [Normal Forms and Reduction Strategy], page 50), unless the function to be defined is a special form (cf. Chapter 9 [Special Forms], page 69). Otherwise the equation may be useless, albeit syntactically correct. The compiler cannot verify these conditions, so they are at your own responsibility.

Using equations we can also easily define "higher order" functions which take functions as arguments or return them as results. This is made possible by the fact that function application is an explicit operation. For instance, as a generalization of the `sum` function above, we can implement an operation which applies *any* "binary" function `F` to a list, starting from a given initial value `A`:

```
foldr F A []            = A;
foldr F A [X|Xs]        = F X (foldr F A Xs);
```

The `foldr` ("fold-right") function takes as its arguments a function `F`, an initial value `A`, and a list `[X1,...,Xn]`, and computes the value

```
F X1 (F X2 (... (F Xn A)...)).
```

(The name of this function comes from the fact that the recursive applications of `F` are parenthesized to the right.) Now we can easily define the `sum` function in terms of `foldr` as follows:

```
sum                     = foldr (+) 0;
```

There is another version of the fold operation, which accumulates results from left to right rather than from right to left:

```
foldl F A []            = A;
foldl F A [X|Xs]        = foldl F (F A X) Xs;
```

We remark that although both functions work in linear time (with respect to the size of the input list), the `foldl` function actually is more efficient than `foldr` in terms of stack space requirements since its definition is "tail-recursive" (cf. Section 7.10 [Tail Recursion], page 57).

As another example, here is the definition of the function `map` which maps a given function `F` to each member of a list `Xs`:

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

Finally, we also give an example for a function which *returns* another function. It is a toy implementation of the lambda calculus, based on the combinatorial calculus. (This definition will only work with simple kinds of expressions. Please refer to the standard library script `lambda.q` described in Section 11.7 [Lambda Calculus], page 103, for a much more comprehensive implementation which also properly handles the obscure cases. Also note that the `lambda` function is declared as a *special form* which prevents its arguments from being evaluated. This will be explained in Chapter 9 [Special Forms], page 69.)

```
special lambda X Y;

lambda X X              = i;
lambda X (Y Z)          = s (lambda X Y) (lambda X Z);
```

```
lambda X Y              = k Y otherwise;

/* combinator rules */

i X                     = X;
k X Y                   = X;
s X Y Z                 = X Z (Y Z);
```

For instance, you might wish to try the following:

```
==> lambda X (2*X)
s (s (k (*)) (k 2)) i

==> _ 4
8
```

7.2 Non-Linear Equations

If a variable has multiple occurrences on the left-hand side of a rule, each occurrence must be matched to the same value. In term rewriting theory, such rules are called "non-linear" ("non-left-linear", to be more precise). The following definition implements an operation `uniq` which removes adjacent duplicates from a list:

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

It is important to notice the difference between the syntactical notion of equality which plays a role in definitions like the one above, and the "semantic" equality relation defined by the `=` operator (cf. Section 6.4.3 [Relational Operators], page 35). Non-linear equations and equations involving constants, such as

```
uniq [X,X|Xs]           = uniq [X|Xs];
fib 0                   = 0;
```

call for the verification of syntactic identities. While two expressions are considered syntactically equal only if they print out the same in the interpreter, the meaning of the `=` operation depends on built-in rules and equations specified by the programmer. For instance, two different instances of the expression `0` always denote the same object. However, since `0` is an integer and `0.0` a floating point number, these two expressions are syntactically different. Nevertheless, the `=` operator allows to compare integers and floating point numbers. Indeed it asserts that `0` and `0.0` are "equal".

If you would like to have an explicit syntactic equality operation, you can easily define it yourself as follows:

```
eq X X                  = true;
eq _ _                  = false otherwise;
```

In fact, this definition is provided in the `stdlib.q` script, see Chapter 11 [The Standard Library], page 93.

You should note that the strict separation of syntactic and semantic equality is actually quite useful. First of all, it allows the `=` operation to be treated in a manner consistent with other comparison operators such as `<` and `>`. Secondly, it allows you to tailor the definition of `=` to your application. For instance, an application might call for a set data structure, and you would like to be able to test whether two expressions of a certain kind represent the same set. Now there are fairly sophisticated data structures for representing sets efficiently, but almost none of them allow you to decide whether two objects represent the same set simply on the basis of syntactic identity. Instead, you will have to provide an explicit `=` operation which tests whether two set objects contain the same elements.

7.3 Free Variables

On the right-hand side of an equation (as well as in the condition part and the right-hand sides of local definitions), we distinguish between *bound* variables, i.e., unqualified variable symbols which are introduced in the left-hand side of an equation (or the left-hand side of a local definition, see Section 7.4 [Local Variables], page 47), and *free* variables which are *not* bound by any of the left-hand side patterns. Free variables can also be declared explicitly using a `var` declaration (cf. Chapter 5 [Declarations], page 25). They are effectively treated as (parameterless) function symbols, except that they may be assigned a value by means of a *variable definition*. Variable definitions have the following syntax:

```
definition              : 'def' expression '=' expression
                          {',' expression '=' expression} ';'
                        | 'undef' identifier {',' identifier} ';'
```

For instance, in the following equation,

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

the variable `C` occurs free on the right-hand side and we have:

```
==> foo 23
C*23
```

We can assign a value to the variable as follows:

```
def C = 2;
```

The same effect can also be achieved by using the `def` command in the interpreter:

```
==> def C = 2
```

After this definition we have:

```
==> foo 23
46
```

Free variables are useful if you have a script which repeatedly refers to the same value whose calculation may be costly or involve functions with side-effects (such as the built-in `fopen` function, see Chapter 10 [Built-In Functions], page 77). By assigning such a value to a variable, you can refer to it without having to recompute the value each time it is required, as would be the case with a corresponding function definition. Note that free variable definitions cannot be changed inside an equation, and therefore the use of free variables does not compromise referential transparency. This is in contrast to other functional programming languages such as Lisp which allow you to modify the value assigned to a variable in a function definition. In Q, you can count on that the value of a free variable does *not* change during a single expression evaluation, although it may be altered between different evaluations of the same expression.

The right-hand side of a variable definition may be any Q expression, as described in Chapter 6 [Expressions], page 29. The expression is evaluated *once*, at the time the script is loaded, and is then matched against the left-hand side of the definition, binding variables to the corresponding component values of the right-hand side value. In the special case that the left-hand side is a single variable, it is simply bound to the right-hand side value. Whenever a defined variable occurs "free" on the right-hand side of an equation or variable definition, it will then be replaced by the value of the assigned expression. For instance, the following definition assigns the value `exp 1.0 = 2.71828...` to the (explicitly declared) free variable `e`:

```
var e;
def e = exp 1.0;
```

Variable definitions are carried out in the order in which they occur in a script, s.t. a definition may refer to the values of variables assigned in previous definitions. Unless a variable has been declared `const`, it is also possible to override assignments made in previous definitions, and to undefine a variable. Multiple definitions can be performed with a single `def` statement. For instance:

```
def N = 99, K = N, N = 2*K+1, M = N;
undef N, K;
```

As already mentioned, the value assigned to a variable may also be changed with `def` and `undef` statements in the interpreter, see Section B.2 [Command Language], page 181.

Free variable definitions can also involve a pattern on the left-hand side which is to be matched against the supplied value. E.g.,

```
def (X,Y) = foo Z;
```

causes `foo Z` to be evaluated and to be matched against `(X,Y)`. The interpreter then defines the free variables `X` and `Y` accordingly. Pattern matching is performed in the same manner as for the left-hand side of an equation. However, an error is reported if the match fails.

7.4 Local Variables

As already mentioned, an equation may also contain one or more auxiliary *variable definitions* a.k.a. `where` clauses, which bind additional variables to their corresponding values. In difference to the global definitions of "free" variables discussed in Section 7.3 [Free Variables], page 46, the definitions in `where` clauses are "local" to the rule in which they occur, meaning that the values of these variables are only available in the rule they are defined in. As in the case of free variable definitions, the left-hand side of each `where` clause can be an arbitrary expression pattern which is to be matched against the value computed from the corresponding right-hand side. The variables occuring on the left-hand side of the definition will then be bound to their corresponding values, thereby becoming "bound" variables in the context of the rule, in addition to the variables occuring on the left-hand side of the equation. Local definitions also act as additional conditions in that the rule will only be applicable if all left-hand sides in `where` clauses match their corresponding right-hand side values (this is guaranteed, of course, if each left-hand side is a single variable).

Local definitions are useful if the right-hand side of a rule repeatedly refers to the same (maybe complicated) subexpression. You can avoid the recalculation of such a value be assigning it to a variable, and referring to the variable in the right-hand side. For instance:

```
foo X                   = bar Y Z
                          where Y = baz X, Z = qux Y;
```

As already mentioned, pattern-matching definitions are permitted as well (and the anonymous variable also works as usual). A failing match causes the entire rule to fail. For instance,

```
foo Z                   = bar X Z where [X|_] = Z;
```

works like

```
foo [X|Xs]              = bar X [X|Xs];
```

but avoids repeating the list value on the right.

Note that if a variable is bound by the left-hand side or a local definition of a rule, then a global variable of the same name will be “shadowed” within the rule. However, you can still access the value of the shadowed variable with a qualified identifier. For instance, with the following definitions, `foo 2` reduces to `bar 2*99`:

```
/* this is the script foo.q */
def FOO                 = 99;
foo X                   = FOO*foo::FOO where FOO = bar X;
```

The variable bindings in a single `where` clause are performed in the given order, and each definition may refer to all left-hand side variables of the rule, as well as all variables introduced in earlier definitions. Each use of a variable refers to its most recent definition. It is also possible to have multiple `where` clauses, as in:

```
foo X = bar Y where Y = baz Z where Z = qux U where U = quux X;
```

Note that here the definitions will be processed from *right to left*. Such constructs are convenient if there is a series of values to be computed in which one value depends on the next one. Using multiple `where` clauses, you start off with the definitions immediately required by the right-hand side of the rule, then add another `where` clause for the variables introduced in the first clause, etc. The above definition is actually equivalent to the following:

```
foo X = bar Y where U = quux X, Z = qux U, Y = baz Z;
```

It is also possible to mix `where` clauses with conditions, as in:

```
foo X = bar Y if qux Y where Y = baz Z if quux Z where Z = bazz X;
```

Again, the different qualifiers are actually considered from right to left; the rule will be aborted as soon as the first qualifier fails (i.e., a condition evaluates to `false` or the left-hand side of a definition does not match the corresponding right-hand side), so that no time is wasted with definitions which are not needed anyway.

7.5 Type Guards

The Q language allows you to restrict the set of patterns which can be matched by a left-hand side variable. This is done by augmenting any left-hand side occurrence of the variable with a *type guard*, using the notation:

variable:*type*

For instance, to declare a type `BinTree` which comprises the constructor patterns `nil` and `bin X T1 T2`, the following declaration might be used (cf. Chapter 5 [Declarations], page 25):

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

This declaration introduces two function symbols `nil` and `bin`, just as if they were declared by:

```
public const nil, bin X T1 T2;
```

However, it also assigns these symbols to the type symbol `BinTree`. The type symbol can then be used as a guard on variables to ensure that a variable is only matched to objects of the given type. E.g., the following equation will only apply if the argument to the `foo` function has the form `nil` or `bin X T1 T2` for some subexpressions `X`, `T1` and `T2`:

```
foo T:BinTree           = ...;
```

This makes it possible to avoid explicit argument patterns like

```
foo nil                 = ...;
foo (bin X T1 T2)       = ...;
```

in cases in which the component values are not actually required. Also, it allows you to define operations on a given type without actually referring to the constructor symbols of the type, and thereby makes it possible to avoid dependencies on implementation details. You can even "hide" the constructor symbols of a type by declaring them `private`, which makes the symbols accessible only to the script which declares the type they belong to. For instance:

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

The Q language allows you to construct a hierarchy of types by making one type a subtype of another. This is done by specifying a supertype following the type identifier in a type declaration. For instance, to make `BinTree` a subtype of the type `SearchTree`, `BinTree` would be declared as follows:

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

Now a variable of type `SearchTree` will not only match `SearchTree`'s, but also any object of its subtype `BinTree`. In fact, we might have declared `SearchTree` as an "abstract" type which does not provide any constructors at all:

```
type SearchTree;
```

Abstract supertypes like `SearchTree` are useful for factoring out generic operations which are shared by different subtypes; see Chapter 8 [Types], page 61, for examples, and for a more detailed discussion of the type guard concept.

The Q language has nine predefined type symbols which can be used to distinguish the corresponding types of objects built into the language: `Int`, `Float`, `Num` (which is the supertype for both `Int` and `Float`), `String`, `Char` (the subtype of `String` denoting the single-character strings), `File`, `List`, `Tuple` and `Bool`. (Besides these, there are also two types `Exception` and `SysException` for dealing with exception values, cf. Section 10.6 [Exception Handling], page 86.) For instance, an operation `isstr` which determines whether its argument is a string can be defined as:

```
isstr _:String          = true;
isstr _                 = false otherwise;
```

7.6 Normal Forms and Reduction Strategy

The process in which equations are applied in the Q interpreter to evaluate a given expression is fairly straightforward. In fact, it corresponds to the way in which we manipulate algebraic formulae, and we learned how to do that in high school.

We say that an equation `L=R` is *applicable* to a given expression `X`, if we can substitute actual values for the variables occurring in `L` s.t. we obtain `X`. This is also expressed as "`L` matches `X`" or "`X` is an instance of `L`". In this case, we can replace `X` by `Y`, where `Y` is the expression obtained from `R` by replacing the variables occurring in `L` by their corresponding values. We shall call such a replacement step a *reduction*, and write it down as `X` ⇒ `Y`.

For instance, given the rule

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

we have that

```
sqr 2                   ⇒ 2*2
```

since the expression `sqr 2` matches the left-hand side `sqr X`, with `2` being substituted for the variable `X`. This works with compound expressions as well, even if they involve variables themselves. For instance:

```
sqr (X+1)               ⇒ (X+1)*(X+1).
```

Here, the compound expression `(X+1)` has been substituted for the variable `X`.

Equations are usually applied in the context of a larger expression to be evaluated. For instance, we have that

```
sqr 2 + 2               ⇒ 2*2+2
```

by the application of the above definition of `sqr` to the subexpression `sqr 2` of `sqr 2 + 2`.

Before we proceed, a remark on the role of the built-in operations of the Q language is in order. Primitive operations, such as the built-in `+` and `*` operators described in Section 6.4 [Built-In Operators], page 33, do not require any equations specified by the programmer. Instead, they may be thought of as being predefined by a large set of built-in equations. Each application of a built-in rule also counts as a single reduction. For instance, we have that

```
2*2                     ⇒ 4
```

by the built-in rule which handles the multiplication of integers.

Built-in rules take priority over any equations specified by the programmer. However, it is possible to extend the definition of a primitive operation with equations supplied by the programmer. A corresponding example has already been given in Section 7.1 [Equations], page 41. By refining built-in definitions accordingly, you can also treat “exceptions” in built-in operations; see Section 2.5 [Runtime Errors], page 16, for an example.

In order to complete the evaluation of a given expression, we have to repeatedly perform single reductions until no further equations (built-in or user-defined) apply. We then say that the resulting expression is in *normal form*, and interpret it as the *value* of the original expression. For instance, the following reduction sequence leads to the normal form (i.e., value) `6` for the target expression `sqr 2 + 2` (we still assume the definition of `sqr` introduced above):

```
sqr 2 + 2  ⇒  2*2+2  ⇒  4+2  ⇒  6.
```

Normal forms do not necessarily exist. Just as in conventional programming languages, the evaluation may go into an endless loop and never return an answer. Even if a normal form exists, it need not be unique – it may depend on which equations are used in the reduction process and in what order. The problem is that at any point in the reduction process there may be more than one “reducible” subexpression, and even if there is only one such expression (termed a *redex*), there may be more than one applicable equation.

To illustrate this kind of problem, consider the definition of the `sqr` function from above, and the target expression `sqr (1+1)`. This expression admits three distinct reduction sequences:

```
sqr (1+1)  ⇒  sqr 2  ⇒  2*2  ⇒  4
sqr (1+1)  ⇒  (1+1)*(1+1)  ⇒  2*(1+1)  ⇒  2*2  ⇒ 4
sqr (1+1)  ⇒  (1+1)*(1+1)  ⇒  (1+1)*2  ⇒  2*2  ⇒ 4.
```

The second and third reduction sequence appear to be almost the same, but the first reduction sequence makes clear that it matters whether we reduce the subexpression `(1+1)` before applying the definition of `sqr` or not. In the present example, the order in which reductions are performed only affects the number of required reduction steps, but it is easy to construct other systems of equations in which both the termination of a normal form evaluation and the computed normal forms actually depend on the evaluation order.

Non-uniqueness of normal forms does not necessarily mean that there must be two or more rules being applicable to a given expression. Ambiguities can also arise when an equation “overlaps” with other equations or with itself, as in the following example:

```
foo (foo X)             = bar X;
```

This system consisting of a single equation is terminating, but it has two distinct normal forms for the expression `foo (foo (foo X))`, namely `foo (bar X)` and `bar (foo X)`.

Indeed, the termination of rewrite systems and the uniqueness of normal forms are important issues in the theory of term rewrite systems [Dershowitz/Jouannaud 1990]. The Q language simply avoids these problems by imposing a *reduction strategy* which makes the reduction process deterministic. The reduction strategy must specify unambiguously for each reduction step the redex that is to be reduced and the equation which is to be applied. For this purpose the Q interpreter applies the following rules:

- Expressions are evaluated in *applicative order*, that is, from left to right, innermost expressions first.
- Equations are applied in the order in which they appear in a script. Built-in rules always take priority.

The leftmost-innermost strategy is common in many programming languages, and it also corresponds to the "natural" way in which people tend to carry out calculations manually. It means that at any point in the reduction process it is always the leftmost-innermost redex which is to be reduced. In particular, this implies that in an application of the form `F X` first `F` is evaluated, then `X`, before the value of `F` is applied to the value of `X`. Thus the first reduction sequence from above,

```
sqr (1+1)  ⇒ sqr 2  ⇒  2*2  ⇒  4
```

would be the one actually carried out in the interpreter in the course of the evaluation of `sqr (1+1)`.

The leftmost-innermost strategy resolves all ambiguities in the choice of redices during a normal form evaluation. The second disambiguating rule allows to decide which equation to apply if more than one equation is applicable to a given redex. As indicated, the default is to apply equations in the order in which they appear in the source script. This allows you to have equations with overlapping left-hand sides which naturally arise, e.g., in definitions involving "special case rules" such as the definition of the `fib` function in Section 7.1 [Equations], page 41:

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

We remark that the default textual order of equations can also be changed by using an appropriate "rule priority" directive. This is described in Section 7.8 [Rule Priorities], page 53.

The rule for applying equations is extended to the built-in equations of primitive operations by assuming – as already mentioned – that the built-in rules come "before" any equations specified by the programmer, and hence always take priority. We also remark that external functions, which are declared with the `extern` modifier and implemented in a C module, are treated in an analogous manner. In general, the interpreter will first try a built-in rule, then an extern rule, then the equations supplied in the script. (The treatment of extern functions is described in more detail in Appendix C [C Language Interface], page 191.)

7.7 Conditional Rules

Conditional rules are applied in the same manner as simple equations, only we also have to check the conditions of the rule. When the left-hand side of a conditional rule matches the target expression, we first evaluate the condition (with left-hand side variables being replaced as usual). This expression *must* evaluate to a truth value, otherwise we cannot decide whether the rule is applicable or not. (The Q interpreter generates a runtime error in such cases.) The rule is applicable only when the condition evaluates to `true`.

For instance, recall the definition of the factorial function:

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

The first rule is applicable only if `N>0` evaluates to `true`; if it evaluates to `false`, the second rule applies. Furthermore, if `N` happens to be incomparable with `0`, then a runtime error will occur.

Here's how this definition is applied to evaluate the expression `fac 3`:

```
fac 3                   ⇒ 3*fac (3-1)          ⇒ 3*fac 2
                        ⇒ 3*(2*fac (2-1))      ⇒ 3*(2*fac 1)
                        ⇒ 3*(2*(1*fac (1-1))) ⇒ 3*(2*(1*fac 0))
                        ⇒ 3*(2*(1*1))          ⇒ 3*(2*1)
                        ⇒ 3*2                  ⇒ 6.
```

Local variable definitions (cf. Section 7.4 [Local Variables], page 47) are treated in a similar fashion. We first evaluate the right-hand side of the definition and match it against the corresponding left-hand side. If the match fails, the rule is skipped. Otherwise, we bind variables accordingly and proceed with the next qualifier or the right-hand side of the rule.

7.8 Rule Priorities

We already discussed that the Q interpreter normally considers equations in the order in which they appear in a script. Hence, if all your equations go into a single script, you simply write down overlapping equations in the order in which they should be tried by the interpreter. However, if equations for the same expression pattern are scattered out over different modules, then the rule order also depends on the order in which the different modules are processed by the compiler. In the current implementation, the rules of each module are processed when the first import or include declaration for the module is encountered during a preorder traversal of the module graph starting at the main script, but you should not rely on this.

Hence it is usually a bad idea to have overlapping equations scattered out over different source files. However, in some situations this is hard to avoid. For instance, you might decide to put all "default rules" for certain expression patterns in a separate module. But if you simply import this module at the beginning of your main script then the default rules might override all the other equations.

To work around this, the Q language allows you to explicitly attach priorities to equations by means of a *priority declaration*, which has the following format:

```
declaration             : '@' ['+'|'-'] unsigned-number
```

This specifies the given number (which must be an integer, optionally signed with '`+`' or '`-`') as the *priority level* of the following equations. In the current implementation, priority levels must be 32 bit integers, i.e., in the range `-0x80000000..0x7fffffff`. Rules are ordered according to their priorities, highest priority first. Rules with the same priority are considered in the order in which they appear in the script.

The priority level set with a priority declaration applies only to the script in which the declaration is located. The default priority level, which is in effect up to the first @ declaration, is 0.

Note that even with priority declarations the built-in operations still take priority over all user-defined rules, i.e., those operations effectively have infinite priority. (The same applies to external functions, see Appendix C [C Language Interface], page 191.)

As a simple (and contrived) example for the use of priority declarations, consider the following script:

```
@-1
foo X                   = -1; // "default" rule

@0
foo X:Num               =  0; // "standard" rule

@+1
foo X:Int               =  1; // "special case" rule
```

The second equation is at the default priority level, 0. The first equation, although it comes before the second one, becomes a “fallback” rule by assigning it a low priority, -1. The last equation is put on a high priority level, +1, and hence overrides both preceding equations. The net effect is that the equations are actually ordered in reverse textual order. You can verify this in the interpreter:

```
==> foo 77
1

==> foo 77.0
0

==> foo ()
-1
```

Of course, this example is somewhat silly, because in a single script we can easily arrange equations in the correct textual order. However, if for some reason we had to put the three rules into three different modules, then using the given priority declaration ensures that the equations will always be applied in the correct order.

Priority levels should be used sparingly, if at all. Using low priorities to factor out a module with default rules can occasionally be quite useful, but overriding rules with high priorities is considered harmful and should be avoided whenever possible.

7.9 Performing Reductions on a Stack

The present and the following section contain material for the technically interested reader which, while not being strictly necessary to work successfully with the Q programming language, helps to understand the operation of the Q interpreter and to write better

Q scripts. You might wish to skim the following at first reading, and later return to it when you have become more proficient with the Q language and require some further background knowledge.

While the "rewriting model" of expression evaluation introduced in Section 7.6 [Normal Forms and Reduction Strategy], page 50, provides a nice conceptual model for the Q programmer, it is too inefficient to be a useful implementation model for the Q interpreter. In particular, it would be rather inefficient to actually construct the full expressions which occur as intermediate results in the course of a reduction sequence, and repeatedly scan these expressions for the leftmost-innermost redex. Instead, we can evaluate expressions by a simple recursive procedure as follows. The input to the algorithm is an expression `X` to be evaluated.

1. If `X` is an application, list or tuple, evaluate the parts of `X` recursively, from left to right. Proceed with step 2.
2. If a built-in (or extern) rule is applicable to `X`, invoke it and return the corresponding value.
3. Otherwise, determine the first equation `L=R` which is applicable to `X`. If there is no such rule, `X` is in normal form and is returned as the value of the expression. Otherwise we recursively evaluate `R` (with variables instantiated to their corresponding values) and return it as the value of the original expression.

(The above description of course leaves out many important details. For instance, the interpreter will also have to take care of `def`'ed symbols, and we will have to organize the search for an applicable equation. However, here we only want to give the general idea, and not a complete implementation of the interpreter.)

The above procedure can actually be implemented non-recursively if we keep track of the rules which are currently "active", together with the corresponding variable bindings. This information can easily be maintained on a stack. We illustrate this with a simple example. Recall the definition of the sum function:

```
sum [X|Xs]              = X+sum Xs;
sum []                  = 0;
```

The evaluation of `sum [1,2,3]` then proceeds as follows:

```
sum [1,2,3] ⇒ 1+sum [2,3]:
        sum [2,3] ⇒ 2+sum [3]:
                sum [3] ⇒ 3+sum []:
                        sum [] ⇒ 0
                        3+0 ⇒ 3
                sum [3] ⇒ 3
                2+3 ⇒ 5
        sum [2,3] ⇒ 5
        1+5 ⇒ 6
sum [1,2,3] ⇒ 6
```

Each new level of indentation indicates that we "suspend" the current rule (push it on the stack) and activate a new rule in order to evaluate some part of the right-hand side of the suspended rule. When all parts of the right-hand side have been evaluated, we return to the suspended rule (pop it from the stack) and actually perform the reduction. More precisely, we replace the left-hand side of the suspended rule by the result obtained by evaluating the right-hand side, which is already in normal form. (We will of course have to keep track of the results of intermediate reductions, which can be done on a second "evaluation" stack. When the evaluation of the right-hand side is finished, the corresponding result will be on top of the evaluation stack.)

If desired, we can easily reconstruct the "context" of a rule by following the chain of stacked rules starting from the current rule, and proceeding towards the bottom of the stack. For instance, when we perform the reduction

```
sum [] ⇒ 0
```

in the above example, the context of this rule is described by the following stacked rules:

```
sum [3] ⇒ 3+sum []
sum [2,3] ⇒ 2+sum [3]
sum [1,2,3] ⇒ 1+sum [2,3]
```

Thus, when `sum []` gets reduced to `0`, we have actually completed the following initial part of the reduction sequence:

```
sum [1,2,3] ⇒ 1+sum [2,3] ⇒ 1+(2+sum [3]) ⇒ 1+(2+(3+sum []))
            ⇒ 1+(2+(3+0))
```

(Note that we never actually constructed intermediate results like the expression `1+(2+(3+sum []))`. Rather, these expressions are represented implicitly by the contents of the stack.)

We can also apply the above procedure to qualified rules accordingly. When the interpreter comes to consider a conditional rule, it pushes it onto the stack as usual. However, it then first starts to evaluate the qualifying conditions and `where` clauses of the rule. When the value of a condition has been computed, we can check whether it holds. If the computed value is neither `true` nor `false`, the interpreter aborts the evaluation with a runtime error. If it is `true`, it proceeds by evaluating other qualifiers or the right-hand side. If it is `false`, however, it gives up on the current rule (pops it from the stack), and considers the next applicable equation. A similar approach is used to handle local variable definitions.

For instance, consider once more the definition of the factorial function:

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

The computation of `fac 3` then proceeds as follows (the ⋆ symbol indicates that the condition `N>0` has evaluated to false and hence the corresponding rule is abandoned):

```
fac 3 ⇒ 3*fac (3-1):
        3>0 ⇒ true
        3-1 ⇒ 2
        fac 2 ⇒ 2*fac (2-1):
                2>0 ⇒ true
                2-1 ⇒ 1
                fac 1 ⇒ 1*fac (1-1):
                        1>0 ⇒ true
```

```
                        1-1 ⇒ 0
                        fac 0 ⇒ 0*fac (0-1):
                                0>0 ⇒ false   ⋆
                        fac 0 ⇒ 1
                        1*1 ⇒ 1
                fac 1 ⇒ 1
                2*1 ⇒ 2
        fac 2 ⇒ 2
        3*2 ⇒ 6
fac 3 ⇒ 6
```

7.10 Tail Recursion

The above equations for the `sum` and `fac` functions are examples for recursive definitions. The computational processes generated from such definitions are characterized by chains of suspended rules in which the same rules are activated over and over again. For instance, an evaluation of `fac N` with `N>0` requires `N` recursive activations of the rule:

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

Hence, if `N` becomes very large, the recursive definition of `fac` is in danger of running out of stack space. In the following, we show how to avoid this defect by employing the technique of "tail-recursive programming".

It is a well-known fact that the factorial function also has an "iterative" implementation which can be executed in constant memory space. The idea is to maintain a "running product" `P` and a "counter" `I` which counts down from `N` to `1`. The iterative algorithm can be written down in a conventional programming language like Pascal as follows:

```
P := 1; I := N;
while I>0 do begin
  P := P*I;
  I := I-1;
end;
```

While the Q language does not provide any special looping constructs (and it also misses an assignment operation), there is an alternative definition of `fac` which takes up the idea of running products and counters and implements the factorial function in an iterative fashion:

```
fac N                   = fac2 1 N;
fac2 P I                = fac2 (P*I) (I-1) if I>0;
                        = P otherwise;
```

Here, the "state variables" `P` and `I` are implemented as arguments of an "auxiliary" function `fac2` which is invoked from `fac`. Again, this is a recursive definition; it requires `N` recursive applications of the rule

```
fac2 P I                = fac2 (P*I) (I-1) if I>0;
```

when `fac N` is computed. However, in difference to our previous definition of `fac`, the recursive rule is always applied "on top" of the target expression. Such rules are called *tail-recursive*. (The name "tail recursion" comes from the fact that the recursive application of fac2 is the last operation considered during the leftmost-innermost evaluation of the right-hand side.) For instance, the evaluation of fac 3 now proceeds as follows (abbreviating reductions by built-in rules for '-' and '*'):

```
fac 3 ⇒ fac2 1 3 ⇒ fac2 (1*3) (3-1)
      ⇒ fac2 3 2 ⇒ fac2 (3*2) (2-1)
      ⇒ fac2 6 1 ⇒ fac2 (6*1) (1-1)
      ⇒ fac2 6 0 ⇒ 6
```

Tail-recursive definitions can be employed for implementing all kinds of functions which can be computed on a machine with a fixed set of "registers" and no auxiliary memory [Abelson/Susssman 1985]. For instance, here is a tail-recursive implementation of the Fibonacci function:

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

(This definition also has the desirable side effect that it cuts down the exponential running time of the "naive" definition given in Section 7.1 [Equations], page 41, to linear.)

The Q interpreter employs a clever optimization trick, commonly known as *tail recursion optimization* (see e.g., [Steele/Sussman 1975]), to actually execute tail-recursive definitions within constant stack space. Hence no special looping constructs are required for implementing iterative algorithms efficiently in the Q language.

Assume that in our stack model of expression evaluation we are working on a reduction X ⇒ Y, and that we already recursively evaluated all parts of the right-hand side Y, but not Y itself. Furthermore, suppose that in order to evaluate Y we will have to apply the rule Y ⇒ Z next. Then, instead of keeping the rule X ⇒ Y suspended and evaluating Y using rule Y ⇒ Z recursively, we can also immediately perform the reduction X ⇒ Y and replace this rule with Y ⇒ Z on the stack. Thus, the new rule Y ⇒ Z will not require any additional stack space at all, but simply reuses the existing "activation record" for the X ⇒ Y rule. In other words, instead of invoking the Y ⇒ Z rule as a "subroutine", we effectively perform a kind of "goto" to the new rule. We also refer to this process as performing the *tail reduction* X ⇒ Y. The evaluation now proceeds as if we had been working on the rule Y ⇒ Z in the first place.

For instance, with the new definition of fac the evaluation of fac 3 would be carried out using only a single level of suspended rules (again, the ⋆ symbol signals abortion of a rule with failing qualifier):

```
fac 3 ⇒ fac2 1 3:
fac2 1 3 ⇒ fac2 (1*3) (3-1):
        3>0 ⇒ true
        1*3 ⇒ 3
        3-1 ⇒ 2
fac2 3 2 ⇒ fac2 (3*2) (2-1):
        2>0 ⇒ true
        3*2 ⇒ 6
        2-1 ⇒ 1
fac2 6 1 ⇒ fac2 (6*1) (1-1):
```

```
        1>0 ⇒ true
        6*1 ⇒ 6
        1-1 ⇒ 0
fac2 6 0 ⇒ fac2 (6*0) (0-1):
        0>0 ⇒ false                  ⋆
fac2 6 0 ⇒ 6
```

Besides the tail recursion optimization technique discussed above, the Q interpreter also automatically optimizes toplevel *sequences* (i.e., applications of the `||` operator which are not inside a nested subexpression) on the right-hand side of equations, s.t. basic imperative-style looping constructs can be executed in a tail-recursive fashion as well. Suppose, for instance, that we want to implement a `do` function which applies a function to each member of a list, like the `map` function discussed in Section 7.1 [Equations], page 41, but instead of collecting the results in an output list simply throws away the intermediate values and returns `()`. This is useful, of course, only if the function is evaluated solely for its side-effects, e.g., if we want to print out all elements of a list. A straightforward definition of `do` looks as follows:

```
do F []                 = ();
do F [X|Xs]             = F X || do F Xs;
```

Now this definition is *not* tail-recursive in the strict sense alluded to above, because the last application on the right-hand side of the second rule actually involves `(||)` and not the `do` function. (Recall that an infix expression like `X||Y` is nothing but syntact sugar for the function application `(||) X Y`.)

However, as a remedy, the Q interpreter actually implements toplevel sequences on the right-hand side of a rule by direct stack manipulation. That is, the first argument of a sequence is thrown away as soon as it has been computed. By these means, the interpreter, after having computed `F X`, effectively carries out the reduction `do F [X|Xs]` ⇒ `do F Xs` on which it can perform tail recursion optimization as usual. Thus the above definition actually works in constant stack space, as one might reasonably expect.

7.11 Error Handling

There are a few conditions which may force the Q interpreter to give up on a reduction sequence and abort the evaluation with a runtime error message. These conditions are listed below.

- The user interrupts the evaluation by hitting the interrupt key (usually `Ctl-C`). In this case it is possible to resume the evaluation in the debugger (cf. Appendix D [Debugging], page 199), provided that debugging has been enabled with the `break on` command (see Section B.2 [Command Language], page 181).
- The qualifying expression of a conditional rule evaluates to something which is not a truth value. In this case, if `break` is `on`, the debugger is invoked to inform the user of the equation which gave rise to the error condition.
- The attempt to allocate memory for runtime symbols, stack space or an expression on the heap fails (memory overflow).

- Stack overflow. (The interpreter actually allows you to make the stack as large as you like, provided you have enough main memory, but it is usually a good idea to provide a reasonable limit to catch infinite recursions before you run into a memory overflow condition.)

All of the above error conditions can also be caught and handled by the running script in any desired manner, see Section 10.6 [Exception Handling], page 86.

8 Types

Q uses *dynamic typing* like, e.g., Lisp or Smalltalk, as opposed to *statically-typed* languages such as Pascal, Eiffel or Haskell. In languages with static typing, a variable or function parameter can only hold a value which matches its prescribed type (which can be a "polymorphic" type in languages like Eiffel and Haskell, but still the type of the value is restricted). In dynamically typed languages, however, the actual value of a variable or function parameter is not known in advance. Consequently, in Q it is only possible to distinguish different types of objects – such as search trees, queues, arrays and the like – by selecting an appropriate set of constructor symbols for each type of object. This chapter discusses Q's notion of *type guards* which allows you to make the assignment of constructors to a type explicit, and to use this information in order to restrict the applicability of equations to objects of a given type.

8.1 Using Type Guards

As in any programming language, the careful design of application-dependent data structures is one of the main concerns when developing Q scripts which go beyond simple numeric, string and list processing. As a typical example for a non-list data structure which plays a prominent role in many Q scripts, let us consider binary search trees, which are a convenient means to implement sets, bags, dictionaries and the like. We will use this data structure as a running example throughout this chapter.

A typical choice of constructors for binary trees is the following:

```
public const nil, bin X T1 T2;
```

To make explicit the fact that `nil` and `bin` belong to the binary tree type, we can also use a type declaration which introduces the type symbol `BinTree`, as discussed in Chapter 5 [Declarations], page 25:

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

This declaration tells the interpreter that each expression of the form `nil` or `bin X T1 T2` should be considered as a member of the `BinTree` type. The type symbol can then be used as a *guard* on variables to ensure that a variable is only matched to objects of the given type, see Section 7.5 [Type Guards], page 49. E.g., the following rule employs such a type guard in order to restrict the argument of the `foo` function to `BinTree` objects:

```
foo T:BinTree           = ...;
```

This makes it possible to avoid explicit argument patterns like

```
foo nil                 = ...;
foo (bin X T1 T2)       = ...;
```

in cases in which the component values are not actually required. This can simplify matters a lot, in particular if multiple arguments have to be matched to a given type. Also, it is more efficient than checking the type of an object in the qualifier part of a rule by using a user-defined predicate, since the interpreter can use the type information right away in the pattern matching process.

Another important reason for preferring type guards over explicit argument patterns is the issue of "information hiding". With the former definition of the `foo` function above we do not make any explicit reference to the constructor patterns making up the `BinTree` data type. This makes it possible to treat `BinTree` as an *abstract data type* (*ADT*) which hides the underlying implementation details (in particular the constructors), while still being able to verify that the proper kind of object is supplied as an argument. Any access to objects of the ADT will then be implemented by referring to the appropriate operations supplied by the type. In fact, we can make the constructors private symbols which are only accessible to the script which implements the `BinTree` type:

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

As a concrete example, let us assume the standard search tree operations `insert T X` and `delete T X`, which insert an element `X` into a tree `T`, and remove it from the tree, respectively. These operations can be implemented as follows (see [Bird/Wadler 1988]):

```
public insert T X, delete T X;
private join T1 T2, init T, last T;

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 if X=Y;

delete nil Y            = nil;
delete (bin X T1 T2) Y  = bin X (delete T1 Y) T2 if X>Y;
                        = bin X T1 (delete T2 Y) if X<Y;
                        = join T1 T2 if X=Y;

join nil T2             = T2;
join T1 T2              = bin (last T1) (init T1) T2 otherwise;

init (bin X T1 nil)     = T1;
init (bin X T1 T2)      = bin X T1 (init T2) otherwise;

last (bin X T1 nil)     = X;
last (bin X T1 T2)      = last T2 otherwise;
```

(Note that the `last` and `init` operations return the last element of a binary tree, and a binary tree with the last element removed, respectively. The `join`, `init` and `last` functions are for internal use only, and can hence be implemented as private functions.)

Furthermore, we assume the following function `mkbintree` which constructs a binary tree from a list, and the function `members` which returns the list of elements stored in a tree (in ascending order):

```
public mkbintree Xs;
mkbintree Xs:List       = foldl insert nil Xs;

public members T;
members nil             = [];
```

```
members (bin X T1 T2)   = members T1 ++ [X|members T2];
```

(The definition of mkbintree employs the standard foldl operation, see Chapter 11 [The Standard Library], page 93.) We can use the interface operations of the BinTree ADT in order to implement the functions union and diff which add and remove all members of a tree to/from another tree:

```
public union T1 T2; // add elements of T2 to T1
union T1:BinTree T2:BinTree
                        = foldl insert T1 (members T2);

public diff T1 T2; // remove elements of T2 from T1
diff T1:BinTree T2:BinTree
                        = foldl delete T1 (members T2);
```

Observe that no explicit reference is made to the BinTree constructors; we only employ the public "interface" functions insert, delete and members of the BinTree ADT. This allows us to change the realization of the data structure without having to rewrite the definitions of union and diff. Still, the definitions of union and diff are "safe"; the BinTree type guard ensures that the arguments passed to union and diff are indeed BinTree objects capable of carrying out the requested operations.

8.2 Built-In and Enumeration Types

Type guards are also the only way for verifying that the actual value of a variable belongs to one of the built-in types integers, floating point numbers, strings and files, since there is no way for writing out all "constructors" for these kinds of objects – there are infinitely many (at least in theory). For this purpose, the type symbols Int, Float, String and File are predefined.

There also is a type named Num which, when used as a guard on variables, matches numeric (i.e., both integer and floating point) values, and the type Char which denotes the single-character strings. Technically, Num is the supertype of both Int and Float, and Char is a subtype of the String type; more about that in Section 8.3 [Sub- and Supertypes], page 64. Moreover, Char is also treated as an *enumeration type*, see below.

The built-in List type matches all list expressions of the form [] or [X|Xs]. This type is used to restrict the applicability of an equation to list arguments. For instance, the following equations define a function reverse which reverses a list:

```
reverse Xs:List         = foldl push [] Xs;
push Xs:List X          = [X|Xs];
```

The Tuple type is completely analogous: it matches tuples of arbitrary sizes, i.e., expressions of the form () and (X|Xs).

The predefined Bool type is a means to refer to objects which are truth values. It can be thought of as being predefined as follows:

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

Types like the built-in `Bool` type, which only consist of nullary `const` symbols, are also called *enumeration types*. You can easily define such types yourself, e.g.:

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

The Q language provides special support for enumeration types (including the built-in `Bool` type, and also the `Char` type), by means of the following operations:

- Members of an enumeration type can be compared using the relational operators (cf. Section 6.4.3 [Relational Operators], page 35), assuming the order in which the constants are listed in the type declaration.
- The `succ` and `pred` functions (see Section 10.1 [Arithmetic Functions], page 77) produce the successor and the predecessor of an enumeration type member.
- The `ord` function (see Section 10.4 [Conversion Functions], page 79) computes the ordinal number of an enumeration type member.

Thus, for instance, `sun < mon` ⇒ `true`, `ord sun` ⇒ `0`, and `succ sun` ⇒ `mon`. Note that there is no builtin operation for converting ordinal numbers back to the corresponding members of a given type. However, using the builtin `isconst` function (see Section 10.7 [Miscellaneous Functions], page 90) and the standard library `while` function (see Section 11.1 [Standard Functions], page 94), you can list all members of an enumeration type starting at a given constant `A` as follows:

```
enum A                  = while isconst succ A;
```

Now it is an easy matter to define a variable which stores the member list:

```
var days;
def days                = tuple (enum sun);
```

Note that we convert the list of days to a tuple here, which gives us constant-time access to the individual members, and is also more space-efficient. Using these definitions, we have the following:

```
==> days
(sun,mon,tue,wed,thu,fri,sat)

==> days!4
thu
```

As already mentioned, the built-in `Char` type is also an enumeration type, which consists of all the characters in the local character set. Thus the `enum` function defined above also works on character values.

8.3 Sub- and Supertypes

The Q programming language also provides a subtype concept similar to the notion of single inheritance in object-oriented programming languages such as Smalltalk. For instance, we can modify our declaration of the `BinTree` type (cf. Section 8.1 [Using Type Guards], page 61) in order to make `BinTree` a subtype of the supertype `SearchTree` as follows:

```
public type BinTree : SearchTree = private const nil, bin X T1 T2;
```

Quite often, supertypes are *abstract* types which do not provide their own set of constructor symbols, but are simply used to factor out common operations shared among several "concrete" types. For instance, the `SearchTree` type might have been declared simply as follows:

```
public type SearchTree;
```

Now variables of type `SearchTree` will also match objects of its subtype `BinTree`, as well as of any other subtype of `SearchTree`. We can turn the `union` and `diff` functions from Section 8.1 [Using Type Guards], page 61, into operations on the `SearchTree` type as follows:

```
union T1:SearchTree T2:SearchTree
                        = foldl insert T1 (members T2);

diff T1:SearchTree T2:SearchTree
                        = foldl delete T1 (members T2);
```

As the next step, we might introduce another type `AVLTree` providing the same interface operations `insert`, `delete` and `members` as the `BinTree` type, but implementing these operations in terms of AVL trees rather than simple binary trees. (AVL trees are a variant of binary search trees in which the trees are kept balanced, and thus logarithmic insertion and deletion times can be guaranteed.) If we make `AVLTree` another subtype of `SearchTree`, then the `union` and `diff` operations can be applied to `AVLTree` objects just as well as to `BinTree`'s. In fact, the operations will even work if we mix argument types, e.g., provide a `BinTree` as the first argument of `union` and an `AVLTree` as the second! By these means, it is possible to define polymorphic operations which are applicable to several different types sharing the same (subset of) interface functions.

For the sake of concreteness, here is an implementation of the `AVLTree` type. The `shl`, `shr`, `rol` and `ror` functions implement the common AVL tree rotation operations which are used to keep the tree balanced; see [Bird/Wadler 1988] for details. We assume that the declaration of the `SearchTree` type is in a separate module `searchtree.q`, along with the declarations of the shared `members`, `insert` and `delete` operations, as well as the declarations and definitions of the generic `union` and `diff` functions.

```
include searchtree;

/* H denotes the height of a nonempty AVL tree */

public type AVLTree : SearchTree = private const nil, bin H X T1 T2;

public mkavltree Xs;

private mknode X T1 T2;
private join T1 T2, init T, last T, height T, slope T, rebal T;
private rol T, ror T, shl T, shr T;

mkavltree Xs:List       = foldl insert nil Xs;
```

```
members nil             = [];
members (bin H X T1 T2) = members T1 ++ [X|members T2];

insert nil Y            = bin 1 Y nil nil;
insert (bin H X T1 T2) Y
                        = rebal (mknode X (insert T1 Y)) T2 if X>Y;
                        = rebal (mknode X T1 (insert T2 Y)) if X<Y;
                        = bin H Y T1 T2 if X=Y;

delete nil Y            = nil;
delete (bin H X T1 T2) Y
                        = rebal (mknode X (delete T1 Y) T2) if X>Y;
                        = rebal (mknode X T1 (delete T2 Y)) if X<Y;
                        = join T1 T2 if X=Y;

join nil T2             = T2;
join T1 T2              = rebal (mknode (last T1) (init T1) T2) otherwise;

init (bin H X T1 nil)   = T1;
init (bin H X T1 T2)    = rebal (mknode X T1 (init T2)) otherwise;

last (bin H X T1 nil)   = X;
last (bin H X T1 T2)    = last T2 otherwise;

/* mknode constructs a tree node, computing the height value */

mknode X T1 T2          = bin (max (height T1) (height T2) +1) X T1 T2;

/* height and slope compute the height and slope (difference between
   heights of the left and the right subtree), respectively */

height nil              = 0;
height (bin H T1 T2)    = H;

slope nil               = 0;
slope (bin H X T1 T2)   = height T1 - height T2;

/* rebal rebalances after single insertions and deletions */

rebal T                 = shl T if slope T = -2;
                        = shr T if slope T = 2;
                        = T otherwise;

/* rotation operations */

rol (bin H1 X1 T1 (bin H2 X2 T2 T3))
                        = mknode X2 (mknode X1 T1 T2) T3;
```

```
ror (bin H1 X1 (bin H2 X2 T1 T2) T3)
                        = mknode X2 T1 (mknode X1 T2 T3);

shl (bin H X T1 T2)     = rol (mknode X T1 (ror T2)) if slope T2 = 1;
                        = rol (bin H X T1 T2) otherwise;

shr (bin H X T1 T2)     = ror (mknode X T1 (ror T2)) if slope T2 = -1;
                        = ror (bin H X T1 T2) otherwise;
```

9 Special Forms

As discussed in Chapter 7 [Equations and Expression Evaluation], page 41, the Q interpreter evaluates expressions in applicative, i.e., leftmost-innermost, order. This means that the arguments of a function are usually evaluated before the function is applied to them, which is also known as *call by value*. Occasionally, it is useful or even essential to defer the evaluation of arguments until they are actually required in the course of a computation. For this purpose, the Q language lets you introduce so-called *special forms* which receive their arguments unevaluated (i.e., using *call by name*). This chapter discusses how these constructs are defined and used.

9.1 Basic Concepts

Consider the following definition from the standard library which implements a simple kind of conditional expression (see also Section 11.10 [Conditional Expressions], page 108):

```
ifelse P X Y            = X if P;
                        = Y otherwise;
```

The `ifelse` function takes as its first argument a truth value which is used to determine whether the second or third argument is to be returned as the value of the conditional expression. Although the above definition is perfectly correct, using applicative order evaluation with this definition is clearly inappropriate since all arguments must already have been evaluated before the `ifelse` function gets applied to them. Since either the second or third argument is simply thrown away, the effort involved in the evaluation of this argument is wasted. As an extreme case, e.g., `Y` might not have a terminating evaluation at all, in which case the evaluation of `ifelse P X Y` would not terminate either even though `Y` is actually not required if `P` happens to evaluate to `true`.

Instead, we would like to have `ifelse` evaluate its arguments only as they are required. In the Q language, this can be done by simply declaring `ifelse` as a *special form*. All we have to do is to precede the above equations with the following declaration:

```
public special ifelse P X Y;
```

The syntax of such declarations has already been introduced in Chapter 5 [Declarations], page 25. The `special` keyword must be followed by a function identifier and a sequence of variable symbols (the variable symbols only serve to specify the number of arguments, and are otherwise treated as comments). If the argument list is omitted, the function symbol is actually treated as an ordinary (non-special) symbol. Otherwise the given arguments are declared as *special* (a.k.a. *call by name*) arguments, i.e., arguments which are treated as literals and hence are not evaluated when the given function is applied to them. Special arguments will be left unevaluated as long as they are “protected” by a special form; but as soon as the corresponding values are referred to in a “non-special” context, the interpreter will automatically evaluate them.

Thus the above declaration of the `ifelse` function tells the Q interpreter that this function expects three special arguments `P`, `X` and `Y` which should be left unevaluated until their values are actually required in the qualifier or the right-hand side of an equation in `ifelse`’s definition. Consequently, when the interpreter comes to consider the first rule,

```
ifelse P X Y            = X if P;
```

it will first evaluate `P` to obtain the value of the condition part of this rule. If `P` evaluates to `true`, `X` will be evaluated and returned as the value of the left-hand side expression. Otherwise (if `P` evaluates to `false`), `X` will be left unevaluated, and the interpreter considers the next rule,

```
ifelse P X Y            = Y otherwise;
```

which causes it to evaluate `Y` and reduce `ifelse P X Y` to this value.

Note that a special argument is evaluated each time its value is required on the right-hand side of a definition. Consider, for instance:

```
special foo X;
foo X                   = bar X X;
```

When this rule is applied, `X` actually gets evaluated twice, once for each occurrence on the right-hand side of the equation. If this is not desired, you can introduce a local variable binding, e.g.:

```
special foo X;
foo X                   = bar Y Y where Y = X;
```

Another important point is that in the Q language, special forms are indeed a runtime feature. This means that special forms are not only recognized at compile time, but also when they are passed as arguments or returned as function results in the course of a computation (which usually cannot be predicted at compile time). For instance, if we define `foo` as follows:

```
special bar X;
foo                     = bar;
```

then `foo (1+1)` evaluates to `bar (1+1)` and *not* to `bar 2`, although `foo` itself is not declared to be a special form. This also works if you pass `bar` as a functional parameter:

```
special bar X;
foo F X                 = F (X+1);
```

Given the above definition, `foo bar 1` will evaluate to `bar (1+1)`.

On the other hand, you must take care that arguments to special functions which are passed on by other functions are not evaluated too early. For instance, if the `apply` function is defined as follows:

```
apply F X               = F X;
```

and is then invoked as `apply bar (1+1)` then `(1+1)` will be evaluated even though `bar` is a special form. The reason is that the argument `(1+1)` is evaluated *before* `apply` is applied to it, since we have not declared `apply` as a special form as well. As a general rule, you should make any function a special form that passes its arguments to another special form, unless you explicitly wish to evaluate these arguments.

Up to now, we have only considered "pure" special forms, which receive *all* their arguments unevaluated. Many functions, however, require a mixture of special and non-special arguments. The Q language provides two different methods for dealing with such situations.

First, you can *force* the evaluation of a special argument at runtime using the '~' (tilde or "force") operator:

```
special foo P X;
foo P X                 = ifelse ~P (bar X) X;
```

Here, the condition `P` is evaluated *before* it is passed on to the `ifelse` special form (which is as defined above). This method is useful if we only occasionally have to evaluate a parameter of a special form.

We remark that you can also force expressions outside a special form; in this case '~' acts as the identity operation, i.e., the argument expression (which is already in normal form) is simply returned as is.

If we *always* want to evaluate a given parameter of a special function, we can also declare the corresponding argument as *non-special* which effectively turns the argument into an ordinary *call by value* parameter. This is done by preceding the argument with the '~' symbol in the declaration of the function symbol. For instance, in the above definition of the `ifelse` function the first argument will be evaluated anyway. Hence we may as well make it a non-special argument. The corresponding declaration reads as follows:

```
public special ifelse ~P X Y;
```

With this declaration (which is precisely how `ifelse` is actually declared in the `cond.q` standard library module), the `ifelse` function always receives its first argument evaluated, while the other two arguments are special. This works exactly like an explicit application of the '~' operator, but releases you from the burden of having to force evaluation of the argument in every application of the `ifelse` function. It is also slightly more efficient since the argument does not have to be constructed as a literal expression, but can be evaluated right away before the `ifelse` function is applied to it.

Let us briefly comment on Q's special forms versus "real" lazy evaluation in functional languages like Haskell. It should be clear by now that Q allows you to keep a special argument from being evaluated as long as you like, maybe "to eternity". That is, the interpreter actually computes so-called "weak" normal forms in which special arguments are left unevaluated, even if these arguments are reducible. This is similar to the way deferred evaluation is handled in traditional functional languages featuring a basic eager evaluation strategy. In contrast, a pure lazy functional language interpreter will eventually reduce *each* expression to a "true" normal form to which no definition is applicable. Both approaches have their pros and cons. The leftmost-outermost strategy of lazy functional language interpreters is in fact only truly lazy for the lambda and combinatorial calculi used by these systems, which are very special kinds of rewriting system. In general term rewriting, however, there is no single "optimal" evaluation strategy; in fact, the problem to devise a terminating, let alone an optimal reduction strategy for a given rewriting system, even for a single input expression, amounts to solving the infamous halting problem which is undecidable. Although the Q interpreter uses a fixed built-in leftmost-innermost evaluation strategy, special forms effectively give you full control over the evaluation strategy. Moreover, special forms are also a powerful device for implementing "meta functions" which operate on other, literal expressions. This is discussed in more detail in Section 9.4 [The Quote Operator], page 73.

9.2 Special Constructors

Special forms prevent their arguments from being evaluated until they are used in a context which is not "protected" by another special form. In particular, you can also have *special constructors* which protect their arguments until they are extracted in an ordinary context. A prime example for this are *streams*, a kind of infinite list data structure. We can implement streams using a type definition like the following:

```
type Stream = special const nil, bin X Xs;
```

Just like lists, a stream consists of a head element, `X`, and a stream of remaining elements, `Xs`. Now the "head" and "tail" operations for lists (cf. Section 7.1 [Equations], page 41) carry over to streams as follows:

```
hd (bin X _)            = X;
tl (bin _ Xs)           = Xs;
```

Lisp programmers should note that this implementation does not require any explicit "delay" and "force" operations; all necessary evaluations are performed implicitly by the Q interpreter. For instance, consider the stream of all integers starting from `N`:

```
ints N                  = bin N (ints (N+1));
```

With leftmost-innermost evaluation, the above definition would certainly cause grief, because the recursive invokation of `ints` would send the interpreter into an infinite recursion. However, since `bin` is a special form, the evaluation of the nested `ints` term is deferred, and we have the following:

```
==> def nat = ints 1

==> nat
bin 1 (ints (1+1))

==> hd nat; tl nat
1
bin 2 (ints (2+1))
```

We can also have finite streams if we use the `nil` constant to signal the end of a stream, and most common list operations carry over to streams accordingly. In difference to lists, streams produce their elements only "on demand". This paves the way for some important and powerful programming techniques. For instance, streams are a useful device to implement different kinds of backtracking and dynamic programming algorithms in a uniform setting. Please refer to [Abelson/Sussman 1985] for more details. The standard library includes an implementation of streams which is described in Section 11.9 [Streams], page 106.

9.3 Built-In Special Forms

Some built-in operations of the Q language are actually implemented as special forms. This is necessary, in particular, in the case of the logical connectives `and then` and `or else` which are evaluated in "short-circuit mode", see Section 6.4.4 [Logical and Bit Operators], page 36. The first argument of these operations is non-special, but the second argument is only evaluated if it is needed. For instance, the `and then` operator first checks whether its first (evaluated) argument is a truth value (if not, the built-in rule fails). Then, if the first argument is `false`, `false` is returned – there is no need to take a look at the second argument. Otherwise, the second argument is evaluated and returned as the value of the expression. The `or else` operation is defined analogously. These rules allow the Q interpreter to perform the following reductions immediately, without ever having to evaluate the second argument `foo X`:

```
true or else foo X        ⇒ true
false and then foo X      ⇒ false
```

9.4 The Quote Operator

Another built-in special form is the predefined *quote* operator ‘'’, which acts as a constructor that protects its single argument expression from being evaluated. This construct is useful if you wish to treat certain expressions (which may or may not be in normal form) as literal objects rather than having them evaluated. For instance:

```
==> '(1+1)
'(1+1)
```

Lisp programmers should note that the quote operator is in fact a constructor, and *not* an operation which returns its single argument unevaluated. This is essential since in the Q language an expression only remains unevaluated as long as it is protected by a special form – an expression which occurs outside the context of a special form is always in normal form.

Another fact that deserves mentioning is that ‘'’ is just an “ordinary” special form, and does not involve any special “magic” on the side of the interpreter. In particular, ‘'’ does *not* inhibit variable replacements in rules like the following:

```
special foo X;
foo X                    = '(X+1);
```

Given the above definition, `foo Y` ⇒ `'(Y+1)`, i.e., the `X` variable on the right-hand side is *not* treated as a literal. However, as the very same example shows, you *can* employ ‘'’ to quote a *free* variable, and thereby defer its evaluation:

```
==> def Y = 99; foo Y
'(Y+1)

==> foo ~Y
'(99+1)
```

The force operator ‘~’ works in quoted expressions as usual:

```
==> '(1+~(2+3))
'(1+5)
```

Moreover, there is another, similar operation, the ‘`’ (backquote or “splice”) operator, which forces evaluation of its argument like the ‘~’ operator, but also “unquotes” the result. For instance, using the same `foo` function as above, we have:

```
==> '(`(foo Y)/2)
```

```
'((Y+1)/2)
```

Like the force operator, the splice operator also works outside a special form, in which case the unquoted expression is evaluated as usual. Moreover, if the evaluated argument is an unquoted expression, it is returned as is; in this case, '`' does exactly the same as the '~' operator.

The splice operator is *the* fundamental operation when constructing quoted expressions from smaller quoted pieces, by "splicing" the pieces into a "template" expression. Lisp programmers will be familiar with this technique. However, there are some notable differences between Q's '~'/'`' and Lisp's '`,`'/'`@,`' constructs:

- Splicing removes a quote level, rather than a list level.
- Q's force and splice operators are *active* operations which can be applied in arbitrary contexts, not only in quoted terms. Thus no special "quasiquote" construct is needed.
- There is no means to quote the force and splice operations themselves – they always force evaluation, in *any* context.

We also remark that, in contrast to the quote operator, the force and splice operations *do* need special support from the interpreter. They are the *only* operations which are evaluated while a special form is being processed.

When used in concert, the quotation operators ''', '`' and '~' become powerful tools for the symbolic manipulation of literal expressions. They allow you to define functions which analyze and transform an expression before the modified expression is finally evaluated. Such "meta-functions" are useful in many applications, and are an essential requisite in artificial intelligence algorithms. Substantial examples can be found in the standard library scripts, see Chapter 11 [The Standard Library], page 93; in particular, take a look at how the `lambda` and `listof` functions are implemented. As a simple (and somewhat contrived) example, let us write a function which takes an arbitrary quoted expression and replaces each instance of a `foo X` subterm by a corresponding `bar X` expression:

```
foo X                   = X-1;
bar X                   = X+1;

foobar '(foo X)         = '(bar `(foobar 'X));
foobar '(X Y)           = '(`(foobar 'X) `(foobar 'Y));
foobar '(X|Y)           = '(`(foobar 'X)|`(foobar 'Y));
foobar '[X|Y]           = '[`(foobar 'X)|`(foobar 'Y)];
foobar X                = X otherwise;
```

To see `foobar` in action, try the following:

```
==> def x = 12, y = 14, X = '(x+foo (y+foo (x*y)))

==> `X
192

==> foobar X
'(x+bar (y+bar (x*y)))
```

```
==> `_
196
```

10 Built-In Functions

Besides the built-in operators already discussed in Chapter 6 [Expressions], page 29, the Q language also provides a collection of predefined functions which cover a variety of elementary operations. The built-in functions can be divided into seven major groups, arithmetic and numeric functions, string functions, conversion functions, I/O functions, exception handling functions, and miscellaneous functions. We will describe each of these in turn.

10.1 Arithmetic Functions

The Q language provides two functions for simple "arithmetic" on enumeration types (cf. Chapter 8 [Types], page 61):

`succ X` successor function

`pred X` predecessor function

As already discussed in Section 8.2 [Built-In and Enumeration Types], page 63, these function produce the successor and predecessor of a given constant in an enumeration type (including the built-in `Char` type). Note that the builtin definitions of these operations only apply to enumeration types. However, it is an easy matter to extend them to integers accordingly, and the standard library script `stdlib.q` actually contains such definitions, cf. Section 11.1 [Standard Functions], page 94.

Two additional functions are provided for integer arithmetic, `shl` and `shr`, which are used to shift the bits of an integer value a given number of bit positions to the left and right, respectively:

`shl N COUNT`
shift `N COUNT` bits to the left

`shr N COUNT`
shift `N COUNT` bits to the right

If the `COUNT` argument is negative, then the opposite shift by `-COUNT` bits is performed, i.e., `shl X COUNT = shr X (-COUNT)`.

These operations provide the fastest possible way to multiply an integer with, and divide it by, a power of two. More precisely, if `X` is an arbitrary and `N` a nonnegative integer, then `shl X N` resp. `shr X N` is the same as `X*pow2 N` resp. `X div pow2 N`, where `pow2 N` denotes the `N`th power of 2, as an integer. We remark that in contrast to bit shifts on fixed size machine integers, there is no "loss" of most significant bits, because Q integers have arbitrary precision; hence the results obtained with bit shifting are always "arithmetically correct".

The bitwise logical operations provide a means to implement sets of nonnegative integers in a fairly efficient manner. As usual, such "bitsets" are obtained by turning on exactly those bits whose positions are the members of the set. Thus, `0` encodes the empty set, and `shl 1 I` the set consisting of the single member `I`. You then employ `or` as set union, `and` as set intersection, and `not` as set complement. Set difference can be implemented by taking the intersection with the complement set.

For convenience, you might wish to define yourself a `bit` function as follows:

```
bit                     = shl 1;
```

Then you can test for membership using an expression like `X and bit I` which returns nonzero (namely `bit I` itself) iff `I` is in the set. You can also build a set from its members by `or`'ing the corresponding `bit` values, e.g.: `bit 1 or bit 4 or bit 39`.

A final remark: Because Q integers have arbitrary precision, bitsets do not have an a priory size restriction in Q (in fact, they can even be infinite, as negative integers are used to encode the complements of the finite bitsets). However, you should note that a finite bitset (a.k.a. a nonnegative integer) needs space proprotional to the set's largest member, and hence this method will be practical only if the potential set members are within a reasonable range.

10.2 Numeric Functions

`exp X` exponential function

`ln X` natural logarithm

`sqrt X` square root

`sin X` sine

`cos X` cosine

`atan X` arcus tangent

`atan2 Y X` arcus tangent of `Y/X`

`random` random number

`seed N` initialize random number generator

This group includes the usual trigonometric and exponential functions, logarithms and square root. All these functions take both integer and floating point arguments. They return a floating point number. The `atan2` function is like `atan`, but takes two arguments and computes the arcus tangent of their ratio. In difference to `atan`, it takes into account the signs of both arguments to determine the quadrant of the result.

Besides this, there is a parameterless function `random` which returns a 32 bit pseudo random integer in the range 0..2^32-1. To obtain a random 32 bit floating point value in the range [0,1], you simply divide `random` by 0xffffffff. The current implementation uses the "Mersenne Twister", a fast uniform random number generator with a period of 2^19937-1, written by Makoto Matsumoto and Takuji Nishimura. The generator is initialized automatically with a seed taken from the system clock at the time the interpreter starts up. The `seed` function allows to initialize the generator explicitly with a given nonnegative integer seed; this is useful if it is necessary to reproduce random sequences. Note that only the least significant 31 bits of the given seed value are actually used; these 31 bits are padded with a constant 1 bit, since the generator works best with odd seed values.

10.3 String/List/Tuple Functions

This group provides some additional operations on sequences (strings, lists and tuples).

`sub Xs I J`
: extract the subsequence consisting of members `I` thru `J` of a string, list or tuple `Xs`

`substr S K L`
: return substring of length `L` at position `K` in `S`

`pos S1 S`
: position of substring `S1` in `S`

The `sub` function returns a subsequence (also called a "slice") of a string, list or tuple, given by the zero-based index values `I` and `J`. For instance:

```
==> sub "abcde" 2 3
"cd"

==> sub [a,b,c,d,e] 2 3
[c,d]

==> sub (a,b,c,d,e) 2 3
(c,d)
```

The `sub` function handles all combinations of index arguments in a reasonable manner. Thus, if `J < I` or `I >= #Xs` then an empty sequence is returned, `I = 0` is assumed if `I < 0`, and the remainder of the sequence is returned if `J >= #Xs`.

Besides `sub`, there are two additional functions operating exclusively on strings. The `substr` function returns a substring of a string with a given length at a given position. This function is provided for backward compatibility; note that `substr S K L` is equivalent to `sub S K (K+L-1)`. The `pos` function returns the position of a substring in a string (`-1` if there is no occurrence of the given substring). For instance:

```
==> pos "cd" "abcde"
2

==> substr "abcde" 2 2
"cd"
```

10.4 Conversion Functions

This group consists of functions for converting between different kinds of objects:

`trunc X`
: truncate floating point to integer value

`round X`
: round floating point to nearest integer value

`float X`
: convert integer to floating point number

`int X`
: integer part of floating point number

`frac X`
: fraction part of floating point number

`hash X`	32 bit hash code of an expression
`ord X`	ordinal number of enumeration type member (or character)
`chr N`	character with given ordinal number
`list Xs`	convert a tuple to a list
`tuple Xs`	convert a list to a tuple
`str X`	convert expression to string
`strq X`	convert quoted expression to string
`val S`	convert string to expression
`valq S`	convert string to quoted expression

The `ord` and `chr` functions convert between characters and their ordinal numbers in the local character set. The `ord` function can also be used to compute the ordinal number of a member of an arbitrary enumeration type, see Section 8.2 [Built-In and Enumeration Types], page 63.

The `hash` function returns a nonnegative 32 bit hash code for an arbitrary expression. The only guarantee is that syntactically equal objects are mapped into the same hash code. This function is useful, e.g., for implementing hashed dictionaries which map arbitrary keys to corresponding values, see Section 11.6.6 [Hashed Dictionaries], page 102 for an application.

The `str` function converts its argument expression to the corresponding print representation conforming to the Q expression syntax, which is returned as a string; the argument is evaluated as usual. The `strq` function is similar, but converts a quoted expression (cf. Section 9.4 [The Quote Operator], page 73). For instance:

```
str (1+1)              ⇒ "2"
strq '(1+1)            ⇒ "1+1"
```

The `val`/`valq` functions are the counterpart of `str` and `strq`; they convert a string containing the print representation of an expression back to an expression. In the case of `val`, the expression is evaluated, while `valq` returns the parsed expression as it is, protected with the quote operator:

```
val "1+1"              ⇒ 2
valq "1+1"             ⇒ '(1+1)
```

These functions require that the contents of their string arguments conform to the Q expression syntax. In case of a syntax error, the built-in rules fail.

All expression conversion routines use the global scope, i.e., the namespace of the main script, for the purpose of converting identifiers, just like the interpreter itself. Thus these functions work consistently in a given main script no matter which module they are invoked from. However, this also means that the precise results may depend on which main script is currently loaded. Note that this only affects the parsing and unparsing of identifiers. For instance, if a function symbol `foo` in module `bar` is visible in the main scope then it will be unparsed simply as `"foo"`, otherwise the qualified form `"bar::foo"` is used.

A word of caution: Having a "universal" string representation of Q expressions is very useful to facilitate interaction with the user, but you should note that the expression conversion routines are so powerful that they can easily be abused for rather questionable purposes. For instance, the `val` function gives you a "backdoor" to access *any* function or variable symbol in a script, even private symbols in other modules. The only way around this would have been to severely restrict the functionality of the conversion routines, which would have made them much less useful. Thus it is in your responsibility to use these functions in an "orderly" manner.

10.5 I/O Functions

This group provides functions for handling input/output from/to the terminal or a text file. These functions implement operations with side-effects; the side-effects consist in modifying the terminal display or a file.

`fopen NAME MODE`
: open file `NAME` in mode `MODE`

`popen CMD MODE`
: open pipe `CMD` in mode `MODE`

`fclose F`
: close a file

`read, fread F`
: read an expression from the terminal or a file

`readq, freadq F`
: read a quoted expression from the terminal or a file

`readc, freadc F`
: read a character from the terminal or a file

`reads, freads F`
: read a string from the terminal or a file

`write X, fwrite F X`
: write an expression to the terminal or a file

`writeq X, fwriteq F X`
: write a quoted expression to the terminal or a file

`writec C, fwritec F C`
: write a character to the terminal or a file

`writes S, fwrites F S`
: write a string to the terminal or a file

`eof, feof F`
: check for end-of-file on the terminal or a file

`flush, fflush F`
: flush output buffer on the terminal or a file

10.5.1 Terminal I/O

Input from the terminal (i.e., the standard input device) is done through the parameterless functions `read`, `readq`, `readc` and `reads` which read and return, respectively, an expression, a quoted expression, a single character, and a string. Terminal input is line-buffered which means that you must type an entire input line before anything is returned by these functions.

The `reads` function obtains one line of input, strips off the trailing newline character, and returns the result as a string. For instance (here and in the following, <CR> means that you hit the carriage return key to terminate the input line, rather than actually typing the string "<CR>"):

```
==> reads
one line of text<CR>
"one line of text"
```

The `readc` function allows you to read input character by character; at the end of a line the newline character `"\n"` is returned:

```
==> readc
<CR>
"\n"
```

The `read` and `readq` functions read one line from the input, as with `reads`, and convert the resulting string to an expression, as with `val` and `valq`, respectively. For instance:

```
==> read
1+1<CR>
2

==> readq
1+1<CR>
'(1+1)
```

The corresponding functions for producing output on the terminal (the standard output device) are named `write`, `writeq`, `writec` and `writes`. They print an expression, a quoted expression, a character and a string, respectively. These functions take one argument, the object to be printed, and return the empty tuple `()`. For instance:

```
==> writec "x"
x()

==> writes "one line of text\n"
one line of text
()

==> write (1+1)
2()

==> writeq '(1+1)
1+1()
```

(Output operations are invoked solely for their side-effects. However, any Q expression must have a value, and since there is no other meaningful value to return, the write functions return (). In the above examples, this value is output by the interpreter after the printed text, which explains, e.g., the x() in response to writec "x".)

It is common practice to combine these operations by means of the || operator in order to implement dialogs such as the following prompt/input interaction:

```
==> writes "Input: " || reads
Input: one line of text<CR>
"one line of text"
```

You will also often encounter several output operations for interpolating data into text fragments:

```
==> writes "The result is " || writeq '(1+1) || writes ".\n"
The result is 1+1.
()
```

The eof function allows you to check for *end-of-file* on the terminal. Actually, this causes another line of input to be read from the terminal if no input is currently available, to see whether the end of the input has been reached. Most operating systems allow you to type a special character to indicate end-of-file, such as the Ctl-D character on the UNIX system:

```
==> eof
<Ctl-D>
true
```

The flush function writes any pending output to the terminal. It is rarely necessary to call this function explicitly; see also the discussion of the fflush function in Section 10.5.2 [File I/O], page 83.

10.5.2 File I/O

File input and output is implemented by the functions fread, freadq, freadc, freads, fwrite, fwriteq, fwritec and fwrites. There also is an feof function which allows to check whether the end of a file has been reached, and an fflush function which flushes the output buffer of a file. These operations are analogous to their terminal equivalents, but take an additional first argument, a *file object*. A file object is a special kind of elementary object in the Q language which is returned by the built-in fopen function.

The fopen function takes two string arguments, the *name* of the file to be opened (which is the name under which the file is known by the operating system), and the *mode* in which the file is to be opened. The mode string "r" means to open an existing file for reading, "w" to open a new file for writing (existing files are truncated to zero size), and "a" to create a new or append to an existing file. You can also add the letter "b" at the end of the mode string to indicate that the file should be opened as a *binary file*. This only has the effect to suppress the LF/CR-LF conversion on MS-DOS/Windows systems, which is essential if you read/write binary data from/to the file. On UNIX systems this flag is ignored.

For instance, a function checking whether a file exists and is accessible for reading can be implemented as follows:

```
exists NAME             = isfile (fopen NAME "r");
isfile X:File           = true;
isfile X                = false otherwise;
```

Files are closed automatically when the corresponding file objects are no longer accessible in a computation. E.g., with the above definition of the `exists` function, if you invoke this function as `exists "myfile"`, then the file object returned by `fopen "myfile" "r"` (assuming that the file actually exists) will become inaccessible as soon as it has been processed by the `isfile` function, at which point it gets closed. Similarly, if you assign a file to a variable in the interpreter,

```
==> def F = fopen "myfile" "r"
```

the file will be closed as soon as you undefine the variable:

```
==> undef F
```

Occasionally, it might be necessary to close a file explicitly, e.g., a file object might still be accessible, but you want to close it before you do some other processing. In this case you can invoke the `fclose` function on the file:

```
==> def F = fopen "myfile" "r"; fclose F
()
```

After closing a file, the file object still exists, but all further I/O operations on it (including `fclose` itself) will fail:

```
==> freads F; fclose F
freads <<File>>
fclose <<File>>
```

(Note the notation `<<File>>` which represents a file object, since file objects have no printable representation. This syntax is also used by the expression output and unparsing functions, `write`, `str` etc. Of course, such objects cannot be reparsed using `read`, `val`, etc.)

We remark that the standard input and output devices used by the terminal I/O functions (the `readx` and `writex` functions, see Section 10.5.1 [Terminal I/O], page 82) are actually special instances of the file I/O functions, which read from standard input and write to standard output. The standard input and output devices are implemented by the interpreter as predefined file objects which are assigned to the `INPUT` and `OUTPUT` variables, see Section B.2 [Command Language], page 181. Thus, for instance, `reads` and `writes X` are equivalent to `freads INPUT` and `fwrites OUTPUT X`, respectively.

As an example for the use of file I/O, here's a tail-recursive function which copies the contents of one file opened in read mode to another file opened in write or append mode:

```
fcopy F G               = () if feof F;
                        = fwritec G (freadc F) || fcopy F G otherwise;
```

Note that file objects are indeed modified by input and output operations; at least, the file pointer is moved accordingly. Otherwise the above definitions would not work. This also becomes apparent when manipulating files interactively:

```
==> def F = fopen "xyz" "r"
```

```
==> freads F
"first line in file xyz"

==> freads F
"second line in file xyz"

...
```

The `fflush` function is used to write any buffered output to a file. This operation is only needed when the target file must be updated immediately. For instance, the target file may actually be a *pipe* and it may be necessary to get an immediate response from the program which reads the output at the other end of the pipe (see Section 10.5.3 [Pipes], page 85). Then you can use the `fflush` function as follows:

```
==> fwrites F S
()

==> fflush F // force S to be written to F immediately
()
```

We remark that if the Q interpreter runs interactively, then it automatically flushes the standard output files whenever an evaluation is finished, see Section B.1 [Running Compiler and Interpreter], page 175. And, of course, buffered output is always flushed when a file is closed. In case you have to flush standard output explicitly, use `flush` or, equivalently, `fflush OUTPUT`. (This should only be necessary if the standard output stream has been redirected to a disk file or a pipe.)

The interpreter currently does not provide direct support for reading and writing binary files, except that such files may be interpreted as a stream of bytes represented as character values. However, such functionality can easily be implemented using the C language interface (see Appendix C [C Language Interface], page 191), and in fact there is the `clib` standard library module which provides such facilities (see Chapter 12 [Clib], page 119).

10.5.3 Pipes

The `popen` function is used to create a file object connected to a *pipe*. The file objects constructed with `popen` allow you to pipe data into an operating system command, and to read the output of such a command from a file. Like `fopen`, `popen` takes two string arguments. The first parameter denotes the command to be executed, and the second parameter specifies the mode in which the pipe is to be opened. The mode argument must be either `"r"` (read from the output of the given command) or `"w"` (write to the input of the command), and causes a file object open for reading or writing to be returned, respectively. The `"b"` flag may also be used to open the pipe as a binary file, see Section 10.5.2 [File I/O], page 83.

Input pipes are typically employed for retrieving information from the host operating system. For instance, on UNIX systems we can use the `ls` command to obtain a list of filenames matching a given wildcard specification. A corresponding function `ls` which returns such a list can be implemented as follows:

```
ls S:String             = freadls (popen ("ls "++S) "r");
```

```
freadls F:File          = [] if feof F;
                        = [freads F|freadls F] otherwise;
```

As an example for an output pipe, the following function `more` pipes a file through the `more` program which displays the file page by page:

```
more F:File             = fcopy F (popen "more" "w");
```

(The definition of `fcopy` is as in Section 10.5.2 [File I/O], page 83.)

Just like ordinary files, pipes are closed automatically when the corresponding file object is no longer accessible, or explicitly by an invokation of the `fclose` function. Furthermore, the interpreter waits for the command started with `popen` to finish when closing the pipe.

Output pipes are a convenient means to implement specialized output devices in the Q language. For instance, the standard library script `graphics.q` writes its graphics output to a file `GRAPHICS` which is often defined as a pipe to a PostScript previewer like, e.g., Ghostscript:

```
def GRAPHICS = popen "gs -q -" "w";
```

10.6 Exception Handling

As already mentioned, the Q language only knows a few "hard" runtime error conditions a.k.a. *exceptions* such as stack or memory overflow. However, "soft" exceptions can also be generated and handled in any desired manner. The following functions are used to deal with all kinds of exceptions during expression evaluation:

`halt` halt evaluation

`quit` exit the Q interpreter

`break` invoke the debugger

`fail` abort the current rule

`catch F X` handle an exception

`throw X` raise an exception

`trap ACT SIG`
trap signals

The `halt` function never returns a result, but raises an exception which normally causes the evaluation process to be aborted immediately. This last resort is commonly used in case of a fatal error condition. The `quit` function is like `halt`, but also causes exit from the interpreter and returns you to the operating system shell; this operation is often used interactively to terminate a session with the interpreter. (Note that in a multithreaded script, see Section 12.12 [POSIX Threads], page 147, `quit` only terminates the program when it is invoked from the main thread. In other threads it just acts like `halt`.)

If the `break` flag is `on` (see Appendix D [Debugging], page 199), the `break` function interrupts an evaluation and invokes the symbolic debugger built into the Q interpreter, just as if the user typed `Ctl-C`. This operation allows you to set breakpoints in a script. For instance,

```
foo X           = break || bar X;
```

causes the debugger to be invoked as soon as the rule is executed. If the `break` flag is `off` then this operation has no effect. In any case, the `break` function returns `()`.

The `fail` function is used to abort the current rule, just like a failing qualifier. The difference is that `fail` can be used anywhere in the right-hand side or qualifier of a rule, and causes the rule to be exited *immediately*. This allows you to handle complicated error conditions which occur while a rule is already being executed, and also provides an efficient method for implementing backtracking algorithms. For instance, here is a quick solution for the famous "N queens" problem:

```
queens N        = search N 1 1 [];

search N I J P  = write P || writes "\n" if I>N;
                = search N (I+1) 1 (P++[(I,J)]) || fail if safe (I,J) P;
                = search N I (J+1) P if J<N;
                = () otherwise;

safe (I1,J1) P  = not any (check (I1,J1)) P;

check (I1,J1) (I2,J2)
                = (I1=I2) or else (J1=J2) or else
                  (I1+J1=I2+J2) or else (I1-J1=I2-J2);
```

This algorithm prints out all valid placements (i.e., lists of (row,column) pairs) of N queens on an N times N board. Note the use of `fail` in the second equation for the `search` function, which causes the rule to fail *after* the current placement has been tried, after which evaluation proceeds with the third rule which places the queen into the next column. This turns the definition into a backtracking algorithm. The `safe` function verifies that a placement `(I1,J1)` does not put the queen in check with any of the other queens which have already been placed.

The `catch` function allows you to handle both "hard exceptions" which are raised when the interpreter encounters one of the runtime error conditions discussed in Section 7.11 [Error Handling], page 59, and "soft exceptions" which are raised by the functions `halt` and `quit` already explained above, and the `throw` function which is discussed below. Moreover, the interpreter also handles certain signals sent to it, e.g., via the `kill`(2) system call, by raising appropriate exceptions. In the current implementation, by default only the signals `SIGINT` ("break"), `SIGTERM` ("terminate") and `SIGHUP` ("hangup") are handled; the latter two are both treated as termination requests, like a call to the `quit` function. Signal handlers can also be installed by the running script using the `trap` function discussed below.

Both arguments of `catch` are special. The `catch` function first evaluates its second argument, the *target expression* and, if all is well, returns the value of that expression. Otherwise, if any exception was raised during the evaluation of the target expression, it evaluates the first argument (the *exception handler*), applies it to the value of the exception, and returns the result computed by the exception handler.

In the case of a runtime error condition and the exceptions raised with the `halt` and `quit` functions, the *exception value* is of the form `syserr N` where `N` is one of the integer error codes listed below:

1: *Break*: break signal, user typed `Ctl-C`.

2: *Halt*: invokation of the `halt` function, or request to halt evaluation from the debugger.

3: *Quit*: termination signal, invokation of the `quit` function, or request to exit the interpreter from the debugger.

4: *Memory overflow*: the attempt to allocate memory for the stack or a new expression on the heap fails.

5, 6: *Stack overflow*: the attempt to push an expression on the stack, or the activation of a rule fails because the stack size limit has been reached (see the comments on memory management in Section B.1 [Running Compiler and Interpreter], page 175). 5 signals an expression stack overflow, 6 an overflow of the rule stack.

7: *Symbol table overflow*: the attempt to create a new variable symbol fails.

8: *Conditional error*: the qualifying condition of a rule does not evaluate to a truth value.

9: *External function error*: an external function (see Appendix C [C Language Interface], page 191) signals an error condition.

Let's try this by forcing a stack overflow condition:

```
==> iter 100000000 (+) 1
! Stack overflow
>>> iter 100000000 (+) 1
                         ^

==> catch exception (iter 100000000 (+) 1)
exception (syserr 5)
```

The `syserr` constructor belongs to the built-in `SysException` type, which in turn is a subtype of `Exception`. These types may be thought of as being predefined as follows:

```
public type Exception;
public type SysException : Exception = const syserr N;
```

User-defined exceptions can be generated with `throw`. The `throw` function raises an exception whose value is given by its single (non-special) argument. For instance:

```
X/0             = throw '(X/0);
exception X     = writes "Exception: " || write X || writes "\n";
```

With these definitions we have:

```
==> catch exception (17+4*3/2)
23.0

==> catch exception (17+4*3/0)
Exception: '(12/0)
()
```

```
==> 17+4*3/0
! Exception
'(12/0)
>>> 17+4*3/0
              ^
```

Note that if an exception raised with `throw` is not handled with a corresponding `catch`, then the interpreter aborts the evaluation and prints an error message (and it will also invoke the debugger to report the rule which generated the exception, if `break` is `on`).

The `catch` and `throw` functions can also be used to implement non-local value returns. For instance, here is a variation of the N queens function which returns the first solution as soon as it has been found during backtracking:

```
queens1 N       = catch id (search1 N 1 1 []);

search1 N I J P = throw P if I>N;
                = search1 N (I+1) 1 (P++[(I,J)]) || fail if safe (I,J) P;
                = search1 N I (J+1) P if J<N;
                = () otherwise;
```

Here, the "exception handler" is just the identity function `id`, as defined by the standard library, see Section 11.1 [Standard Functions], page 94.

Another neat programming trick is the following which makes a rule fail if an exception occurs during evaluation of the right-hand side:

```
foo X           = catch fail (bar X);
```

This works because the handler argument, which is special, is evaluated only if an exception is actually encountered during evaluation of the target expression. Otherwise, `foo X` will just return the value of `bar X`.

Lisp and C++ programmers will probably miss an additional parameter which restricts the kind of exceptions handled with `catch`. You can implement this easily yourself by checking the exception value in your handler function, and "throw on" an unknown exception to the next enclosing `catch`:

```
type MyException : Exception = special const div_by_zero X;

foo X           = catch myexception (1/X);

X/0             = throw (div_by_zero (X/0));

myexception X:MyException
                = writes "myexception: " || write X ||
                  writes "\n" || halt;
myexception X   = throw X otherwise;
```

Note how we implemented the `div_by_zero` exception using our own `Exception` subtype `MyException`. This method of structuring error exceptions is recommended because it makes it easier to spot the source of an exception. It also enables us to use a type guard (cf. Chapter 8 [Types], page 61) in order to check for specific exception types, rather than having to discriminate over different exception patterns.

Finally, the `trap` function allows you to have exceptions be generated when a signal arrives. The `trap` function takes two arguments, `ACT` and `SIG`, denoting the action to be performed and the number of the signal to be trapped. It returns the previous action associated with the signal. `ACT` must be an integer value. If it is positive then the signal raises an exception of the form `syserr (-SIG)`; if it is negative then the signal is completely ignored; and if it is zero then the interpreter reverts to the default handling of the signal. You can also use `trap` to redefine the default handling of the break and termination signals. To allow signals to be caught reliably, further signals are "blocked" (i.e., they are queued for later delivery) while `catch` evaluates an exception handler. For instance, here is a little script which sets up some signal handlers and keeps on printing the trapped signals until it either gets terminated with `kill -9` or the timed wait with the `sleep` function times out. (The `sleep` function is described below. The symbolic signal constants and the `getpid` and `printf` function are provided by the `clib` module, see Chapter 12 [Clib], page 119.)

```
test            = do (trap 1) [SIGINT,SIGTSTP,SIGTERM,SIGHUP,SIGQUIT] ||
                  printf "I'm running as pid %d, try to kill me!\n" getpid ||
                  loop;

loop            = flush || catch sig (sleep 100);

sig (syserr K)  = printf "Hey, I got signal %d.\n" (-K) || loop;
```

10.7 Miscellaneous Functions

This group consists of some special operations which do not fit into any of the other groups treated above.

`version, sysinfo`
: determine version/system information

`which S`
: determine absolute pathname of file on Q library path

`time`
: get the system time

`sleep X`
: pause for some time

`isspecial X`
: predicate checking whether argument is a special form

`isconst X`
: predicate checking whether argument is a constant

`isfun X, isvar X`
: predicates checking whether argument is a function or a variable symbol, respectively

`isdef X`
: predicate checking whether the variable `X` has been assigned a value

`flip F`
: flip arguments of a binary function

The `version` and `sysinfo` functions are useful when writing code which depends on a particular version of the Q interpreter or a machine/operating system type. They return a string describing the Q interpreter version and the host system on which it is installed, respectively.

The `which` function performs a search for the given filename on the Q library path. If the file is found then the absolute pathname is returned; otherwise the function fails. The path information is useful, e.g., if a script installed on the library path has to locate additional data files. If the data files are stored with the script (say, in a subdirectory `foo` of the library directory for a script named `foo.q`) then you can determine the path to these files simply as follows:

```
prefix NAME     = substr ANAME 0 (#ANAME-2) where ANAME = which NAME;
```

For instance, if `/usr/share/q/lib/foo.q` is the absolute name of the installed script then `prefix "foo"` returns `"/usr/share/q/lib/foo"`.

Two functions are provided for timing purposes. The `time` function returns, as a floating point value, the time in seconds since the "epoch" (00:00:00 UTC, January 1, 1970). The `sleep` function suspends evaluation for the given time (integer or floating point value, again in seconds). Be warned that the actual resolution of these functions depends on the timing routines available on your system, and even if a decent resolution is provided, the accuracy of results will depend on various factors such as system load.

The `isspecial`, `isconst`, `isfun`, `isvar` and `isdef` predicates are all special forms checking whether their single argument has a certain property. The `isspecial` function allows you to determine whether its argument is a special form which will receive its next argument unevaluated. The `isconst` operation checks whether its argument is a constant, i.e., a constant belonging to any of the built-in types, a variable or function symbol declared with `const`, or an application whose head element is such a constant. The `isfun` and `isvar` predicates are used to check whether an expression is a function symbol (built-in or defined by the loaded script, can also be a constant symbol), or a free variable symbol, respectively. The `isdef` function can be used to check whether its argument (a variable) has been assigned a value using `def`.

The `flip` function exchanges arguments of a (binary) function, as if it was defined by the following equation:

```
flip F X Y              = F Y X;
```

This function is used internally to implement operator sections with missing left argument. E.g., `(+1)` is nothing but syntactic sugar for `flip (+) 1`. You can apply `flip` to other, user-defined functions as well; thus, `flip foo X` denotes the function which maps `Y` to `foo Y X`.

In order to handle special forms like, e.g., `(or else)` correctly, `flip` automagically adjusts to the argument pattern of its first argument. That is, if the function given as the first argument (which is always evaluated) has a special first (resp. second) argument, then `flip`'s second (resp. first) argument will be special as well.

11 The Standard Library

This chapter gives an overview of the data types and operations provided by the standard library modules supplied with the Q programming system. The library currently consists of the following scripts:

`assert.q` print diagnostics

`clib.q` various system functions (external module)

`comp.q` comparison of lists and tuples

`complex.q`
complex numbers

`cond.q` conditional expressions

`error.q` print error messages

`graphics.q`
PostScript graphics interface

`lambda.q` an implementation of the lambda calculus

`list.q` list comprehensions

`math.q` mathematical functions

`prelude.q`
standard prelude

`sort.q` algorithms to sort a list

`stddecl.q`
shared declarations of the container data structures (cf. `stdtypes.q`)

`stdlib.q` a collection of common standard functions

`stdtypes.q`
a collection of efficient container data structures

`stream.q` an implementation of streams

`string.q` additional string functions

`typec.q` type-checking predicates

The `prelude.q` script implements the standard prelude, which loads the entire collection (except `graphics.q` which must always be imported explicitly).

The `stdlib.q` script has those operations which will probably be used most frequently by the average programmer; it contains a lot of additional list processing functions and other useful stuff mostly adopted from [Bird/Wadler 1988]. This module is also included by many other library scripts. Historically, it was the first script written for the standard library.

The `stdtypes.q` script simply includes all modules which implement common container data structures; currently these are `array.q`, `bag.q`, `dict.q`, `hdict.q`, `heap.q` and `set.q`, see Section 11.6 [Standard Types], page 98. Some declarations shared by these modules are in the `stddecls.q` script.

The `clib.q` module gives access to various useful "system" functions from the C library, including C-style formatted I/O, binary file I/O, process and thread management and regular expression routines. The operations of this module are mostly written in C, using the C language interface discussed in Appendix C [C Language Interface], page 191. This module is described in its own chapter, see Chapter 12 [Clib], page 119.

Besides this, the standard distribution of the Q programming system also includes some additional modules for interfacing to GNU dbm, ODBC, Octave, Tcl/Tk, GGI (a portable graphics interface) and IBM's Data Explorer visualization software. Preliminary documentation for these modules can be found in the `etc` subdirectory of your Q installation directory (usually in `/usr/share/q` or `/usr/local/share/q`).

11.1 Standard Functions

The `stdlib.q` script provides frequently used list operations and other stuff mostly adopted from [Bird/Wadler 1988].

`abs X` absolute value of X

`all P Xs` verify that each element of the list `Xs` satisfies the predicate `P`

`any P Xs` verify that the list `Xs` contains an element satisfying predicate `P`

`append Xs Y`
: append a value `Y` to the list or tuple `Xs`

`apply X Y` apply function `X` to argument `Y`

`cat Xs` concatenate a list of lists

`compose X Y`
: function composition: `compose X Y Z` ⇒ `X (Y Z)`

`cons X Xs` prepend an element to a list or a tuple

`cst X` constant-valued function: `cst X Y` ⇒ `X`

`do F Xs` apply a function `F` to each member of a list `Xs`, return `()`

`drop N Xs` remove the first `N` elements from the list `Xs`

`dropwhile P Xs`
: remove elements from the beginning of `Xs` while the predicate `P` is satisfied

`eq X Y` syntactic equality (cf. Section 7.2 [Non-Linear Equations], page 45)

`filter P Xs`
: filter a list with a predicate

`foldl F A Xs`
: fold-left

`foldl1 F Xs`
: fold-left over nonempty lists

`foldr F A Xs`
: fold-right

`foldr1 F Xs`
: fold-right over nonempty lists

`fst Xs` return first element of a tuple

`hd Xs` return the head element of a list

`hds Xs` return the list of all head elements in a list of lists

`id` the identity function: `id X` ⇒ `X`

`init Xs` return list `Xs` without its last element

`iter N F A`
: generate the list of the first `N` values `A`, `F A`, `F (F A)`, . . .

`last Xs` return the last element of a list

`map F Xs` apply function `F` to each member of a list

`max X Y` maximum of two values

`min X Y` minimum of two values

`mklist X N`
: create a list of `N` `X`'s

`neg P` negate a predicate

`neq X Y` syntactic inequality

`null Xs` check whether a string, list or tuple is empty (`[]` resp. `()`)

`nums N M` generate a list of numbers in a given range

`numsby K N M`
: generate a list of numbers with a given step size

`pair X Y` construct a pair

`pop Xs` remove the head element from a list or a tuple

`prd Xs` product of a list of numbers

`push Xs X` prepend an element to a list or a tuple (`cons` with arguments reversed)

`reverse Xs`
: reverse a list

`scan F A Xs`
: apply `foldl` to every initial part of a list

`scan1 F Xs`
: apply `foldl1` to every nonempty initial part of a list

`sgn X` sign of a number

`snd Xs` return second element of a tuple `sum Xs` sum of a list of numbers

`take N Xs` select the first `N` elements from the list `Xs`

`takewhile P Xs`
select elements from the beginning of `Xs` while the predicate `P` is satisfied

`tl Xs` remove the head element from a list

`tls Xs` return a list of lists with all head elements removed

`top Xs` return the head element from a list or a tuple

`transpose Xs`
transpose a list of lists

`trd Xs` return third element of a tuple

`triple X Y Z`
construct a triple

`tuplecat Xs`
concatenate a list of tuples

`until P F X`
repeat applying `F` to `X` until `P` is satisfied

`unzip Xs` transform a list of pairs into a pair of lists

`unzip3 Xs` `unzip` with triples

`while P F A`
list repeated applications of `F` to `A` while `P` is satisfied

`zip Xs Ys` take two lists and return a list of corresponding pairs

`zip3 Xs Ys Zs`
`zip` with three lists

`zipwith F Xs Ys`
take two lists and map a binary function to corresponding elements

`zipwith3 F Xs Ys Zs`
`zipwith` with three lists

The `stdlib.q` script also overloads `succ` and `pred` with the usual successor and predecessor functions on integers. For instance, `succ 5 = 6` and `pred 0 = -1`.

11.2 String Functions

The `string.q` script provides a collection of additional string functions. Currently the following operations are implemented:

`chars S` return the list of individual characters in `S`

`join DELIM Xs`
concatenate a list of strings, interpolating the given `DELIM` string between each pair of consecutive strings in the list

split DELIM S
: split a string into a list of substrings delimited by characters in the given DELIM string

strcat Xs
: concatenate a list of strings

11.3 Comparison Functions

The comp.q script overloads the =, <>, <, >, <= and >= operators with rules for deciding equality and inequality of lists and tuples, and rules for ordering lists lexicographically. That is, both tuples and lists can be compared with = and <>, which is done by checking that the operands are of the same size and recursively comparing all members of the operands. Lists can also be compared *lexicographically* by recursively comparing the list members; the first unequal members decide the comparison. This works just like the lexicographic comparison of strings. Thus, e.g., [1,2] is considered to be less than both [1,3] and [1,2,1], but more than [1,1,3] or [0,5].

11.4 Type-Checking Predicates

The typec.q script contains a collection of predicates which check whether an expression is of a given type. The following operations are provided:

isbool X
: check for truth values

ischar X
: check for single character strings

isexcept X
: check for Exception values (cf. Section 10.6 [Exception Handling], page 86)

isfile X
: check for file objects

isfloat X
: check for floating point numbers

isint X
: check for integers

islist X
: check for lists

isnum X
: check for numbers

isstr X
: check for strings

issym X
: check for function and variable symbols (special form)

istuple X
: check for tuples

11.5 Sorting Algorithms

The sort.q script provides mergesort and quicksort algorithms for sorting a list using a given order predicate:

msort P Xs
: mergesort algorithm

`qsort P Xs`
quicksort algorithm

The mergesort algorithm is more involved than quicksort, but may run *much* faster if input lists are large enough and are already partially sorted. Both algorithms take an order predicate as their first argument, which makes it possible to sort lists using different criteria. The order predicate must be a function accepting two arguments, and must return `true` iff the first argument is strictly less than the second. For instance, to sort a list in ascending order, you could say:

```
==> qsort (<) [1,5,3,2,4]
[1,2,3,4,5]
```

By reversing the order predicate, the list is sorted in descending order:

```
==> qsort (>) [1,5,3,2,4]
[5,4,3,2,1]
```

Custom order predicates also allow you to sort a list according to different sort keys. For instance, the following example shows how to sort a list of pairs using the first component of each pair as the sort key (see Section 11.7 [Lambda Calculus], page 103, for a description of the `lambda` function):

```
==> def L = [(1,2),(5,1),(3,3),(1,1),(4,5)]

==> def le1 = lambda X (lambda Y (fst X < fst Y))

==> qsort le1 L
[(1,2),(1,1),(3,3),(4,5),(5,1)]
```

Both algorithms provided by this module are "stable", i.e., they preserve the relative order of list elements with "equal" sort keys. This is important when successive sorts are used to order elements according to different criteria. For instance:

```
==> def le2 = lambda X (lambda Y (snd X < snd Y))

==> qsort le2 L
[(5,1),(1,1),(1,2),(3,3),(4,5)]

==> qsort le1 _
[(1,1),(1,2),(3,3),(4,5),(5,1)]
```

11.6 Standard Types

The `stdtypes.q` script implements a collection of efficient container data structures, which currently comprises arrays, heaps (priority queues), ordered sets and bags, and (ordered as well as hashed) dictionaries. The different data types are actually implemented in the scripts `array.q`, `bag.q`, `dict.q`, `hdict.q`, `heap.q` and `set.q`. Many operations of these modules are overloaded; the declarations of these operations can be found in the `stddecl.q` script which is included by the different modules.

All data structures support equality checking with = and <>, as well as the operation # to determine the size of an object (number of elements it contains). Furthermore, the set and bag data structures overload the <, >, <= and >= operators to implement subset/subbag comparisons and the +, - and * operators to compute the union, difference and intersection of two sets or bags, respectively.

11.6.1 Arrays

The `array.q` script provides a zero-based array data structure `Array` with logarithmic access times, implemented as size-balanced binary trees. The following operations are provided:

`array Xs` create an array from list `Xs`

`array2 Xs` create a two-dimensional array from a list of lists

`emptyarray`
: return the empty array

`mkarray X N`
: create an array consisting of `N` `X`'s

`mkarray2 X (N,M)`
: create a two-dimensional array with `N` rows and `M` columns

`isarray X` check whether `X` is an array

`null A` check whether `A` is the empty array

`A1 = A2, A1 <> A2`
: array equality/inequality

`#A` size of an array

`A!I` return `I`th member of an array

`A!(I,J)` two-dimensional subscript

`members A, list A`
: list the members of `A`

`members2 A, list2 A`
: list a two-dimensional array

`first A, last A`
: return the first and last element of an array

`rmfirst A, rmlast A`
: remove the first and last element from an array

`insert A X`
: insert `X` at the beginning of `A`

`append A X`
: append `X` at the end of `A`

`update A I X`
: replace the `I`th member of `A` by `X`

`update2 A (I,J) X`
: update two-dimensional array

11.6.2 Heaps

The `heap.q` script provides an efficient heap (priority queue) data structure `Heap` implemented as size-balanced binary trees. Heaps allow quick (i.e., constant-time) access to the smallest element, and to insert new elements in logarithmic time. The present implementation does not allow fast random updates of heap members; if such functionality is required, bags should be used instead (see Section 11.6.4 [Bags], page 101).

Heap members must be ordered by the `<=` predicate. Multiple instances of the same element may be stored in a heap; however, the order in which equal elements are retrieved is not specified.

The following operations are provided:

`emptyheap`
: return the empty heap

`heap Xs`
: construct a heap from a list of its members

`isheap X`
: determine whether `X` is a heap

`null H`
: check whether `H` is the empty heap

`H1 = H2, H1 <> H2`
: heap equality/inequality

`#H`
: size of a heap

`members H, list H`
: list the members of `H` in ascending order

`first H`
: return the first element of `H`

`rmfirst H`
: remove the first element from `H`

`insert H X`
: insert `X` into `H`

11.6.3 Sets

The `set.q` script provides an ordered set data structure `Set` implemented as AVL trees, and thus guaranteeing logarithmic access times. The following operations are defined in `set.q`:

`emptyset`
: return the empty set

`set Xs`
: create a set from a list of its members

`isset X`
: check whether `X` is a set

`null M` check whether `M` is the empty set

`member M X`
check whether `M` contains element `X`

`M1 = M2, M1 <> M2`
set equality/inequality

`M1 < M2, M1 > M2, M1 <= M2, M1 >= M2`
set comparison

`M1 + M2, M1 - M2, M1 * M2`
set union, difference and intersection

`#M` size of a set

`members M, list M`
list the members of `M` in ascending order

`first M, last M`
return the first and last member of `M`

`rmfirst M, rmlast M`
remove the first and last member from `M`

`insert M X`
insert `X` into `M`

`delete M X`
delete `X` from `M`

11.6.4 Bags

The `bag.q` script defines the type `Bag` as a variant of the set data structure which may contain multiple instances of the same element. The operations are analogous to those of the set data structure; see Section 11.6.3 [Sets], page 100. The `emptybag` function returns the empty bag, and `bag Xs` constructs a bag from a list `Xs` of its members. The `isbag` predicate checks whether its argument is a bag.

11.6.5 Dictionaries

The `dict.q` script supplies an (ordered) dictionary data structure `Dict` which maps *keys* from an ordered set to corresponding *values*. Like sets and bags, dictionaries are implemented as AVL trees, thus guaranteeing logarithmic access times to individual members of the dictionary. The following operations are defined in `dict.q`:

`emptydict`
return the empty dictionary

`dict XYs` create a dictionary from a list of key/value pairs

`mkdict Y Xs`
create a dictionary from a list of keys and an initial value

`isdict X` check whether X is a dictionary `null D` check whether `D` is the empty dictionary

`member D X`
check whether `D` contains `X` as a key

`D1 = D2, D1 <> D2`
dictionary equality/inequality

`#D` size of a dictionary

`D!X` return the value `Y` associated with `X` in `D`

`members D, list D`
list the members (key/value pairs) of `D` in ascending order by key

`keys D` list the keys of `D` in ascending order

`vals D` list the corresponding values

`first D, last D`
return the first and last member of `D`

`rmfirst D, rmlast D`
remove the first and last member from `D`

`insert D (X,Y)`
insert a key/value pair `(X,Y)` into `D`; update an existing entry for `X` if present

`delete D X`
remove key `X` from `D`

`update D X Y`
same as `insert D (X,Y)`

11.6.6 Hashed Dictionaries

The `hdict.q` script implements hashed dictionaries (`HDict` type), a variation of the `Dict` type which uses hashed key values obtained with the builtin `hash` function. The actual key-value pairs are stored in "buckets" for each hash value. This kind of data structure is also known as "hashes" or "associative arrays" in other programming languages.

The main advantage of the `HDict` type is that key values can be of any type and do not have to belong to an ordered set. For instance, `hdict [(0,1),(foo,2),("bar",3)]` is a legal `HDict` value which maps the integer `0` to `1`, the symbol `foo` to `2`, and the string `"bar"` to `3`.

There are some other notable differences between the `Dict` and the `HDict` type. First of all, a `HDict` stores it members in an apparently random order which may depend on the order in which new entries are added to the dictionary. Hence equality testing for `HDicts` is more involved than for `Dicts`, as the member lists of two "equal" `HDicts` may be arbitrary permutations of each other. This also means that the `first`, `rmfirst`, `last` and `rmlast` operations do not make much sense with hashed dictionaries and are not supported by the `HDict` type. Moreover, key values are always compared syntactically when looking up and updating entries. Hence, e.g., `0` and `0.0` are different key values in a `HDict` object, whereas they are considered to be the same for the `Dict` type.

Apart from these differences, the operations of the HDict and Dict types work analogously. The HDict constructors are named emptyhdict, hdict and mkhdict which take the same arguments as the corresponding Dict constructors, and the ishdict predicate checks for HDict values.

11.7 Lambda Calculus

The lambda.q script implements the special form lambda which can be used to create simple function objects "on the fly". The lambda function is invoked as:

```
lambda PAT EXPR
```

The PAT argument can be an arbitary expression which is to be matched against the actual parameter of the lambda function. If the parameter matches, the value of EXPR is returned, with the variables in PAT replaced by the corresponding values:

```
==> def foo = lambda (X,Y) (2*X+Y); foo (2,3)
7

==> def bar = lambda X (lambda Y (2*X+Y)); bar 2 3
7
```

As indicated, multi-argument functions are created using nested lambda's. Each variable is bound by the *innermost* lambda in which it occurs:

```
==> lambda X ((lambda X (X*X)) (X+1)) 2
9
```

Pattern matching is performed by traversing the pattern from left to right, and each variable is bound as it is matched. Note that, to speed up lambda compilation, it is not verified that the lambda pattern is "linear", i.e., contains each variable only once. Consequently, if a variable occurs more than once in the pattern, the different occurrences are simply matched independently from each other, and the value bound to the variable is the actual value which corresponds to the *rightmost* occurrence:

```
==> lambda (X,X) (2*X) (99,101)
202
```

If a pattern match fails, you will see some unevaluated matching "combinators" in the output. This can be cured by making judicious use of exceptions. (See the section on lambda internals below for more details on this.)

Lambdas can also be used to substitute variables in special forms like the quote operator:

```
==> lambda X '(X+1) (2*3)
'(6+1)
```

Note that while the pattern and body of a lambda expression are special arguments, the actual parameter expected by the function *created* with lambda is non-special. Thus the value 2*3 in the example above is evaluated before it is substituted for the X variable. If you want to prevent lambda arguments from being evaluated, you have to protect them with a special form (e.g., a quote), and extract the value for use in the lambda body using an appropriate pattern:

```
==> lambda 'X '(X+1) '(2*3)
'(2*3+1)
```

The `lambda` function normally expands nested lambdas in the body of a lambda, s.t. variables are always bound by the innermost lambda in which they occur. However, this recursive lambda expansion is inhibited inside the arguments of special forms. This means that lambda constructs inside a special form are effectively treated as literal expressions:

```
==> lambda X '(lambda X (2*X)) 77
'(lambda 77 (2*77))
```

(Special form detection uses static analysis of the lambda body, hence this only works with special forms explicitly present in the lambda body. That is, a non-special function in the lambda body will never be treated as special, even if it reduces to a special form at runtime.)

A word of caution: The `lambda` function uses free variable symbols in the pattern to denote the "lambda variables". However, for the interpreter a lambda term is just a normal expression, and hence it will perform its own variable replacement for left-hand side, i.e., bound, variables in an equation. Hence, if you have a lambda expression on the right-hand side of an equation, you should make sure that you only use free variable symbols for the lambda variables.

Lambda Internals

Chances are that when using lambdas, occasionally you will be confronted with partially evaluated function objects which consist of a bunch of weird-looking symbols like `_A`, `_HK` etc. A quick look "behind the scenes" will help you understand what's going on with these objects.

To improve performance, lambdas are not evaluated directly, but are "compiled" to function objects consisting of *combinators* which handle the variable matching and replacement process. The current implementation is based on the standard combinatorial calculus as described, e.g., in [Henson 1987], which has been extended to handle pattern matching, and to perform lambda substitutions in lists and tuples in an efficient manner.

A complete description of the combinatorial calculus is well beyond the scope of this little section, but here is a brief description of the most important combinators:

`_H X, _HL X, _HT X, _HK K X`
: These are the matching combinators, which match, respectively, applications, lists, tuples and constants in the argument value.

`_A X, _L X, _T X`
: Combinators for constructing function applications, lists and tuples.

`_I, _K K`
: The identity and the constant function.

`_S X Y, _B X Y, _C X Y`
: The "argument dispatching" combinators, which pass on an argument to the `X` and `Y` subfunctions. `_S` dispatches to both subterms, `_B` to `Y`, and `_C` to `X` only. Additional combinators like these, but named with a trailing `L` or `T`, are used to dispatch arguments into lists and tuples.

All combinators are implemented as special forms, in order to prevent the function body from being evaluated before the actual lambda parameter is applied. There also is an additional set just like the one sketched out above (stropped with two leading underscores instead of one), in which also the lambda *argument* is protected. These combinators are used to protect special argument subterms extracted from the lambda argument from premature evaluation.

For example, here is the combinator expression for the `lambda (X,Y) (X*Y)` function:

```
_HT (_B _HT (_B (_B (_HK ())) (*)))
```

This looks a bit frightening at first sight, but once we strip the matching combinators and the `_B` combinator which just dispatches argument values to the correct places, we are left with nothing but our familiar `(*)` function.

All combinators are provided as public symbols, so you can invoke them directly, if you know what you are doing. You can also extend the combinator definitions with additional rules, e.g., in order to implement exceptions in response to argument mismatches of the matching combinators. Here are some rules which will do the trick:

```
_H X Y                 = throw '(_H X Y);     // bad application
_HL X Y                = throw '(_HL X Y);    // bad list
_HT X Y                = throw '(_HT X Y);    // bad tuple
_HK K X Y              = throw '(_HK K X Y);  // bad constant
// combinators for special arguments
__H X Y                = throw '(__H X Y);    // bad application
__HL X Y               = throw '(__HL X Y);   // bad list
__HT X Y               = throw '(__HT X Y);   // bad tuple
__HK K X Y             = throw '(__HK K X Y); // bad constant
```

Performance Considerations

The size of a compiled lambda, i.e., the size of the corresponding combinator expression, is at most quadratic in the size of the lambda expression, and at most linear in the pattern size. The time needed to actually perform the variable replacements for a given argument is roughly proportional to the size of the combinator term. This should be efficient enough for most practical purposes, as long as you avoid huge expressions in the lambda body.

It should also be pointed out that the size of combinator terms explodes combinatorically with nested lambda expressions, so there also is a practical limit on the parameter count in multi-argument lambdas. (So-called "supercombinators" might be used to cure this, but these have not been implemented yet.)

Thus lambdas are most useful for defining simple ad-hoc functions "on the fly". Don't expect any performance miracles. A conventional function definition using equations incurs much less overhead, since it uses the built-in pattern matching and reduction engine of the interpreter.

11.8 List Comprehensions

The `list.q` script provides the special form `listof` which allows to specify a list of values in a manner similar to the way sets are described in mathematics:

```
listof EXPR CLAUSES
```

`CLAUSES` may either be a singleton value or a tuple consisting of binding clauses and conditional clauses. A *binding clause* is an expression of the form `X in Xs` (using the relational `in` operator, cf. Section 6.4.3 [Relational Operators], page 35) which specifies that the (presumably linear) pattern `X` should be matched in turn to each member of the list `Xs`; only values matching the pattern will be extracted, and free variables in the pattern are bound to their corresponding values using `lambda`. Binding clauses are considered from left to right, which means that a clause may refer to any variable introduced in an earlier binding clause.

Any other expression specifies a *conditional clause* which must evaluate to a truth value; only those elements will be listed for which the conditional clause is satisfied.

List comprehensions are best illustrated by an example. The following equations define an operation which returns the list of all triples `(I,J,I+J)` s.t. `1<=J<I<=N` and `I+J` is a prime number.

```
prime_sum_pairs N       = listof (I,J,I+J) (I in nums 1 N,
                          J in nums 1 (I-1), isprime (I+J));
isprime N               = (N>1) and then prime_test N 2;
prime_test N M          = true if M*M>N;
                        = false if N mod M = 0;
                        = prime_test N (M+1) otherwise;
```

11.9 Streams

The `stream.q` script implements streams, a variant of the list data structure with "call be need" evaluation. Streams can be used to represent infinite sequences of objects, since the objects in the sequence are only produced as they are required in the course of a computation.

The `Stream` type is declared as follows:

```
public type Stream = special const nil, bin X Xs;
```

Most list operations are overloaded and work analogously to their list counterparts. Furthermore, the script incorporates rules for comparing streams using the lexicographic order, and provides the following additional operations:

`isstream X`
: check whether an object is a stream

`stream Xs`
: convert a list to a stream

`list Xs`
: convert a stream to a list

`numstream N`
: generate the stream of all numbers `>=N`

`numstreamby K N`
: generate a number stream with given step size

`mkstream X`
: generate an infinite stream of `X`'s

`iterate F A`
: generate the stream of all values `A`, `F A`, `F (F A)`, ...

`streamcat Xs`
: concatenate a stream or list of streams and/or lists (see comments below)

`streamof X CLAUSES`
: stream comprehensions (see discussion below)

Operations like `#`, `all`, `foldl` will of course cause troubles with infinite streams since it can take them an infinite time to compute the result. The same holds for the `foldr` function unless the folded operation is a special form.

The `streamcat` function (which is analogous to the `cat` function on lists defined in `stdlib.q`) concatenates a stream of streams in a fully lazy manner, i.e., you can concatenate a (possibly infinite) stream of (possibly infinite) streams. The argument of `streamcat` can actually be a list or stream, which consists of any mixture of streams and lists. The result is always a stream.

The `streamof` special form works like `listof` (see Section 11.8 [List Comprehensions], page 106), except that it accepts both streams and lists, which may contain an arbitrary collection of stream and list values, and returns a stream instead of a list as the result. For instance, the following recursive definition is used to generate the stream of all possible permutations of a given list `Xs`:

```
perms []                = stream [[]];
perms Xs                = streamof [Y|Ys] (Y in stream Xs,
                          Ys in perms (filter (neq Y) Xs));
```

Here is an example for the use of infinite streams. It implements the well-known sieve of Erathosthenes:

```
ints N                  = bin N (ints (N+1));
primes                  = sieve (ints 2);
sieve (bin X Xs)        = bin X (sieve (filter (ndivby X) Xs));
ndivby M N              = N mod M <> 0;
```

A common pitfall with streams is that the size of a recursively defined stream expression can grow very fast, unless some care is taken. For instance, a Haskell programmer might be tempted to define the stream of all Fibonacci numbers as follows:

```
fibs = bin 1 (bin 1 (zipwith (+) fibs (tl fibs)));
```

However, this definition has a similar defect as the naive definition of the `fib` function in Section 7.1 [Equations], page 41, in that `fibs` invokes itself *twice*, and hence the tails of this stream grow exponentially with the number of applied `tl` operations. Of course this also means that the time needed to extract the `N`th member of the stream grows exponentially with `N`. Haskell programmers get away with this only because their compiler applies a clever optimization technique. In Q, the right way to define this stream is to work with a recursively defined stream which keeps track of pairs of consecutive Fibonacci numbers, like so:

```
fibs                    = fibs2 1 1;
fibs2 A B               = bin A (fibs2 B (A+B));
```

11.10 Conditional Expressions

The `cond.q` script provides some functions for implementing conditional expressions:

`ifelse ~P X Y`
: simple conditional expression (special form)

`matchp P ~X`
: match pattern against expression (special form)

`switch CASES`
: general conditional expression

`match X CASES`
: pattern-matching conditional

The `ifelse` special form has already been discussed in Section 9.1 [Basic Concepts], page 69. It returns either `X` or `Y`, depending on whether the first argument is `true` or `false`.

A more general conditional expression is also provided, which allows you to discriminate between an arbitrary number of cases. It has the form

```
switch (case P1 X1, case P2 X2, ...)
```

and returns the first value `X` for which the corresponding condition `P` evaluates to `true`. The case tuple can also contain a `default X` expression which denotes the default value to be returned when the preceding cases have failed. If no case matches, and no `default` clause is present, `switch` returns `()`. The `case` and `default` symbols are special constructor symbols declared as follows:

```
public special const case X Y, default X;
```

The `matchp` predicate returns `true` if the expression given as the second argument matches the literal pattern in the first argument, `false` otherwise.

The `match` conditional is a specialized form of `switch` in which the first `case` argument of each case is a pattern to be matched against the given expression `X`. If the expression matches, the second `case` argument is evaluated, with the variables in the pattern bound to the corresponding values using `lambda`.

Some examples:

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

f X Y Z                 = switch (case (X>0) (foo X),
                                  case (Y>0) (bar Y),
```

```
                                        default Z);

g X                       = match X (case (foo Y Z) (Y*Z),
                                     case (bar Y Z) (Y/Z),
                                     default X);

==> fac 41
33452526613163807108170062053440751665152000000000

==> f (-1) 1 0
bar 1

==> g (bar 9 10)
0.9
```

11.11 Mathematical Functions

The `math.q` script contains some additional operations on floating point numbers:

`asin X, acos X, tan X`
: additional trigonometric functions

`lg X, log X`
: base 2 and 10 logarithms

`sinh X, cosh X, tanh X`
: hyperbolic functions

`asinh X, acosh X, atanh X`
: inverse hyperbolic functions

11.12 Complex Numbers

The `complex.q` script implements complex numbers as pairs `(X,Y)` of integer or floating point values `X` and `Y`. It generalizes the usual arithmetic operations (including exponentiation), `sqrt`, `exp`, logarithms, trigonometric and hyperbolic functions accordingly. The following basic operations are also provided:

`iscomplex Z`
: check whether argument is complex (including ordinary numbers)

`abs Z, arg Z`
: absolute value and argument

`re Z, im Z`
: real and imaginary part

`conj Z`
: complex conjugate

All these operations apply to both real and complex numbers. The `arg` function returns the polar angle (in radians), thus `(abs Z,arg Z)` denotes the polar coordinates of a complex number `Z`.

11.13 Graphics

The `graphics.q` script implements an interface to Adobe's PostScript language [Adobe 1990]. PostScript is a page description language which has become one of the main standards in the desktop publishing world. The nice thing about PostScript is that page descriptions are *device-independent* – the same page description can be used for different output devices such as a screen previewer or a laser printer. The `graphics.q` script allows you to produce graphics output on any PostScript device. Note that, as already pointed out, this script does *not* belong to the set of "standard" scripts included by the prelude, and hence has to be imported explicitly if you want to use the operations described in the following.

The graphics device is implemented by the `GRAPHICS` variable, which by default is assigned to standard output. This is useful for debugging purposes, but in a real application you will of course redirect the output to some PostScript file or device, which can be done by assigning a suitable value to the `GRAPHICS` variable. Any file object open for writing will do, however the script also provides the following convenience functions which each return an output file or pipe which can be used as the value of the `GRAPHICS` variable:

`gsdev` pipe to the `ghostscript` program (see below)

`gvdev` pipe to `ghostview` (see below)

`lpdev` printer device (usually a pipe to `lpr`(1))

`filedev X` output file with name `X`

`nulldev` the null device (a synonym for `filedev "/dev/null"`, or `filedev "nul"` under DOS/Windows)

For instance, you define the graphics device as a pipe to ghostscript as follows:

```
def GRAPHICS = gsdev;
```

You can either add this definition to your main script, or enter it directly at the command prompt of the interpreter (see Section B.2 [Command Language], page 181).

Ghostscript is a popular PostScript previewer available for a wide range of different platforms. Ghostview is an improved X interface for ghostscript. A similar program, GSView, is also available for the Windows operating system.

Please note that currently only the `nulldev` and `filedev` devices are available when running under DOS or Windows (the remaining devices types are assigned to `nulldev`), since GSView currently does not support input from a pipe. To preview your PostScript output under DOS/Windows, use `filedev` to set up an output file and then invoke Ghostscript or GSView manually. For instance (assuming that the `gsview32` program is on your DOS path):

```
==> def GRAPHICS = filedev "graphics.ps"

==> // output your PostScript graphics here ...

==> !gsview32 graphics.ps
```

If output goes to a printer or a file, you will probably need a minimal header which identifies the file as a PostScript document. To include such a header in the output file, use the `psheader` function *before* invoking any other graphics operation:

```
==> psheader
```

You can also set up a custom header and include other DSC and EPSF comments by means of the `ps` function; see Section 11.13.8 [DSC and EPSF Comments], page 116, for details.

In the following we give an overview of the graphics operations in the PostScript language, as they are provided by this module, and describe the implemented functions. For more details about PostScript please refer to [Adobe 1990].

11.13.1 Coordinate System

The PostScript coordinate system has its origin `(0,0)` in the lower left corner of the output page or display window, with the positive `X` and `Y` axes extending horizontally to the right and vertically upward, respectively. The default unit length is 1 *point* which is 1/72 of an inch.

The origin of the coordinate system, as well as the unit lengths and orientation of the `X` and `Y` axes can be changed by means of the `translate`, `scale` and `rotate` operations, cf. Section 11.13.6 [Graphics State], page 114.

11.13.2 Overview of Graphics Operations

The process of painting a graphics object usually consists of the following three steps:

- Modify graphics parameters such as the text font, the color, or the translation and scaling of the coordinate axes.
- Construct a *path* which outlines the shape of the object to be painted.
- Execute the appropriate painting operation to display the object on the output page.

A *path* is an ordered sequence of straight and curved line segments. The individual segments may be connected to each other or they may be disconnected. Thus a path may consist of several connected pieces which are referred to as the *subpaths* of the path. In a subpath, each line segment starts at the point where the previous segment ends. The `newpath` function is used to begin a new path. A new subpath is obtained by invoking `moveto` which specifies the first point in the subpath. Various operations are provided to add straight and curved line segments to the current subpath. For instance, a path consisting of three straight line segments may be denoted as follows:

```
newpath || moveto 0 0 || lineto 1 0 || lineto 1 1 || lineto 0 1
```

The last point on the current subpath can be connected back to its starting point (usually the last point specified with `moveto`) by *closing* the subpath with the `closepath` operation. For instance, a rectangle is specified as follows:

```
newpath || moveto 0 0 || lineto 1 0 || lineto 1 1 || lineto 0 1 ||
closepath
```

Having constructed a path, the `stroke` function draws the line segments contained in the path. Alternatively, `fill` may be used to fill the interior of the path (for this purpose the entire path should consist of closed subpaths).

The precise appearance of stroked and filled objects on the output page is controlled by a collection of parameters referred to as the *graphics state*. Various operations are provided for changing these parameters. For instance, you can set the linewidth and dash pattern used by `stroke`, the color used by the `stroke` and `fill` operations, and the scale and translation of the coordinate axes. The current settings can be saved on a stack using `gsave` and restored (popped from the stack) with `grestore`.

Another important parameter is the current *clipping path* which specifies the regions on the page which can be affected by the painting operations. By default, the paintable area is the whole page. In order to restrict painting to a user-defined region, a path is constructed as usual, and then the `clip` function is used to set this path as the clipping path. Subsequent paint operations will only paint the interior of the clipping path, i.e., the region which would have been filled had we applied the `fill` operation instead of `clip` to the constructed path. Multiple applications of `clip` are accumulative. That is, `clip` intersects the current clipping area (as defined by previous invokations of `clip`) with the interior of the current path.

The treatment of textual output is somewhat special. It is possible to define a path consisting of the outlines of the characters in a given text string by means of the `charpath` function. More commonly, however, text strings are simply displayed at a given position which is accomplished by means of the `show` function. For instance, to display a text string `S` at a position `(X,Y)` on the current page the following expression is used:

```
moveto X Y || show S
```

The `graphics.q` script also provides several operations which deal with a graphics page as a whole. First of all, `showpage` emits the current page, and prepares for the next page by erasing the current page. The `copypage` operation is like `showpage`, but keeps the contents of the current page. This allows you to accumulate the contents of several pages. Both `showpage` and `copypage` are mainly used when output goes to a printer.

Two additional operations are provided for interactive use, when output goes to a display window. The `erasepage` function causes the contents of the current page to be erased. The `flushpage` operation updates the display, like `showpage` or `copypage`, but does not start a new page. (To improve performance, graphics output under the X window system is usually performed in larger chunks. The `flushpage` operation is required to synchronize the display by flushing any unwritten data.) Note that this operation is *not* part of the PostScript standard, but only works with Ghostscript and possibly some similar Postscript viewers.

If you want to achieve special effects which cannot be implemented in terms of the operations provided by `graphics.q`, you can directly invoke PostScript commands by means of the `ps` function. Also, you can copy a PostScript file to the graphics device with the `psfile` operation. As an example for the `ps` function, operations to display a string right-justified or centered at the current position can be implemented as follows:

```
showright S:String      = ps (psstr S++
```

```
                              " dup stringwidth pop neg 0 rmoveto show\n");
showcenter S:String           = ps (psstr S++
                              " dup stringwidth pop 2 div neg 0 rmoveto show\n");
```

(The `psstr` function converts a string to PostScript syntax; see Section 11.13.7 [Miscellaneous Operations], page 116.) The `ps` function is also useful to include DSC and EPSF comments in your graphics output if this is necessary. See Section 11.13.8 [DSC and EPSF Comments], page 116, for details.

11.13.3 Path Construction

The `graphics.q` script defines the following collection of operations to define the current path used by the painting and clipping operations:

`newpath` start a new path

`closepath`
: close the current subpath

`clippath` set the current path to the current clipping path

`moveto X Y`
: absolute move to position `(X,Y)`

`rmoveto DX DY`
: move relatively by `DX` units in horizontal and `DY` units in vertical direction

`lineto X Y`
: straight line segment between the current point and absolute location `(X,Y)`

`rlineto DX DY`
: straight line segment specified by displacement `(DX,DY)` with respect to the current point

`curveto X1 Y1 X2 Y2 X3 Y3`
: Bézier cubic section between the current point and `(X3,Y3)`, using `(X1,Y1)` and `(X2,Y2)` as control points

`rcurveto DX1 DY1 DX2 DY2 DX3 DY3`
: Bézier cubic section, with the points specified as displacements with respect to the current point

`arc X Y R A1 A2`
: arc of a circle with radius `R` centered at location `(X,Y)` starting and ending at angles `A1` and `A2` (`0<=A1,A2<=360`), respectively; if there is a current point, it is connected by a straight line segment to the first point on the arc

`narc X Y R A1 A2`
: negative arc; the arc is drawn in clockwise rather than in counter-clockwise direction

`arct X1 Y1 X2 Y2 R`
: arc specified by tangent lines; the center of the arc is located within the inner angle of the tangents, with the first tangent connecting the current point and `(X1,Y1)`, and the second tangent connecting `(X1,Y1)` and `(X2,Y2)`

`charpath S T`
: path consisting of the character outlines that would result if the string S where shown at the current point using `show`; `T` is a truth value denoting whether the path should be used for stroking to draw the character outlines (`T = false`) or whether the path should be adjusted for use with `fill` or `clip` (`T = true`)

11.13.4 Painting

The following operations are provided for stroking and filling, and for displaying text:

`stroke` draw the line segments in the current path

`fill, eofill`
: fill the interior of the current path (any unclosed subpaths of the current path are closed automatically)

`show S` paint string `S` at the current point

The `fill` function uses the "nonzero winding number" rule for determining which points lie "inside" the current path, while `eofill` uses the "even-odd" rule. Please refer to [Adobe 1990] for the details.

11.13.5 Clipping

The following operations are used to determine the current clipping path, as described in Section 11.13.2 [Overview of Graphics Operations], page 111:

`clip, eoclip`
: intersect the current clipping area with the interior of the current path

The `clip` and `eoclip` operations use the same rules for insideness testing as `fill` and `eofill`, respectively, see Section 11.13.4 [Painting], page 114.

11.13.6 Graphics State

As already indicated in Section 11.13.2 [Overview of Graphics Operations], page 111, the PostScript graphics state is a collection of parameters which control the behavior of the graphics operations. The `graphics.q` script provides the following operations to manipulate these parameters:

`gsave` save the current graphics state

`grestore` restore the previously saved graphics state

`savematrix`
: push the current transformation matrix (CTM) on the PostScript stack; the CTM is manipulated by the `translate`, `scale` and `rotate` operations, see below

`restorematrix`
: restore the CTM from the stack

`translate TX TY`
: move the origin of the coordinate system `TX` units in horizontal and `TY` units in vertical direction

`scale SX SY`
: scale the unit lengths of the coordinate axes by `SX` in horizontal and `SY` in vertical direction

`rotate A`
: rotate the coordinate system by an angle of `0<=A<=360` degrees

`setlinewidth X`
: set the line width to `X` units

`setlinecap N`
: set the line cap style (`0` = butt caps, `1` = round caps, `2` = projecting square caps)

`setlinejoin N`
: set the line join style (`0` = miter joins, `1` = round joins, `2` = bevel joins)

`setdash Xs DX`
: set the dash pattern (see below)

`setgray X`
: set the gray shade (`0` = black, `1` = white)

`setrgbcolor R G B`
: set the color in the RGB model (`R` = red, `G` = green, `B` = blue)

`sethsbcolor H S B`
: set the color in the HSB model (`H` = hue, `S` = saturation, `B` = brightness)

`setcmykcolor C M Y K`
: set the color in the CMYK model (`C` = cyan, `M` = magenta, `Y` = yellow, `K` = black)

`setcolor C`
: set the color specified by symbolic color value `C` (see below)

`setfont S X`
: select font `S` scaled by `X` units (see below)

The `setrgbcolor`, `sethsbcolor` and `setcmykcolor` functions enable you to select arbitrary colors in the RGB, HSB and CMYK model, respectively. A more user-friendly, but less flexible, routine `setcolor` is provided which allows colors to be selected from a fixed set of symbolic constants implemented by the `Color` type. Please refer to `graphics.q` for a list of the possible color values.

The `setdash` function sets the dash pattern for straight and curved line segments. The first argument of `setdash` is a list of length values which alternately specify the lengths of the "on" and "off" segments of the line (i.e., dashes and gaps between the dashes). This list is cycled through by the `stroke` function. For instance, `setdash [2,1] 0` specifies the dash pattern "2 on, 1 off, 2 on, 1 off, …". If the list is empty (`setdash [] 0`) `stroke` produces solid lines. The second argument of `setdash` denotes the "phase" of the dash pattern, which is given by an offset into the pattern. E.g., `setdash [2,3] 11` denotes the pattern "1 on, 3 off, 2 on, 3 off, 2 on, …".

The `setfont` function takes as its first argument a string denoting a PostScript font name such as `"Times-Roman"` or `"Helvetica-Oblique"`. The second argument denotes the size in units to which the font should be scaled. For instance: `setfont "Helvetica" 10`. (This function is implemented by a combination of the PostScript operators `findfont`, `scalefont` and `setfont`.)

11.13.7 Miscellaneous Operations

`showpage` emit the current page

`copypage` like `showpage`, but do not erase the contents of the current page

`flushpage`
update the display (flush any buffered graphics)

`erasepage`
erase the contents of the current page

`copies N` number of copies to be emitted with `showpage`

`psfile NAME`
copy a PostScript file to the graphics device

`psstr S` convert a string to PostScript syntax

`psheader` output a minimal PostScript header

`ps CMD` output a PostScript command

The `showpage`, `copypage`, `flushpage` and `erasepage` functions have already been discussed in Section 11.13.2 [Overview of Graphics Operations], page 111. The `copies` operation determines the number of copies which should be printed when `showpage` is invoked:

```
==> copies 4 || showpage
```

To submit a PostScript file to the graphics device the `psfile` operation may be used. It takes one string argument, the name of the file. For instance:

```
==> psfile "foo.ps"
```

The `ps` function is used to directly invoke a PostScript command. Examples can be found in Section 11.13.2 [Overview of Graphics Operations], page 111. The `psstr` function converts a string to PostScript format. It takes care of embedded backslashes and parentheses. For instance:

```
==> writes (psstr "(silly\\) example") || writec "\n"
(\(silly\\\) example)
()
```

The `psheader` function is used to begin the output file with a minimal header identifying the file as PostScript; see Section 11.13.8 [DSC and EPSF Comments], page 116.

11.13.8 DSC and EPSF Comments

Appendix G and H of [Adobe 1990] define a standard set of comments which should be included in PostScript documents to make explicit the document structure, and to allow inclusion of PostScript files in other documents. These standards are known as the *document structuring conventions* (DSC) and the *encapsulated PostScript file* (EPSF) format. See [Adobe 1990] for a detailed discussion of the purpose of these formats.

The `graphics.q` script currently does not provide any specialized operations for sending DSC and EPSF comments to the graphics device, with the exception of the `psheader` function which outputs a minimum header to make your printer recognize a graphics file as PostScript. Invoke this function as follows, *before* calling any other graphics operation:

```
==> psheader
```

You can also call the `psheader` function at initialization time from within a script, using a line like the following:

```
def INIT = psheader;
```

To include other types of DSC and EPSF comments in the graphics output, you have to specify these comments explicitly using the `ps` function. For instance, a function writing out the necessary DSC header comments of an EPS file may be implemented as follows:

```
/* (X1,Y1) and (X2,Y2) denote the lower left and the upper right corner
   of the bounding box, respectively */
epsf_header X1:Num Y1:Num X2:Num Y2:Num
                        = ps "%!PS-Adobe-3.0 EPSF-3.0\n" ||
                          ps ("%%BoundingBox: "++join " "
                              [str X1, str Y1, str X2, str Y2]++"\n");
```

As indicated, each comment *must* be terminated by a newline character. Use the `epsf_header` function in place of the `psheader` function if you are composing an EPS file which is to be used in other documents. E.g., when invoked as `epsf_header 5 5 105 105`, the following header information will be written to the graphics device:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 5 5 105 105
```

11.14 Diagnostics and Error Messages

The `assert.q` script supplies the special form `assert` for printing diagnostic messages:

```
foo X                   = assert (X>0) || bar (1/X);
```

The `assert` function verifies that the given expression evaluates to `true`, in which case it returns `()`. Otherwise it uses the `error` function to abort evaluation after printing an error message of the following form:

```
! Error: assertion (X) failed, value (value of X)
```

Error messages are printed using the `error` operation of the `error.q` script which prints an error message and then simply stops evaluation using `halt`. For instance:

```
X/0                     = error "Division by zero!";
```

This will print an error message

```
! Error: Division by zero!
```

when executed, and halt evaluation.

12 Clib

`Clib` is the "system" module of the Q programming language. In difference to the other standard library modules, `clib` is an *external* module, i.e., most functions are actually implemented in C (cf. Appendix C [C Language Interface], page 191). `Clib` includes some additional string operations, extended file functions, C-style formatted I/O, low-level and binary I/O, an interface to various system functions, POSIX thread functions, expression references, time functions, filename globbing and regular expression matching, additional integer functions from the GMP library, and, last not least, efficient C replacements for some common standard library list and string processing functions.

Even if you do not use the extra functionality provided by this module, you will benefit from the replacement operations, which considerably speed up basic list and string processing, sometimes by several orders of magnitude.

The module is implemented by the `clib.q` script which, as of release 3.0 of the Q programming system, is included in the default prelude.

NOTE: Not all of the following operations are implemented on all systems. The UNIX-specific operations are marked with the symbol '`(U)`' in the `clib.q` file. Only a portable subset of the UNIX system interface is provided, which encompasses the most essential operations found on many recent UNIX (and other POSIX) systems, as described by the ANSI C and POSIX standards as well as the Single UNIX Specification (SUS). These operations are also available on Linux and OSX systems.

12.1 Manifest Constants

`Clib` defines an abundance of symbolic values for use with various system functions. These vary from system to system; only the most common values are provided as global variables here. The variables are declared `const` (read-only) and are initialized at startup time. Flag values can be combined using bitwise logical operations as usual. A complete list of the variables can be found at the beginning of the `clib.q` file. Flag values which are unavailable on the host system will be set to zero, other undefined values to -1. Thus undefined values will generally have no effect or cause the corresponding operations to fail.

12.2 Additional String Functions

These functions provide an interface to some familiar character routines from the C library.

Character predicates: These work exactly like the corresponding C library routines.

```
public extern islower C, isupper C, isalpha C, isdigit C, isxdigit C,
  isalnum C, ispunct C, isspace C, isgraph C, isprint C, iscntrl C,
  isascii C;
```

String conversion: Convert a string to lower- or uppercase (like the corresponding C functions, but work on arbitrary strings, not just on single characters).

```
public extern tolower S, toupper S;
```

Examples

Count the number of alphanumeric characters in a text:

```
==> #filter isalnum (chars "The little brown fox.\n")
17
```

Convert a string to uppercase:

```
==> toupper "The little brown fox.\n"
"THE LITTLE BROWN FOX.\n"
```

12.3 Byte Strings

The following type represents unstructured binary data implemented as C byte vectors. This data structure is used by the low-level I/O functions and other `clib` functions which operate on binary data.

```
public extern type ByteStr;

public isbytestr B;                     // check for byte strings
```

Byte strings are like ordinary character strings, but they do not have a printable representation, and they may include zero bytes. (Recall that a zero byte in a character string terminates the string.) They can be used to encode arbitrary binary data such as C vectors and structures. The `bytestr` function can be used to construct byte strings from integers, floating point numbers, string values or lists of unsigned byte values:

```
public extern bytestr X;                // create a byte string
```

The `X` argument denotes the data to be encoded and can be either a list of byte values (unsigned integers in the range from 0 to 255), or an atomic data object, i.e., an integer, floating point number or string constant. In the latter case, the argument can also have the form `(X,SIZE)` indicating the desired byte size of the object; otherwise a reasonable default size is chosen. If the specified size differs from the actual size of `X`, the result is zero-padded or truncated accordingly. Integer values are encoded in the host byte order, with the least significant GMP limb first. Floating point values are encoded using double precision by default or if the byte count is sufficient (i.e., at least 8 on most systems), and using single precision otherwise.

Like ordinary character strings, byte strings can be concatenated, size-measured, indexed, sliced and compared lexicographically. Moreover, a byte string can be converted back to a (multiprecision) integer, floating point number, string value, or a list of byte values. For these purposes the following operations are provided.

```
public extern bcat Bs;                  // concatenate list of byte strings
public extern bsize B;                  // byte size of B
public extern byte I B;                 // Ith byte of B
public extern bsub B I J;               // slice of B (bytes I..J)
public extern bcmp M1 M2;               // compare M1 and M2

public extern bint B;                   // convert to unsigned integer
```

```
public extern bfloat B;                    // convert to floating point number
public extern bstr B;                      // convert to string
public ::list B as bytes;                  // convert to list
```

Note that the `bytes` symbol is just a synonym for `list`. This function is defined as follows:

```
bytes B:ByteStr                            = map (B!) (nums 0 (#B-1));
```

For convenience, the common string operators and the `sub` function are overloaded to work on byte strings as well. Thus `#B` returns the size of `B` (the number of bytes it contains) and `B!I` the `I`th byte of `B`. `B1++B2` concatenates `B1` and `B2`, `sub B I J` returns the slice from byte `I` to `J`, and the relational operators '`=`', '`<`', '`>`' etc. can be used to compare byte strings lexicographically. These operations are all implemented in terms of the functions listed above.

Examples

Encode an integer as a byte string, take a look at its individual bytes, and convert the byte string back to an integer:

```
==> hex

==> def B = bytestr 0x01020304; bytes B; bint B
[0x4,0x3,0x2,0x1]
0x1020304
```

(Note that this result was obtained on a little-endian system, hence the least significant byte `0x04` comes first in the byte list.)

Negative integers are correctly encoded in 2's complement:

```
==> def B = bytestr (-2); bytes B; bint B
[0xfe,0xff,0xff,0xff]
0xfffffffe
```

To work with these binary representations you must be aware of the way GMP represents multiprecision integers. In particular, note that the default size of an integer is always a multiple (at least one) of GMP's limb size which is usually 4 or 8 bytes depending on the host system's default `long` integer type. The actual limb size can be determined as follows:

```
==> #bytes (bytestr 0)
```

In order to get integers of arbitrary sizes, an explicit `SIZE` argument may be used. For instance, here is how we encode small (1 or 2 byte) integers:

```
==> bytes (bytestr (0x01,1)); bytes (bytestr (0x0102,2))
[0x1]
[0x2,0x1]
```

We remark that the host system's byte sizes of various atomic C types can be determined with symbolic values declared at the beginning of `clib.q`, such as `SIZEOF_CHAR`, `SIZEOF_SHORT`, `SIZEOF_LONG`, `SIZEOF_FLOAT` and `SIZEOF_DOUBLE`.

Another fact worth mentioning is that even on big-endian systems, integers are always encoded with the "least significant limb" first. So, for instance, given that the limb size is 4, as in the above examples, the 2-limb integer `0x0102030405060708` consists of bytes `0x8 0x7 0x6 0x5 0x4 0x3 0x2 0x1` on a little-endian system, in that order, whereas the byte order on a big-endian system is `0x5 0x6 0x7 0x8 0x1 0x2 0x3 0x4`.

Here is how we can quickly check the byte order of the host system:

```
==> hd (bytes (bytestr 1))
```

This expression returns 1 on a little-endian system and zero otherwise.

As long as an integer does not exceed the machine's word size (which usually matches the limb size), we can simply convert between big-endian and little-endian representation by reversing the byte list:

```
==> bytestr (reverse (bytes B))
```

Floating point values can be encoded either in double or single precision, depending on the `SIZE` argument. The default size is double precision (usually 8 bytes).

```
==> bfloat (bytestr (1/3)); bfloat (bytestr (1/3,SIZEOF_FLOAT))
0.333333333333333
0.333333343267441
```

The default size of the encoding of a character string is the length of the string. If an explicit size is given, the string is zero-padded or truncated if necessary.

```
==> dec

==> def S1 = bytestr "ABC", S2 = bytestr ("ABC",2), S3 = bytestr ("ABC",5)

==> bytes S1; bytes S2; bytes S3
[65,66,67]
[65,66]
[65,66,67,0,0]

==> bstr S1; bstr S2; bstr S3
"ABC"
"AB"
"ABC"
```

By combining elements like the ones above, and including appropriate "tagging" information, more complex data structures can be represented as binary data as well. For this purpose, the byte strings of the tags and the data elements can be concatenated with `bcat` or the '`++`' operator. This is useful, in particular, for compact storage of objects in files. Moreover, some system functions involve binary data which might represent C structures and/or vectors. Such data can be assembled from the constituent parts by simply concatenating them. For instance, consider the following C struct:

```
struct { char foo[108]; short bar; int baz; };
```

A value of this type, say `{"Hello, world.", 4711, 123456}`, can then be encoded as follows:

```
==> bytestr ("Hello, world.",108) ++ bytestr (4711,SIZEOF_SHORT) ++ \
bytestr (123456,SIZEOF_INT)
```

Similarly, a list of integers can be converted to a corresponding C vector as follows:

```
==> bcat (map bytestr (nums 1 100))
```

When encoding such C structures you must also consider alignment issues. For instance, most C compilers will align non-byte data at even addresses.

12.4 Extended File Functions

`Clib` provides the following enhanced and additional file functions:

```
public extern ::fopen NAME MODE, fdopen FD MODE, freopen NAME MODE F;
public extern fileno F;
public extern setvbuf F MODE;
public extern tmpnam, tmpfile;
public extern ftell F, fseek F POS WHENCE;
public rewind F;
public extern gets, fgets F;
public extern fget F;
public extern ungetc C, fungetc F C;
```

The `fopen` version of `clib` handles the '`+`' flag in mode strings, thus enabling you to open files for both reading and writing. The mode `"r+"` opens an existing file for both reading and writing; the initial file contents are unchanged, and both the input and output file pointers are positioned at the beginning of the file. The `"w+"` mode creates a new file, or truncates it to zero size if it already exists, and positions the file pointers at the beginning of the file. The `"a+"` mode appends to an existing file (or creates a new one); the initial file pointer is set at the beginning of the file for reading, and at the end of the file for writing. All these modes also work in combination with the `b` (binary file) flag.

The `freopen` function is like `fopen`, but reopens an existing file object on another file. Just as in C programming, the main purpose of this operation is to enable the user to redirect the standard I/O streams associated with the interpreter process (available in the interpreter by means of the `INPUT`, `OUTPUT` and `ERROR` variables).

The `fdopen` function opens a new file object on a given file descriptor, given that the mode is compatible. Conversely, the `fileno` function returns the file descriptor of a file object. (See also the functions for direct file descriptor manipulation in Section 12.8 [Low-Level I/O], page 134.)

The `setvbuf` function sets the buffering mode for a file (`IONBF` = no buffering, `IOLBF` = line buffering, `IOFBF` = full buffering). This operation should be invoked right after the file has been opened, *before* any I/O operations are performed.

The `tmpnam` and `tmpfile` functions work just like the corresponding C routines: `tmpnam` returns a unique name for a temporary file, and `tmpfile` constructs a temporary file opened in `"w+b"` mode, which will be deleted automatically when it is closed. See the `tmpnam`(3) and `tmpfile`(3) manual pages for details.

The `ftell`/`fseek` functions are used for file positioning. The `ftell` function returns the current file position, while `fseek` function positions the file at the given position. The `rewind` function provides a convenient shorthand for repositioning the file at the beginning. These operations work just like the corresponding C functions. The `WHENCE` argument of `fseek` determines how the `POS` argument is to be interpreted; it can be either `SEEK_SET` (`POS` is relative to the beginning of the file, i.e., an absolute position), `SEEK_CUR` (`POS` is relative to the current position) or `SEEK_END` (`POS` is relative to the end of the file). In the latter two cases `POS` can also be negative.

Portability Notes:

- According to the ISO C specification, you should use `fflush` or `fseek` before switching between reading and writing on a file opened with the '`+`' flag.
- On non-UNIX systems, `ftell` and `fseek` might only work reliably if the file is opened in binary mode (`b` flag).

The `gets`/`fgets` functions work like the C `fgets` function, i.e., they read a line from standard input or the given file *including* the trailing newline, if any. The `fget` function reads an entire file at once and returns it as a string. The `ungetc`/`fungetc` functions push back a single character on standard input or the given input file, like the C `ungetc` function.

Moreover, the following additional aliases are provided for C afficionados:

```
public ::readc as getc, ::freadc F as fgetc;
public ::writes S as puts, ::fwrites F S as fputs;
public ::writec C as putc, ::fwritec F C as fputc;
```

Examples

Open a new file for both reading and writing:

```
==> def F = fopen "test" "w+"
```

Write a string to the file:

```
==> fwrites F "The little brown fox.\n"
()
```

Current position is behind written string (at end-of-file):

```
==> ftell F
22
```

Rewind (go to the beginning of the file):

```
==> rewind F
()
```

Read back the string we've written before:

```
==> fgets F
"The little brown fox.\n"
```

Check that we're again at end-of-file:

```
==> feof F
true
```

Output another string:

```
==> fwrites F "The second line.\n"
()
```

Position behind the first string:

```
==> fseek F 22 SEEK_SET
()
```

Reread the second string:

```
==> fgets F
"The second line.\n"
```

And here's how to read an entire text file at once:

```
==> def T = fget (fopen "clib.q" "r")
```

To quickly compute a 32 bit checksum of the file:

```
==> sum (map ord (chars T)) mod 0x100000000
2851508
```

Finally, let's split the text into lines and add line numbers using `sprintf` (see Section 12.5 [C-Style Formatted I/O], page 125):

```
==> def L = split "\n" T

==> def L = map (sprintf "%3d: %s\n") (zip (nums 1 (#L)) L)

==> do writes (take 5 L)
  1:
  2: /* clib.q: Q's system module */
  3:
  4: /* This file is part of the Q programming system.
  5:
()
```

12.5 C-Style Formatted I/O

These functions provide an interface to the C `printf` and `scanf` routines:

```
public extern printf FORMAT ARGS, fprintf F FORMAT ARGS,
  sprintf FORMAT ARGS;
public extern scanf FORMAT, fscanf F FORMAT, sscanf S FORMAT;
```

Arguments to the `printf` routines and the results of the `scanf` routines are generally encoded as tuples or single non-tuple values (if only one item is read/written).

All the usual conversions and flags of C `printf`/`scanf` are supported, except `%p` (pointer conversion). The basic `h` and `l` length modifiers are also understood, but not the fancy ISO C99 extensions like `ll` or `hh`, or `l` modifiers on characters and strings. Two further unsupported features of the `printf` functions are the `%n` (number of written characters) conversion and explicit argument indexing (`m$`); thus all arguments have to be in the same order as specified in the `printf` format string. The `%n` conversion *is* implemented for the `scanf` functions, though.

As these functions are simply wrappers for the corresponding C functions, integer conversions are generally limited to values which fit into machine integers. To handle integers of arbitrary sizes, you might treat them as strings (`%s`) in the format string and do the actual conversion manually with `val` or `str`.

Seasoned C programmers will appreciate that the wrapper functions provided here are *safe* in that they check their arguments and prevent buffer overflows, so they should never crash your program with a segfault. To these ends, if a `%s` or `%[...]` conversion without maximum field width is used with `scanf`, the field width will effectively be limited to some (large) value chosen by the implementation.

Examples

See the `printf`(3) and `scanf`(3) manual pages for a description of the format string syntax. Some basic examples follow (`<CR>` indicates that you hit the carriage return key to terminate a line):

```
==> printf "%d\n" 99
99
()

==> printf "%d\n" (99)
99
()

==> printf "%s %s %d\n" ("foo","bar",99)
foo bar 99
()

==> scanf "%d"
99<CR>
99

==> scanf "%s %s %d"
foo bar 99<CR>
```

```
("foo","bar",99)
```

As indicated, multiple values are denoted as tuples, and the `printf` function accepts both a single value or a one-tuple for a single conversion. The `scanf` function always returns a single, non-tuple value if only a single conversion is specified. Zero items are represented using the empty tuple. Note that you always have to supply the `ARGS` argument of `printf`, thus you specify an empty tuple if there are no output conversions:

```
==> printf "foo\n" ()
foo
()
```

The `scanf` function also returns an empty tuple if no input items are converted. For instance (as usual, using the `*` flag with a `scanf` conversion suppresses the corresponding input item):

```
==> scanf "%*s"
foo<CR>
()
```

Note that while `scanf` for most conversions skips an arbitrary amount of leading whitespace, the trailing whitespace character at which a conversion stops is *not* discarded by `scanf`. You can notice this if you invoke, e.g., `readc` afterwards:

```
==> scanf "%s %d"; readc
foo 99<CR>
("foo",99)
"\n"
```

If you really have to skip the trailing whitespace character, you can do this with a suppressed character conversion, e.g.:

```
==> scanf "%s %d%*c"; writes "input: "||reads
foo 99<CR>
("foo",99)
input: <reads function waiting for input here>
```

The `fprintf/fscanf` functions work analogously, but are used when writing or reading an arbitrary file instead of standard output or input. For instance:

```
==> def msg = "You're not supposed to do that!"

==> fprintf ERROR "Error: %s\n" msg
Error: You're not supposed to do that!
()
```

The `sprintf` function returns the formatted text as a string instead of writing it to a file:

```
==> sprintf "%s %s %d\n" ("foo","bar",99)
"foo bar 99\n"
```

Likewise, `sscanf` takes its input from a string:

```
==> sscanf "foo bar 99\n" "%s %s %d"
```

```
("foo","bar",99)
```

The `%n` conversion is especially useful with `sscanf`, since it allows you to determine the number of characters which were actually consumed:

```
==> sscanf "foo bar 99 *** extra text here ***\n" "%s %s %d%n"
("foo","bar",99,10)
```

You might then use the character count, e.g., to check whether the input format matched the entire string, or whether there remains some text to be processed.

Some remarks about the role of the length modifiers `h` and `l` are in order. Just as with the C `scanf` routines, you need the `l` modifier to read a double precision value; a simple `%f` will only read single precision number:

```
==> scanf "%f"
1e100<CR>
inf

==> scanf "%lf"
1e100<CR>
1e+100
```

The `printf` functions, however, always print double precision numbers, so the `l` modifier is not needed:

```
==> sprintf "%g" 1e100
"1e+100"
```

For the integer conversions, the `h` and `l` modifiers denote short (usually 2 byte) and long (usually 4 byte) integer values. If the modifier is omitted, the default integer type is used (this usually is the same as `long`, but your mileage may vary).

As already indicated, the `printf` and `scanf` routines are limited to machine integer sizes. Thus a `scanf` integer conversion will always return a short or long integer value, depending on the length modifier used. If a `printf` integer conversion is applied to a "big" integer value, only the least significant bytes of the value are printed, as if the printed number (represented in 2's complement if negative) had been cast to the corresponding integer type in C. Thus the printed result will be consistent with C `printf` output under all circumstances. For instance:

```
==> def N = 0xffff70008000 // big number

==> printf "%hu %lu\n" (N,N)
32768 1879080960
()

==> printf "%hd %ld\n" (N,N)
-32768 1879080960
()
```

To correctly print a big integer value, you can convert it manually with Q's builtin `str` function, then print the value using a `%s` conversion:

```
==> printf "%s\n" (str 1234567812345678)
1234567812345678
()
```

Similarly, you can read a big integer value by converting it as a string, and then apply the val builtin.

```
==> val (scanf "%s")
1234567812345678<CR>
1234567812345678
```

Here you might use the `%[...]` conversion to ensure that the number is in proper format (the initial blank is needed here to skip any leading whitespace):

```
==> val (scanf " %[0-9-]")
-1234567812345678
-1234567812345678
```

On output, the integer and floating point conversions can all be used with either integer or floating point arguments; integers will be converted to floating point values and vice versa if necessary:

```
==> printf "An integer: %d\n" 99.9
An integer: 99
()

==> printf "A floating point value: %e\n" 99
A floating point value: 9.900000e+01
()
```

12.6 File and Directory Functions

These functions provide the same functionality as their C counterparts.

```
public extern rename OLD NEW;          // rename a file
public extern unlink NAME;             // delete a file
public extern truncate NAME LEN;       // truncate a file (U)
public extern getcwd, chdir NAME;      // get/set the working directory
public extern mkdir NAME MODE;         // create a new directory
public extern rmdir NAME;              // remove a directory
public extern readdir NAME;            // list the files in a directory
public extern link OLD NEW;            // create a hard link (U)
public extern symlink OLD NEW;         // create a symbolic link (U)
public extern readlink NAME;           // read a symbolic link
public extern mkfifo NAME MODE;        // create a named pipe (U)
public extern access NAME MODE;        // test access mode
public extern chmod NAME MODE;         // set the file mode
public extern chown NAME MODE UID GID; // set file ownership (U)
public extern lchown NAME MODE UID GID; // set link ownership (U)
public extern utime NAME TIMES;        // set the file times
public extern umask N;                 // set/get file creation mask
public extern stat NAME, lstat NAME;   // file and link information
```

The stat/lstat functions return a tuple consisting of the commonly available fields of the C stat struct, see stat(2). For your convenience, the following mnemonic functions are provided for accessing the different components:

```
public st_dev STAT, st_ino STAT, st_mode STAT, st_nlink STAT,
  st_uid STAT, st_gid STAT, st_rdev STAT, st_size STAT, st_atime STAT,
  st_mtime STAT, st_ctime STAT;
```

Examples

```
==> mkdir "tmp" 0777||chdir "tmp"||mkfifo "foo" 0666||\
rename "foo" "bar"||unlink "bar"||chdir ".."||rmdir "tmp"
()
```

(Create a tmp subdirectory, change to it, create a new FIFO special file, rename that file, delete it, change back to the original directory, and remove the tmp directory. All with a single expression which realizes identity.)

Now for something more useful. We can retrieve the current umask while setting it to zero, and then reset it to the original value as follows:

```
==> def U = umask 0; oct; umask U || U; dec
022
```

List the files in the current directory:

```
==> readdir "."
[".","..","Makefile","givertcap","clib.c","clib.q","Makefile.am",
"Makefile.in","README-Clib","examples","Makefile.mingw"]
```

Get the size of a file:

```
==> st_size (stat "README-Clib")
355
```

12.7 Process Control

The system function returns the status code of the command if execution was successful, and fails otherwise:

```
public extern system CMD;              // exec command using the shell
```

Clib also provides the usual UNIX process creation and management routines. Most of these really require a UNIX system; no attempt is made to emulate operations like fork on systems where they are not implemented. Thus the only process operations which currently work under Windows are system, exec, spawn, _spawn, exit and getpid.

```
public extern fork;                    // fork a child process (U)
public extern exec PROG ARGS;          // execute program
public extern spawn PROG ARGS;         // execute program in child process
public extern _spawn MODE PROG ARGS;   // execute child with options
public extern nice INC;                // change nice value (U)
public extern exit N;                  // exit process with given exit code
```

```
public extern pause;                    // pause until a signal occurs (U)
public extern raise SIG;                // raise signal in current process
public extern kill SIG PID;             // send signal to given process (U)
public extern getpid;                   // current process id
public extern getppid;                  // parent's process id (U)
public extern wait;                     // wait for any child process (U)
public extern waitpid PID OPTIONS;      // wait for given child process (U)
```

All these operations are simply wrappers for the corresponding C library routines. Note, however, that the `kill` function takes the signal to send as its *first* argument, which makes it easier to use the function in a curried form, e.g., to iterate a kill operation over a list of process numbers (as in '`do (kill SIGTERM) PIDs`').

The `exec` function performs a path search like the C `execlp`/`execvp` function; the parameters for the program are given as a string list `ARGS`, and as usual the first argument should repeat the program file name. This function never returns unless it fails. The `spawn` and `_spawn` operations are provided to accommodate Windows' lack of `fork` and `wait`; these functions work on both UNIX and Windows. The `spawn` function works like `exec`, but runs the program in a new child process. It returns the new process id (actually the process handle under Windows). The `_spawn` function is like `spawn`, but accepts an additional `MODE` parameter which determines how the child is to be executed, either `P_WAIT` (wait for the child, return its exit status), `P_NOWAIT` (do not wait for the child, same as `spawn`), `P_OVERLAY` (replace the current image with the new process, same as `exec`) and `P_DETACH` (run the new process in the background). (Note that the `P_DETACH` option is ignored on UNIX systems; the correct way to code a "daemon" on UNIX is shown in the examples section below.)

On UNIX, the following routines are provided to interpret the status code returned by the `system`, `_spawn`, `wait` and `waitpid` functions:

```
public extern isactive STATUS;
// process is active
public extern isexited STATUS, exitstatus STATUS;
// process has exited normally, get its exit code
public extern issignaled STATUS, termsig STATUS;
// process was terminated by signal, get the signal number
public extern isstopped STATUS, stopsig STATUS;
// process was stopped by signal, get the signal number
```

For more information about the process functions, we refer the reader to the corresponding UNIX manual pages.

Operations to access the process environment are also implemented. The `getenv` function fails if the given variable is not set in the environment; this lets you distinguish this error condition from a defined variable with empty value. The `setenv` function overwrites an existing definition of the given variable:

```
public extern getenv NAME, setenv NAME VAL;
// get/set environment variables
```

On UNIX, the following operations provide access to process user and group information, as well as process groups and sessions. Not all operations may be implemented on all UNIX flavours. Please see the UNIX manual for a description of these functions.

```
/* User/group-related functions (U). */

public extern setuid UID, setgid GID;   // set user/group id of process
public extern seteuid UID, setegid GID; // set effective user/group id
public extern setreuid RUID EUID, setregid RGID EGID;
                                        // set real and effective ids
public extern getuid, geteuid;          // get real/effective user id
public extern getgid, getegid;          // get real/effective group id
public extern getlogin;                 // get real login name

// get/set supplementary group ids of current process
public extern getgroups, setgroups GIDS;

/* Session-related routines (U). */

public extern getpgid PID, setpgid PID PGID;    // get and set process group
public extern getpgrp, setpgrp;                 // dito, for calling process
public extern getsid PID;                       // get session id of process
public extern setsid;                           // create a new session
```

Examples

Invoke the `system` function to execute a shell command:

```
==> system "ls -l"
```

You can also run the program directly with the `spawn` function:

```
==> spawn "ls" ["ls","-l"]
```

Get and set an environment variable:

```
==> getenv "HOME"
"/home/ag"

==> getenv "MYVAR" // variable is undefined
getenv "MYVAR"

==> setenv "MYVAR" "foo bar"
()

==> getenv "MYVAR"
"foo bar"
```

Here are some examples demonstrating the use of named pipes and the process functions on UNIX systems. The `mkfifo` function allows the creation of so-called “FIFO special files” a.k.a. *named pipes*, which provide a simple inter-process communication facility.

For instance, create a named pipe as follows:

```
==> mkfifo "pipe" 0666
()
```

You can then open the writeable end of the pipe:

```
==> def OUT = fopen "pipe" "w"
```

Note that this call blocks until the input side of the pipe has been opened. For this purpose, start another instance of the interpreter (e.g., in another `xterm`), and from there open the pipe for reading:

```
==> def IN = fopen "pipe" "r"
```

Both `fopen` calls should now have finished, and you can write something to the output end of the pipe in the first instance of the interpreter:

```
==> fwrites OUT "Hello, there!\n"
```

Go to the other interpreter instance, and read back the string from there:

```
==> freads IN
"Hello, there!"
```

As usual, each end of the pipe is closed as soon as the corresponding file object is no longer accessible. When you close the writeable end of the pipe using, e.g., `undef OUT` in the first instance of the interpreter, the input side of the pipe will reach end-of-file, and thus `feof IN` will become `true`. After closing the pipe also on the input side, you can remove the FIFO special file with the `unlink` function.

You can also use named pipes to set up a communication channel to child processes created with `fork`. For instance:

```
def NAME = tmpnam;
def PIPE = mkfifo NAME 0666;
def MSG  = "Hello there!\n";

test     = printf "Parent writes: %s" MSG ||
           fwrites (fopen NAME "w") MSG ||
           writes "Parent waits for child ...\n" ||
           printf "Parent: child has exited with code %d\n" wait
               if fork > 0;
         = printf "Child reads: %s\n" (freads (fopen NAME "r")) ||
           writes "Child exiting ...\n" || exit 0
               otherwise;


==> test
Parent writes: Hello there!
Parent waits for child ...
Child reads: Hello there!
Child exiting ...
Parent: child has exited with code 0
()
```

```
==> unlink NAME
()
```

Another method for accomplishing this with anonymous pipes is discussed in Section 12.8 [Low-Level I/O], page 134.

On UNIX, it also possible to implement "daemons", i.e., processes which place themselves in the background and continue to run even when you log out. The following little script shows how to do this.

```
/* Becoming a daemon is easy: Just fork, have the parent exit, and call setsid
   in the child to start a new session. The new process becomes a child of the
   init process and has no controlling terminal. Thus it keeps running even if
   you log out, until it gets killed or the system shuts down. */

daemon          = setsid || main if fork = 0;
                = exit 0 otherwise;

/* The main code of the daemon then closes file descriptors inherited by the
   parent and starts executing. In this example we just open a logfile and
   start logging messages in regular intervals. We also handle the condition
   that we are terminated by a signal. */

main            = do close [0,1,2] || log F "daemon started" ||
                  do (trap 1) [SIGINT, SIGTERM, SIGHUP, SIGQUIT] ||
                  catch (sig F) (loop F) where F:File = fopen "log" "w";
                = perror "daemon" || exit 1 otherwise;

sig F (syserr SIG)
                = log F (sprintf "daemon stopped by signal %d" (-SIG)) ||
                  exit 0;

loop F          = sleep 5 || log F "daemon still alive" || loop F;

log F MSG       = fprintf F "%s at %s" (MSG, ctime time) || fflush F;
```

12.8 Low-Level I/O

These functions provide operations for direct manipulation of files on the file descriptor level. For a closer description of the following operations we refer the reader to the corresponding UNIX manual pages.

```
public extern open NAME FLAGS MODE;     // create a new descriptor
public extern close FD;                 // close a descriptor
public extern dup FD, dup2 OLDFD NEWFD; // duplicate a descriptor
public extern pipe;                     // create an unnamed pipe
public extern fstat FD;                 // stat descriptor
public extern fchdir FD;                // change directory (U)
public extern fchmod FD MODE;           // change file mode (U)
public extern fchown FD UID GID;        // set file ownership (U)
```

```
public extern ftruncate FD LEN;         // truncate a file (U)
public extern fsync FD, fdatasync FD;   // sync the given file (U)
```

The following operations can be used on both file descriptors and file objects. They read and write binary data represented as byte strings (see Section 12.3 [Byte Strings], page 120), providing an interface to the system's `read`/`write`(2) and `fread`/`fwrite`(3) functions. The `bread` function returns a byte string of the given size read from the given file. Note that the returned byte string may actually be shorter than `SIZE` bytes because, e.g., end of file has been reached or not enough input was currently available on a pipe. The `bwrite` function returns the number of bytes actually written which is usually the size of the byte string unless an error occurs. The functions fail if an error occurred before anything was read or written. It is the application's responsibility to check these error conditions and handle them in an appropriate manner.

```
public extern bread FD SIZE;            // read a byte string
public extern bwrite FD DATA;           // write a byte string
```

For instance, the following function uses `bread` and `bwrite` to copy an input to an output file, using chunks of 8192 bytes at a time:

```
fcopy F G               = () if bwrite G (bread F 8192) < 8192;
                        = fcopy F G otherwise;
```

The file pointer of a descriptor can be positioned with `lseek`. In difference to `fseek`, this function returns the new offset. To determine the current position you can hence use an expression like '`lseek FD 0 SEEK_CUR`'.

```
public extern lseek FD POS WHENCE;
```

Some terminal-related routines are also provided:

```
public extern isatty FD;                // is descriptor a terminal?
```

The following are UNIX-specific:

```
public extern ttyname FD;               // terminal associated with descriptor
public extern ctermid;                  // name of controlling terminal
public extern openpty, forkpty;         // pseudo terminal operations
```

The `openpty` function returns a pair `(MASTER, SLAVE)` of file descriptors opened for both reading and writing on a "pseudo terminal". `MASTER` is to be used in the controlling process, while `SLAVE` can be used for the standard I/O streams in a child process. The `forkpty` function combines `openpty` with `fork` and makes the slave device the controlling terminal of the child process; it returns a pair `(PID, MASTER)`, where `PID` is zero in the child process and the process id of the child in the parent, and `MASTER` is the master end of the pseudo terminal to be used by the parent. These functions are commonly used to implement applications which drive other programs through a terminal emulation interface.

On UNIX systems, `clib` also provides access to the following `fcntl` operation (see Section 2 of the UNIX manual):

```
public extern fcntl FD CMD ARG;
```

The `ARG` parameter of `fcntl` depends on the type of command `CMD` which is executed. The available command codes and other relevant values are defined as global variables, as listed below. Flags are bitwise disjunctions of the symbolic values listed below. (The following are the values present on most systems. Specific implementations may provide additional flags.)

```
public var const
  // fcntl command codes
  F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL,
  F_GETLK, F_SETLK, F_SETLKW,

  // lock types
  F_RDLCK, F_WRLCK, F_UNLCK,

  // file access modes and access mode bitmask
  O_RDONLY, O_WRONLY, O_RDWR, O_ACCMODE,

  // file descriptor flags
  FD_CLOEXEC,

  // status flags
  O_CREAT, O_EXCL, O_TRUNC, O_APPEND, O_NONBLOCK, O_NDELAY, O_NOCTTY,
  O_BINARY;
```

The following types of commands are implemented:

```
fcntl FD F_DUPFD ARG                      // duplicate a file descriptor

fcntl FD F_GETFD ()                       // get file descriptor flags
fcntl FD F_SETFD FLAGS                    // set file descriptor flags

fcntl FD F_SETFD ()                       // get status flags/access mode
fcntl FD F_SETFD FLAGS                    // set status flags

fcntl FD F_GETLK (TYPE,POS,LEN[,WHENCE])  // query file lock information
fcntl FD F_SETLK (TYPE,POS,LEN[,WHENCE])  // set an advisory file lock
fcntl FD F_SETLKW (TYPE,POS,LEN[,WHENCE]) // blocking variant of F_SETLK
```

The first five commands serve to duplicate descriptors and to retrieve and change the file descriptor and status flags. The remaining commands are used for advisory file locking. A file lock is specified as a triple `(TYPE, POS, LEN)` or quadrupel `(TYPE, POS, LEN, WHENCE)`, where `TYPE` is the type of lock (`F_RDLCK`, `F_WRLCK` or `F_UNLCK` for read locks, write locks and unlocking, respectively), `POS` the position in the file, `LEN` the number of bytes to be locked (0 means up to the end of the file) and `WHENCE` specifies how the `POS` argument is to be interpreted. (This parameter has the same meaning as for the `fseek` and `lseek` functions, see above. If `WHENCE` is omitted, it defaults to `SEEK_SET`, i.e., absolute positions.) The value returned by the `F_GETLK` command is the lock description with `TYPE` set to `F_UNLCK` if the given lock would be accepted, and the description of a current lock blocking the lock request otherwise. (In the latter case the return value is actually a quadrupel, with the id of a process currently owning a conflicting lock in the last component.)

Note that the standard I/O operations use buffered I/O by default which might interfere with record locking. Therefore in applications requiring individual record locking you should work with the low-level operations (`open`, `bwrite`, etc.) instead.

The following `select` function waits for a set of files to change I/O status. Note that this operation is available on Windows as part of the socket interface, but it only applies to sockets there.

```
public extern select FILES;
```

The input is a tuple `(IN, OUT, ERR, TIMEOUT)` consisting of three lists of file descriptors and/or file objects to be watched, and an optional integer or floating point value specifying a timeout in seconds. The function returns as soon as either a member of `IN` or `OUT` becomes ready for performing an I/O operation (without blocking), or an error condition is signaled for a member of `ERR`. The returned value is a triple `(IN, OUT, ERR)` with all the members of the original lists which are now ready for I/O. If the timeout is exceeded before any of the files has become ready, a triple of three empty lists is returned. If no timeout is specified then the function may block indefinitely.

Examples

These examples are mostly UNIX-specific, thus Windows users might wish to skip ahead.

Fcntl

The following definitions show how the `fcntl` function can be used to change a file's "non-blocking" flag. This is useful, e.g., if we want to read from standard input or a pipe but do not want to be blocked until input becomes available. Instead, having set the non-blocking flag, input operations will fail immediately if there is no input to be read right now.

```
/* set and clear the O_NONBLOCK flag of a file */
set_nonblock FD:Int     = fcntl FD F_SETFL (FLAGS or O_NONBLOCK)
                            where FLAGS = fcntl FD F_GETFL ();
clr_nonblock FD:Int     = fcntl FD F_SETFL (FLAGS and not O_NONBLOCK)
                            where FLAGS = fcntl FD F_GETFL ();
```

And here is how we can perform advisory locking on an entire file.

```
/* place an advisory read or write lock on an entire file (fail if error) */

rdlock FD:Int           = () where () = fcntl FD F_SETLK (F_RDLCK, 0, 0);
wrlock FD:Int           = () where () = fcntl FD F_SETLK (F_WRLCK, 0, 0);

/* remove the lock from the file */

unlock FD:Int           = () where () = fcntl FD F_SETLK (F_UNLCK, 0, 0);

/* predicates to check whether a read or write lock could be placed */

rdlockp FD:Int          = (LOCK!0 = F_UNLCK)
```

```
                              where LOCK:Tuple =
                                fcntl FD F_GETLK (F_RDLCK, 0, 0);
wrlockp FD:Int          = (LOCK!0 = F_UNLCK)
                              where LOCK:Tuple =
                                fcntl FD F_GETLK (F_WRLCK, 0, 0);
```

Note that to apply these functions to standard file objects you can use the `fileno` function (see Section 12.4 [Extended File Functions], page 123) as follows:

```
==> rdlock (fileno F)
```

Select

The `select` function accepts both files and file descriptors as input. Here is a way to test whether input is currently available from a file/descriptor:

```
avail F                 = not null (select ([F],[],[],0)!0);
```

This is useful, in particular, if the file is actually a pipe. For instance:

```
==> def F = popen "sleep 5; echo done" "r"

==> avail F   // no input to be read yet, wait ...
false

==> avail F   // ... input available now
true

==> fget F
"done\n"
```

However, most of the time `select` is used for multiplexing I/O operations. For instance, the following loop processes input from a set of files, one line at a time:

```
loop FILES              = loop (proc FILES F)
                              where ([F|_],_,_) = select (FILES,[],[])
                              if not null FILES;
                        = () otherwise;

proc FILES F            = // done with this file, get rid of it
                          filter (neq F) FILES
                              if feof F;
                        = // process a line
                          writes (fgets F) || FILES;
```

Anonymous Pipes

The `pipe` and `dup2` operations provide a quick way to reassign input and output of a child process and connect it to corresponding file objects in the parent. For instance, here's how we can implement a `popen2` function which works like the built-in `popen` routine, but allows to redirect *both* the input and output side of a child process:

```
/* Create two unnamed pipes, one for the parent to read and the child to
   write, the other one for the child to read and the parent to write. */

popen2 CMD      = spawn2 CMD (P_IN, P_OUT) (C_IN, C_OUT)
                    where (P_IN, C_OUT) = pipe, (C_IN, P_OUT) = pipe;

/* Fork the child and redirect its standard input and output streams to the
   child's ends of the pipe. This is accomplished with dup2 SRC DEST which
   closes the file descriptor DEST and then makes DEST a copy of SRC. In the
   parent we use fdopen to open two new file objects for the parent's ends of
   the pipes. */

spawn2 CMD (P_IN, P_OUT) (C_IN, C_OUT)
                = close P_IN || close P_OUT ||
                  dup2 C_IN (fileno INPUT) || dup2 C_OUT (fileno OUTPUT) ||
                  exec "/bin/sh" ["/bin/sh", "-c", CMD]
                    if fork = 0;
                = close C_IN || close C_OUT ||
                  (fdopen P_IN "r", fdopen P_OUT "w")
                    otherwise;
```

The `popen2` function employs `fork` and `exec` to spawn a child process which executes the given command using the shell, after redirecting the child's input and output to two pipes. In the parent process, the `popen2` function returns a pair of files opened on the other ends of the child's descriptors.

The following piece of Q code shows how to apply the `popen2` function defined above in order to pipe a string list into the UNIX `sort` program and construct the sorted list from the output:

```
mysort STRL     = do (fprintf OUT "%s\n") STRL || fclose OUT || digest IN
                    where (IN, OUT) = popen2 "sort";

digest IN       = [] if feof IN;
                = [freads IN|digest IN] otherwise;
```

Example:

```
==> mysort ["five","strings","to","be","sorted"]
["be","five","sorted","strings","to"]

==> wait // get exit code of child process (sort program)
(15804,0)
```

12.9 Terminal Operations

The following (UNIX-specific) operations provide an interface to the POSIX termios interface. Terminal attributes are stored in a "termios" structure, represented as a 7-tuple `(IFLAG, OFLAG, CFLAG, LFLAG, ISPEED, OSPEED, CC)`. The control character set `CC` is represented as a list of character numbers, indexed by the symbolic constants `VEOF` etc. See `termios`(3) for further details.

```
public extern tcgetattr FD;             // get terminal attributes
public extern tcsetattr FD WHEN ATTR;   // set terminal attributes

public extern tcsendbreak FD DURATION;  // send break
public extern tcdrain FD;               // wait until all output finished
public extern tcflush FD QUEUE;         // flush input or output queue
public extern tcflow FD ACTION;         // control input/output flow

public extern tcgetpgrp FD;             // get terminal process group
public extern tcsetpgrp FD PGID;        // set terminal process group

/* Access components of the termios structure. */

public c_iflag ATTR, c_oflag ATTR, c_cflag ATTR, c_lflag ATTR,
  c_ispeed ATTR, c_ospeed ATTR, c_cc ATTR;
```

Example

This example shows how to use the `termios` functions to read a password from the terminal without echoing. This is an almost literal translation of the C program described in Richard Stevens: Advanced Programming in the UNIX Environment, Addison-Wesley, 1993, cf. p. 350. The main difference is that we merely ignore the `SIGINT` and `SIGTSTP` signals instead of blocking them (the latter is not supported by Q's `trap` builtin).

```
getpass PROMPT  = fwritec F "\n" || unprep F SAVE || PW
                    where F:File = fopen ctermid "r+", SAVE = prep F,
                      PW = fwrites F PROMPT || fflush F || freads F;

/* prep F: ignore SIGINT and SIGTSTP and prepare the terminal */

prep F          = tcsetattr (fileno F) TCSAFLUSH NATTR || (ATTR,TRAPS)
                    where TRAPS = map (trap SIG_IGN) [SIGINT,SIGTSTP],
                      ATTR = tcgetattr (fileno F),
                      (IF,OF,CF,LF,IS,OS,CC) = ATTR,
                      LF = LF and not (ECHO or ECHOE or ECHOK or ECHONL),
                      NATTR = (IF,OF,CF,LF,IS,OS,CC);

/* unprep F SAVE: revert to previous settings */

unprep F (ATTR,TRAPS)
                = tcsetattr (fileno F) TCSAFLUSH ATTR ||
                  zipwith trap TRAPS [SIGINT,SIGTSTP];
```

12.10 System Information

Error codes from various system operations can be retrieved with the following functions:

```
public extern errno, seterrno N;        // get/set last error code
public extern perror S, strerror N;     // print error message
```

The perror function is commonly used to report error conditions in system operations on the standard error file. For instance:

```
==> fopen "/etc/passw" "r"
fopen "/etc/passw" "r"

==> perror "fopen"
fopen: No such file or directory
()
```

If more elaborate formatting is required then you can use strerror on the errno value to obtain the error message as a string:

```
==> fprintf ERROR "fopen returned message ‘%s’\n" (strerror errno)
fopen returned message ‘No such file or directory’
()
```

Note that errno is only set when an error occurs in a system call. You can use seterrno to reset the errno value before a system operation to check whether there actually was an error while executing the system call:

```
==> seterrno 0
()

==> fopen "/etc/passwd" "r"
<<File>>

==> perror "fopen"
fopen: Success
()
```

The remaining operations are used to obtain various information about the system and its information databases. For instance, the uname operation returns a 5-tuple containing information identifying the operating system (this operation is generally only available on UNIX-like systems):

```
public extern uname;

/* Access components of uname result. */

public un_sysname UNAME, un_nodename UNAME, un_release UNAME,
  un_version UNAME, un_machine UNAME;
```

The hostname of the system can be retrieved with the gethostname function.

```
public extern gethostname;
```

The password and group database can be accessed with the following functions. Password entries are encoded as 7-tuples (NAME, PASSWD, UID, GID, GECOS, DIR, SHELL), group entries as 4-tuples (NAME, PASSWD, GID, MEMBERS). This information is only available on UNIX-like systems.

```
public extern getpwuid UID, getpwnam NAME;      // look up a password entry
public extern getpwent;                         // list of all pw entries
```

```
public extern getgrgid GID, getgrnam NAME;      // look up group entry
public extern getgrent;                         // list of all group entries

/* Access components of password and group structures. */

public pw_name PW, pw_passwd PW, pw_uid PW, pw_gid PW, pw_gecos PW,
  pw_dir PW, pw_shell PW;

public gr_name GR, gr_passwd GR, gr_gid GR, gr_members GR;
```

Moreover, the `crypt` function can be used to perform UNIX password encryption (see `crypt`(3) for details).

```
public extern crypt KEY SALT; // (U)
```

The following functions can be used to query host information as well as the network protocols and services available on your system. This information is closely related to the socket interface described in Section 12.11 [Sockets], page 143. For a closer description of these operations we refer the reader to the corresponding manual pages. Note that the `gethostent`, `getprotoent` and `getservent` operations are not available on Windows.

The host database: Host entries are of the form `(NAME, ALIASES, ADDR_TYPE, ADDR_LIST)`, where `NAME` denotes the official hostname, `ALIASES` its alternative names, `ADDR_TYPE` the address family and `ADDR_LIST` the list of addresses.

```
public extern gethostbyname HOST, gethostbyaddr ADDR;
public extern gethostent; // (U)

public h_name HENT, h_aliases HENT, h_addr_type HENT, h_addr_list HENT;
```

Note that both hostnames and IP addresses are specified as strings. Hostnames are symbolic names such as `"localhost"`, and can also have a domain name specified, as in `"www.gnu.org"`. IPv4 addresses use the well-known "numbers-and-dots" notation, like the loopback address `"127.0.0.1"`. IPv6 addresses are usually written as eight 16-bit hexadecimal numbers that are separated by colons; two colons are used to abbreviate strings of consecutive zeros. For example, the IPv6 loopback address `"0:0:0:0:0:0:0:1"` can be abbreviated as `"::1"`.

The protocol database: Protocol entries are of the form `(NAME, ALIASES, PROTO)` denoting official name, aliases and number of the protocol.

```
public extern getprotobyname NAME;
public extern getprotobynumber PROTO;
public extern getprotoent; // (U)

public p_name PENT, p_aliases PENT, p_proto PENT;
```

The service database: Service entries are of the form `(NAME, ALIASES, PORT, PROTO)` denoting official name, aliases, port number and protocol number of the port. The `NAME` argument of `getservbyname` can also be a pair `(NAME, PROTO)` to restrict the search to services for the given protocol (given by its name). Likewise, the `PORT` argument can also be given as `(PORT, PROTO)`.

```
public extern getservbyname NAME;
public extern getservbyport PORT;
public extern getservent; // (U)

public s_name PENT, s_aliases PENT, s_port PENT, s_proto PENT;
```

For instance, here is some information retrieved from a typical Linux system:

```
==> uname
("Linux","obelix","2.4.19-4GB","#2 Tue Mar 4 16:03:51 CET 2003","i686")

==> gethostname
"obelix"

==> gethostbyname gethostname
("obelix.local",["obelix"],2,["127.0.0.2"])

==> gethostbyaddr "::1"
("localhost",["ipv6-localhost","ipv6-loopback"],10,["::1"])

==> getprotobyname "tcp"
("tcp",["TCP"],6)

==> getservbyname "ftp"
("ftp",[],5376,"tcp")

==> getpwuid getuid
("ag","x",500,100,"Albert Graef","/home/ag","/bin/bash")

==> getgrgid getgid
("users","x",100,[])
```

12.11 Sockets

The following functions are available on systems providing a BSD-compatible socket layer, which, besides BSD, includes BEOS, Linux, OSX, Windows and most recent System V flavours. Sockets provide bidirectional communication channels on the local machine as well as across the network. Sockets are represented by file descriptors which can be written to and read from with the `send` and `recv` functions. On most systems, socket descriptors are just ordinary file descriptors which can also be used with low-level I/O functions and `fdopen` as usual. However, on some systems (in particular, BEOS and Windows), socket descriptors are "special" and all socket I/O must be performed with the special socket operations.

At creation time, a socket is described by the following attributes (see `socket`(2) for more details):

- Address family (sometimes also referred to as namespace, protocol family or domain). The adress families supported by this implementation are `AF_LOCAL` (a.k.a. `AF_UNIX` a.k.a. AF_FILE), AF_INET and AF_INET6. Not all protocol families may be available on all systems (e.g., the Windows socket library only supports AF_INET).

- Communication style (also called the socket type). One distinguishes between "connection-based" sockets which are used to establish client/server connections, and "connectionless" sockets which allow data to be received from and sent to arbitrary addresses. The socket types supported by this implementation are SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET, SOCK_RAW and SOCK_RDM. Among these, SOCK_STREAM, SOCK_SEQPACKET and SOCK_RDM are connection-based. Please note that not all socket types are supported for all protocol families, and some socket types may be entirely missing on non-UNIX systems.
- Protocol: For each address a.k.a. protocol family and socket type there may be a number of different protocols available. For the AF_LOCAL namespace, which refers to the local filesystem, the protocol is always 0, the default protocol. The available protocols for the internet namespaces can be retrieved from the protocol database (see Section 12.10 [System Information], page 140).

Before another process can connect to a socket it must also be bound to an address. The address format depends on the address family of the socket. For the local namespace, the address is simply a filename on the local filesystem. For the IPv4 namespace, it is a pair (HOST, PORT) where HOST denotes the host name or IP address, specified as a string, and PORT is a port number. For the IPv6 namespace, it is a quadrupel (HOST, PORT, FLOWINFO, SCOPEID). Host names and known port numbers can be retrieved from the host and service databases (see Section 12.10 [System Information], page 140).

The following operations are provided to create a socket, or a pair of connected sockets, for the given address family, socket type and protocol. They return the file descriptor of the socket (resp. a pair of file descriptors).

```
public extern socket FAMILY TYPE PROTO;
public extern socketpair FAMILY TYPE PROTO; // (U)
```

The shutdown function terminates data transmission on a socket. You can stop reading, writing or both, depending on whether HOW is SHUT_RD, SHUT_WR or SHUT_RDWR. Note that this operation does not close the socket's file descriptor; for this purpose closesocket is used (see below).

```
public extern shutdown SOCKET HOW;
```

The closesocket function closes a socket. On most systems this is just identical to close (see Section 12.8 [Low-Level I/O], page 134), but, as already noted, on some systems socket descriptors are special and you must use this function instead.

```
public extern closesocket SOCKET;
```

The bind function binds a socket to an address. This is also done automatically when the socket is first used. However, if the socket has to be found by another process you'll have to explicitly specify an address for it. The bind function does just that.

```
public extern bind SOCKET ADDR;
```

The following operations are used to start listening for and accept connection requests on a socket. These operations are used on the server side of a connection-based socket.

The argument N of listen denotes the maximum number of pending connection requests for the server. After the call to listen, the server can accept connections from a client with the accept function, which returns a pair (SOCKET, ADDR), where SOCKET is a new socket connected to the client, and ADDR is the client's address.

```
public extern listen SOCKET N;
public extern accept SOCKET;
```

The connect function is used to initiate a connection on a socket. This function can be used on both connection-based and connectionless sockets. In the former case, connect can only be invoked once. In the latter case, it can be invoked multiple times, and sets the remote socket for subsequent send and receive operations.

```
public extern connect SOCKET ADDR;
```

The following routines retrieve information about a socket. The local address of a socket and the address of the remote socket it is connected to can be retrieved with getsockname and getpeername. Socket options, specified using a protocol level LEVEL and an option index OPT, can be queried and changed with getsockopt and setsockopt. The option values are encoded as byte strings (cf. Section 12.3 [Byte Strings], page 120). For a description of the available options see getsockopt(2).

```
public extern getsockname SOCKET;
public extern getpeername SOCKET;
public extern getsockopt SOCKET LEVEL OPT;
public extern setsockopt SOCKET LEVEL OPT VAL;
```

Finally, the following specialized I/O functions are used to transmit data over a socket. All data is encoded as byte strings. The receive operations return the received data (which may be shorter than the requested size, if not enough data was currently available), the send operations the number of bytes actually written. For recvfrom/sendto the data is encoded as a pair (ADDR, DATA) which includes the source/destination address; these operations are typically used for connectionless sockets. The FLAGS argument is used to specify special transmission options (see the MSG_* constants at the beginning of clib.q).

```
public extern recv SOCKET FLAGS SIZE, send SOCKET FLAGS DATA;
public extern recvfrom SOCKET FLAGS SIZE, sendto SOCKET FLAGS DATA;
```

Example

The following script demonstrates how we can implement a connectionless server in the IPv4 namespace which repeatedly accepts a request from a client and sends back an answer. In this example the requests are strings denoting Q expressions; the server evaluates each expression and sends back the result as a string. The client reads input from the user, transmits it to the server and prints the received answer. Note that the transmitted strings are represented as byte strings, as required by the recvfrom and sendto operations. The bytestr and bstr functions are used to convert between character and byte strings, see Section 12.3 [Byte Strings], page 120.

```
def BUFSZ = 500000; // buffer size

/* the server: receive messages, evaluate them as Q expressions, and
```

```
   send back the results */

def SERVER = ("localhost",5001); // the server address

server          = server_loop FD
                    where FD:Int = socket AF_INET SOCK_DGRAM 0,
                      () = bind FD SERVER;
                = perror "server" otherwise;

server_loop FD  = sendto FD 0 (ADDR,eval MSG) || server_loop FD
                    where (ADDR,MSG) = recvfrom FD 0 BUFSZ;
                = server_loop FD otherwise;

/* evaluate an expression encoded as a byte string, catch syntax
   errors and exceptions, convert result back to a byte string */

eval MSG        = catch exception (bytestr (str VAL))
                    where 'VAL = valq (bstr MSG);
                = bytestr ">>> SYNTAX ERROR" otherwise;
exception _     = bytestr ">>> ABORTED";

/* the client: read input from user, send it to the server, print
   returned result */

def CLIENT = ("localhost",5002); // the client address

client          = client_loop FD
                    where FD:Int = socket AF_INET SOCK_DGRAM 0,
                      () = bind FD CLIENT;
                = perror "client" otherwise;

client_loop FD  = sendto FD 0 (SERVER,bytestr MSG) ||
                  printf "%s\n" (bstr (recv FD 0 BUFSZ)) || client_loop FD
                    if not null MSG
                    where MSG:String = writes "\nclient> " || flush || reads;
                = () otherwise;
```

For instance, we can invoke the server in a secondary thread and then execute the client as follows:

```
==> def S = thread server

==> client

client> prd (nums 1 50)
30414093201713378043612608166064768844377641568960512000000000000

client> 1+)
>>> SYNTAX ERROR
```

```
client> quit
>>> ABORTED

client>
```

12.12 POSIX Threads

On systems where the POSIX threads library or some compatible replacement is available (this includes Windows and most modern UNIXes), `clib` provides functions for handling multiple threads of control. *Threads*, a.k.a. “light-weighted processes”, allow you to realize “multithreaded scripts” consisting of different tasks which together perform some computation in a distributed manner. All tasks are executed concurrently. Thus you can, e.g., perform some lengthy calculation in a background task while you go on evaluating other expressions in the interpreter’s command loop. You can also have tasks communicate via *mutexes*, *conditions* and *semaphores*.

The operations described in this section are in close correspondence with POSIX 1003.1b. However, some operations are named differently, and semaphores provide the extra functionality of sending data from one thread to another. Mutexes are also supported, mostly for the purpose of handling critical sections involving operations with side-effects (I/O etc.). Mutexes are *not* required to make conditions work since these have their own internal mutex handling. For more information on POSIX threads, please refer to the corresponding section in the UNIX manual.

Please note that these functions will only work as advertised if the interpreter has been built with POSIX thread support. We also remark that in the current implementation the interpreter effectively serializes multithreaded scripts on the reduction level and thus user-level threads cannot really take advantage of multi-processor machines.

12.12.1 Thread Creation and Management

`Clib` threads are represented using *handles* (objects of type `Thread`). Note that a thread is canceled automatically as soon as its handle is garbage collected, thus you should keep the handle around as long as the thread is needed. For convenience, thread handles are numbered arbitrarily, starting at 0 which denotes the main thread, and are ordered by the thread numbers. This is handy, e.g., if you want to use thread handles as indices in a dictionary.

```
public extern type Thread;             // thread handle type

public isthread THREAD;                // check for thread objects

public extern thread_no THREAD;        // thread number
public extern this_thread;             // handle of the current thread
```

The basic thread operations are listed below. The `thread` function starts evaluating its special argument in a new thread, and returns its handle. You can wait for a thread to terminate and obtain the evaluated result with the `result` function. (If there is no result, because the thread has been canceled, or was aborted with `halt`, `quit` or a runtime error, `result` fails.) Note that `halt` or `quit` in a thread which is not the main thread only terminates the current thread; however, the `exit` function, cf. Section 12.7 [Process Control], page 130, always exits from the interpreter. You can also terminate the current thread immediately and return a given value as its result with the `return` function; in the main thread, this function is equivalent to `halt` and the return value is ignored. Moreover, all threads except the main thread can also be canceled from any other thread using the `cancel` function. Finally, the `yield` function allows the interpreter to switch threads at any given point (normally the interpreter will only switch contexts in certain builtins and when a new rule is activated).

```
public extern special thread X;         // start new thread
public extern return X;                 // terminate thread with result X
public extern cancel THREAD;            // cancel THREAD
public extern result THREAD;            // wait for THREAD, return result
public extern yield;                    // allow context switch
```

`Clib` threads always use *deferred* cancellation, hence thread cancellation requests are usually not honored immediately, but are deferred until the thread reaches a *cancellation point* where it is safe to do so. Cancellation points occur at certain C library calls listed in the POSIX threads documentation, when a new equation is activated in the Q interpreter, and when `yield` is called.

You can also check whether a thread is still active or has been canceled. If neither condition holds, then the thread has already been terminated and you can obtain its result with the `result` operation.

```
public extern active THREAD;            // check if THREAD is active
public extern canceled THREAD;          // check if THREAD was canceled
```

12.12.2 Realtime Scheduling

Threads are always created with the default "non-realtime" scheduling policy. On some systems it is also possible to increase a thread's priority and have it scheduled in "realtime". This is useful for tasks with strict timing and responsiveness requirements, such as a function performing recording or playback in a multimedia application.

The scheduling policy and the priority of a thread can be changed with the following function which takes two extra arguments, the policy `POL` (0 = default, 1 = realtime round-robin, 2 = realtime fifo scheduling) and the priority `PRIO` (where 0 denotes the default priority). Note that on most systems realtime scheduling will only be granted to privileged processes.

```
public extern setsched THREAD POL PRIO; // set scheduling parameters
```

The current scheduling policy and priority of a thread can be retrieved with the `getsched` function:

```
public extern getsched THREAD;          // get scheduling parameters
```

The actual range of priority values is system-specific. On a typical UNIX system a non-realtime thread can only have priority 0, while realtime threads always have positive priorities. Higher priority threads always take precedence over lower priority ones. All else being equal, threads using round-robin scheduling are each given their "fair" timeslice, while fifo threads are handled on a "first come first served" basis. The latter can only be interrupted by higher priority processes (and signals) and thus should be used with utmost care. In any case you must make sure that a high-priority thread does not run unsuspended for extended periods of time, otherwise it might lock up your system. Thus a realtime thread should typically spend most of its time "in limbo" where it waits for input to arrive or a condition to be signaled.

Assuming that the priority value ranges are contiguous and contain either 0 or 1, you can determine the ranges for your system with the following little script:

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

priotest POL    = (hd L,last L) if not null L
                    where L = reverse (while (test POL) pred 0) ++
                      while (test POL) succ 1;
```

Here is a sample result obtained on Linux:

```
==> map priotest [0,1,2]
[(0,0),(1,99),(1,99)]
```

Realtime Scheduling under Windows

Under Windows, things are a bit different, since Windows does not really have POSIX-compatible thread scheduling. Instead, processes have "priority classes" while threads have "priorities" which are interpreted in the context of the priority class of the process they belong to. Therefore under Windows the `POL` argument of the `setsched` operation is actually interpreted as a priority class for the *entire* process while the `PRIO` argument specifies the priority of the individual thread. `Clib` keeps track of all `setsched` calls and always lets the process have the highest priority class specified for a thread which is still active.

To these ends, the policy values 0, 1 and 2 are mapped to the priority classes "normal", "high" and "realtime". (The latter should be used sparingly, if at all, in a Q script, because it makes Windows very unresponsive.) Moreover, the `setsched` function also accepts a policy value of -1 to denote "idle time" processes. (You'll rarely have any use for this, unless you want to write a screensaver in Q.) For each policy, the possible priority values have a range from -3 to 3, where -3 denotes idle time threads, 0 is the normal priority, and 3 is the highest priority to be used for "time-critical" threads. The remaining priority levels provide some additional amount of control over which thread gets the bone first.

While some "special effects" can be achieved with Windows' more exotic priority values, for typical usage a policy of 0 with zero priority should be employed for ordinary threads, a policy of 1 with some (small) positive priority value for a thread with moderate realtime requirements, and a policy of 2 if time is very critical. If you follow these conventions then your script will be able to run under Windows and most UNIX systems unchanged.

12.12.3 Mutexes

`Clib` mutexes come in three flavours: *fast* (no error checking), *error checking* (fail if the current thread already holds a lock on the mutex) and *recursive* (the same thread may lock the mutex multiple times, and the same number of unlock operations is required to unlock the mutex again). The supported operations are `lock` (wait for the mutex to be unlocked, then lock it and return `()`), `unlock` (unlock the mutex, return `()`) and `try` (lock the mutex if it is available, fail otherwise).

CAVEAT: These operations are *dangerous*, as you can easily create deadlocks which might lock up the interpreter as well. Note that a deadlock will also prevent the involved threads from being canceled since, in accordance with the POSIX standard, the wait for a mutex lock is *not* a cancellation point. Thus these operations should be used with care.

Since Q has no mutable variables and the interpreter's builtins are all thread-safe, mutexes are actually used much less in Q than in other, procedural languages. They are most useful for protecting critical sections in which a sequence of operations with side-effects (such as I/O) is to be carried out in an atomic fashion. This can be done by locking a mutex at the beginning and unlocking it at the end of the sequence. Such usage is safe, provided that no operation in the sequence may block for an extended or even indefinite period of time.

```
public extern type Mutex;               // mutex type

public ismutex MUTEX;                   // check for mutex objects

public extern mutex;                    // standard (fast) mutex object
public extern errorchecking_mutex;      // error checking mutex object
public extern recursive_mutex;          // recursive mutex object

public extern lock MUTEX;               // lock MUTEX
public extern unlock MUTEX;             // unlock MUTEX
public extern try MUTEX;                // try MUTEX
```

12.12.4 Conditions

`Clib` conditions support the following operations: `signal` (wake up one thread waiting for the condition), `broadcast` (wake up all threads waiting for the condition) and `await` (suspend the current thread until the condition is sent). Each of these operations returns `()`. The `await` operation can also be invoked with a `(COND, TIME)` tuple to denote a timed wait. If the wait times out or is interrupted by a signal then the operation fails. Note that in accordance with the POSIX standard the `TIME` value denotes an *absolute* time (integer or floating point value in seconds since the "epoch", see also the description of the builtin `time` function in Section 10.7 [Miscellaneous Functions], page 90).

```
public extern type Cond;                // condition type

public iscond COND;                     // check for condition objects

public extern cond;                     // new condition object
```

```
public extern signal COND;              // signal COND
public extern broadcast COND;           // broadcast COND
public extern await COND;               // wait for COND, or (COND,TIME)
```

12.12.5 Semaphores

`Clib` semaphores are in fact *semaphore queues* which can be used as a a communication channel to pass values between different threads using a FIFO discipline. In extension to the POSIX 1003.1b semaphore operations, `clib` also provides support for *bounded* semaphores which are created with a positive limit on the number of values the semaphore queue may hold at any time. When a new value is posted to a bounded semaphore, the current thread is suspended until the queue has room to receive a new value, if necessary.

The supported operations are `post` (enqueue a value), `get` (dequeue a value and return it, as soon as one is available), `try` (non-blocking version of `get` which fails if no value is currently available), `get_size` or `#` which both return the current queue size, and `get_bound` which returns the queue size limit of a bounded semaphore (or zero if the semaphore is unbounded).

Note that even with unbounded semaphores the maximum semaphore size is actually limited by the operating system, see your local POSIX threads documentation for details. If this implementation-specific limit is exceeded, the attempt to post a new value raises a `syserr 9` exception.

```
public extern type Sem;                 // semaphore type

public issem SEM;                       // check for semaphore objects

public extern sem;                      // semaphore object
public extern bounded_sem MAX;          // bounded semaphore object

public extern post SEM X;               // enqueue a value
public extern get SEM;                  // dequeue a value
public extern try SEM;                  // dequeue a value, fail if none

public extern get_size SEM;             // get the current queue size
public extern get_bound SEM;            // get the max queue size (0 if none)
```

12.12.6 Threads and Signals

Some remarks about the interaction between `clib` threads and the interpreter's signal handling are in order. It is a well-known fact that threads and signals generally do not mix very well. Therefore, in the current implementation, all signal handling is actually performed in the interpreter's main thread. This is where you should set up your signal handlers using the `trap` and `catch` builtins, as explained in Section 10.6 [Exception Handling], page 86. If necessary, the main thread of your script can inform other threads about received signals using the thread synchronization functions described above.

However, the interpreter still allows you to set up traps in secondary threads and keeps track of the configured traps separately for each thread. The traps in secondary threads will not have any effect until such a thread forks (see Section 12.7 [Process Control], page 130). In this case the forking thread will be the one and only thread in the child process and becomes the main thread. At this point, it takes on the signal handling, and any traps which have been set up before become active. (Note that in order to make this work reliably, you should configure the traps *before* you call `fork` and protect the call to `fork` with an enclosing `catch`. Otherwise a signal might arrive in the time between the `fork` and the next function call, causing the new process to exit before it had a chance of setting up its own signal handling.)

12.12.7 Thread Examples

As already mentioned, threads allow you to perform a lengthy calculation as a background task. You can start such a task simply as follows:

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

==> // do some other work ...

==> result TASK // get the result
500000500000

==> stats all
thread #0: 0 secs, 1 reduction, 0 cells
thread #1: 8.42 secs, 2000003 reductions, 2000007 cells
```

As indicated, if the `stats` command is invoked with the parameter `all` then it also shows the statistics of background threads once they are finished. To release all resources associated with a thread (and also remove it from the `stats all` list) you must undefine all variables referencing the thread handle:

```
==> undef TASK
```

Conditions are useful when a thread has to wait for some condition before proceeding with a computation. E.g.:

```
==> def COND = cond, TASK = thread (await COND || writes "Got it!\n")

==> signal COND
()
Got it!
```

Semaphores are used when expression values have to be passed from one thread to another. For instance, let us rewrite the backtracking algorithm for the N queens problem (see Section 10.6 [Exception Handling], page 86) such that it runs as a background task which returns results via a semaphore instead of printing them directly on the terminal:

```
def RES = sem;  // semaphore used to transmit results

queens N        = thread (search N 1 1 [] || post RES ());
```

```
search N I J P  = post RES P if I>N;
                = search N (I+1) 1 (P++[(I,J)]) || fail if safe (I,J) P;
                = search N I (J+1) P if J<N;
                = () otherwise;
```

Note that we added a `post RES ()` to signal that all solutions have been found. The `safe` function is as in Section 10.6 [Exception Handling], page 86. We also remark that in order to limit the size of the `RES` semaphore queue, we could use a bounded semaphore instead, e.g.:

```
def RES = bounded_sem 10;
```

We can use a second thread to process the semaphore queue and print solutions as they become available:

```
print           = thread (loop print1 RES);

print1 ()       = return (); // no more results
print1 P        = write P || writes "\n" otherwise;

/* iterate a function over a semaphore */

loop F SEM      = F (get SEM) || loop F SEM;
```

To use this program, simply start the two background tasks as follows:

```
==> def QUEENS = queens 8, PRINT = print
```

Both threads will begin to execute immediately, and you will see the results scroll by. Once the threads are finished, you can check the resources used by each thread with the `stats all` command:

```
==> stats all
thread #0: 0 secs, 2 reductions, 2 cells
thread #1: 1.71 secs, 700451 reductions, 943 cells
thread #2: 0 secs, 461 reductions, 3 cells
```

Also note that here we used again a `def` statement to assign the thread handles to corresponding variables. It is important that you keep these variables as long as you want the threads to survive. The interpreter automatically cancels a thread as soon as the corresponding thread handle is garbage collected. Hence you can stop the threads at any time by simply undefining the variables using, e.g., the `clear` command.

12.13 Expression References

Expression references work like pointers to expression values. The reference is initialized with the `ref` function, giving the initial value as an argument. The referenced value can then be changed with the `put` function and retrieved with `get`.

```
public extern type Ref;                 // reference type

public isref REF;                       // check for reference objects
```

```
public extern ref X;                    // initialize a reference object

public extern put REF X;                // store a new value
public extern get REF;                  // retrieve the current value
```

NOTE: References can point to arbitrary values, in particular they can also point to other references. Cyclic chains of references are also supported, but should be avoided since in the current implementation the interpreter cannot garbage-collect such structures.

References can be used to implement mutable data structures. This is particularly useful if a function has to maintain some internal state to perform its calculations. For instance, here is a function which memorizes past function calls in a hash table so that each requested value only has to be computed once:

```
def HASH = ref emptyhdict;

public hashed F X;
hashed F X      = get HASH!(F,X) if member (get HASH) (F,X);
                = put HASH (update (get HASH) (F,X) Y) || Y
                    where Y = F X otherwise;
```

Using the `hashed` function, a hashed version of the recursive Fibonacci function can be implemented as follows. Since no value of the function is computed more than once, the definition works in linear time (up to logarithmic factors for the table updates and lookups). In fact, even an expression like `map fib (nums 0 N)` will only require a linear number of table manipulations and other operations.

```
fib             = hashed fib2;

fib2 0          = 0;
fib2 1          = 1;
fib2 N          = fib (N-1)+fib (N-2) if N>1;
```

References also provide a general means for communicating values in a multithreaded script (see Section 12.12 [POSIX Threads], page 147). In this case they are typically used together with a condition which is signaled when the value changes:

```
def VAL_CHANGED = cond, VAL = ref ();
change X        = put VAL X || broadcast VAL_CHANGED;
```

A loop like the following might then be executed by any number of other threads which have to keep track of the value in question, and perform some appropriate action when the value changes:

```
watch           = await VAL_CHANGED || action (get VAL) || watch;
```

12.14 Time Functions

The functions described in this section are used to return information about the active timezone and the current time, and convert time values to various formats. Also available are functions to measure cpu time. These functions are in close correspondence with the date and time functions of the C library.

The calendar date and time functions use two different representations for time values:

- Simple time (denoted `T` in the sequel) is represented as the number of seconds elapsed since the "epoch", 00:00:00 on January 1, 1970, UTC.
- Broken-down time (denoted `TM`) is encoded as a 9-tuple `(YEAR, MONTH, DAY, HOUR, MIN, SEC, WDAY, YDAY, ISDST)`. See the description of the `tm` struct in the `ctime`(3) manual page for more information.

Three functions (`tzname`, `timezone`, `daylight`) are provided to return information about the current timezone and daylight savings settings. The `gmtime` and `localtime` functions convert simple time to broken-down time using UTC or the local timezone, respectively. The functions `mktime` and `asctime` are used to convert a broken-down time value into a simple time value or the standardized string representation used by the `date`(1) program, respectively. The `ctime` function combines `localtime` and `asctime`. For more flexible formatting of time values, the `strftime` function can be used, which takes as its first argument a format string as described on the `strftime`(3) manual page.

```
public extern tzname, timezone, daylight;
public extern ctime T, gmtime T, localtime T;
public extern mktime TM, asctime TM, strftime FORMAT TM;
```

The components of broken-down time. can be accessed with the following functions:

```
public tm_year TM, tm_month TM, tm_day TM, tm_hour TM, tm_min TM,
  tm_sec TM, tm_wday TM, tm_yday TM, tm_isdst TM;
```

On the current 32 bit systems time values must be representable as 4 byte integer values, which means that the available range usually goes from `"Fri Dec 13 20:45:52 1901"` to `"Tue Jan 19 03:14:07 2038"`. (This will probably be fixed before "time ends" on current UNIX systems in January 2038, though.)

Some examples:

```
==> tzname; timezone; daylight
("CET","CEST")
-3600
1

==> ctime time
"Mon Mar 17 04:08:29 2003\n"

==> localtime time
(103,2,17,4,8,41,1,75,0)

==> strftime "Hey it's %c" _
"Hey it's Mon Mar 17 04:08:41 2003"

==> ctime (st_mtime (stat "README-Clib"))
"Wed May  1 16:59:00 2002\n"
```

Two additional functions are provided for measuring cpu time. Note that `clock` and `times` measure cpu time in different units, given by the constants `CLOCKS_PER_SEC` and `CLK_TICK`, respectively. The `times` function returns a 5-tuple `(TOTAL, UTIME, STIME, CHILD_UTIME, CHILD_STIME)`. See `times`(2) for details.

```
public extern clock, times;
```

For instance, you can calculate the cpu time in seconds it takes to evaluate an expression by computing the difference between the `clock` time at the end and at the beginning, divided by `CLOCKS_PER_SEC`:

```
==> -(clock - (prd (nums 1 10000) || clock)) / CLOCKS_PER_SEC
0.14
```

Note that the initial value and the resolution of the `clock` timer is system-dependent. Usually the timer is initialized at process creation time, but that is not guaranteed. Moreover, the value of `clock` typically is a machine integer which wraps around after a finite time interval; on 32 bit systems with `CLOCKS_PER_SEC = 1000000`, as recommended by the POSIX standard, this happens approximately every 72 minutes.

12.15 Filename Globbing

Matching filenames against patterns with shell wildcards (`*`, `?` etc.), also known as filename "globbing", is implemented by the following functions:

```
public extern fnmatch PATTERN S; // check whether S matches PATTERN
public extern glob PATTERN; // list of all filenames matching PATTERN
```

Examples

```
==> map (fnmatch "*.q") ["clib.q","clib.c"]
[true,false]

==> map (fnmatch "*.[cq]") ["clib.q","clib.c"]
[true,true]

==> glob "*.[qc]"
["clib.c","clib.q","factor.q","globexamp.q","regexamp.q"]

==> glob "/h*"
["/home"]
```

Only a single pattern is accepted by both `fnmatch` and `glob`. However, it is a simple matter to provide your own rules which extend the `clib` operations to lists of patterns:

```
fnmatch [] S:String      = false;
fnmatch [P:String|Ps] S:String
                         = fnmatch P S or else fnmatch Ps S;

glob Ps:List             = rmdups (sort (<) (cat (map glob Ps)));
```

```
rmdups []              = [];
rmdups [X,X|Xs]        = rmdups [X|Xs];
rmdups [X|Xs]          = [X|rmdups Xs] otherwise;
```

Note that in our extension of the definition of `glob` we employ the `sort` function (see Section 12.18 [C Replacements for Common Standard Library Functions], page 168) to sort the resulting list, and then remove adjacent duplicates (which could occur because a file may match more than one pattern).

Now let's see these definitions in action:

```
==> map (fnmatch ["*.q","*.c"]) ["clib.q","clib.c"]
[true,true]

==> glob ["*.q","*.c","clib.*"]
["clib.c","clib.q","clib.so","factor.q","globexamp.q","regexamp.q"]
```

12.16 Regular Expression Matching

The `clib` module implements regular expression matching using the POSIX `regcomp` and `regexec` functions provided by the GNU regex library. Regular expressions are generally specified using the *extended* (`egrep`-like) syntax. `Clib` divides the matching functions into two interfaces, the *high-level* interface (`regex` function) and the *low-level* interface (`regmatch` and friends). Both interfaces are used with a third group of *match state* functions (`reg` and friends) which provide access to information about the current match.

For most purposes, the high-level `regex` function should be all that is needed. However, the low-level functions are provided so that you can create your own specialized regular expression engines, should you ever need to do so. In this case you might wish to take a look at the definition of the `regex` function in `clib.q` and modify it to suit your needs.

Note that in accordance with POSIX matching semantics, all matching functions search for an occurrence of the pattern in the target string, rather than simply checking whether the whole string matches the given pattern. If the latter functionality is required, you can use `^` and `$` to tie the match to the beginning and the end of the string.

12.16.1 High-Level Interface

The following functions are implemented as special forms, with the `EXPR` argument being passed unevaluated.

```
special regex ~OPTS ~REGEX ~S EXPR, regex_next EXPR;
```

The `regex` function evaluates, for each match of the given regular expression (also called the *pattern*) in the given string, the special `EXPR` argument, and returns the collection of all results as a list, in the order in which the matches were found. If no match is found, the result list will be empty. The `EXPR` argument typically uses the match state functions (see Section 12.16.3 [Match State Information], page 159) to retrieve the current match information. You can also invoke the `regdone` function (see Section 12.16.2 [Low-Level Interface], page 158) to escape from the search at any time.

The `regex_next` function is used internally by `regex` to iterate over all subsequent matches; normally it is not invoked directly by the user.

Matching generally proceeds from left to right, and if several different substrings match at a given position, the longest match is preferred. The matching process is controlled by means of the `OPTS` string argument, which may contain zero or more of the following option characters (the corresponding flags of the POSIX `regcomp/regexec` functions are given in parentheses, where applicable):

`g` Match globally, i.e., find all non-overlapping occurrences. Otherwise only the first match is reported (if any).

`G` Like `g`, but report overlapping matches as well.

`i` Do case-insensitive matches (`REG_ICASE`).

`n` Do multi-line matches. This makes `^` and `$` match line beginnings and ends (besides beginning and end of the string), and makes `.` and lists `[...]` *not* match the newline character, unless it is explicitly included in a list (`REG_NEWLINE`).

`^` Do *not* match `^` at the beginning of the string (`REG_NOTBOL`).

`$` Do *not* match `$` at the end of the string (`REG_NOTEOL`).

Abnormal error conditions such as bad regular expression syntax are handled by returning an expression of the form `regerr MSG` where `MSG` is a string describing the error. You may give an appropriate definition of `regerr`, or check for literal `regerr MSG` values, to handle such error conditions in any way you like.

The `regex` function is `clib`'s primary interface to perform regular expression matching, and should cover most usual applications. You can find a number of examples showing how to use this function below (see Section 12.16.4 [Basic Examples], page 160). The `regex` function itself is implemented in Q using the low-level functions discussed in the following section. You might wish to take a look at these if you need to do something special which cannot be done with the `regex` function (or cannot be done in an efficient manner).

12.16.2 Low-Level Interface

The following functions are written in C using the GNU regex library.

```
public extern regmatch OPTS REGEX S, regnext, regdone;
```

The `regmatch` function searches for the first (i.e., leftmost) occurrence of the regular expression pattern in the given string, and the parameterless `regnext` function searches for the next occurrence. Both functions normally return a truth value indicating success or failure (but see the remarks concerning error handling below). In case of success, you can retrieve the current match using the match state functions (see Section 12.16.3 [Match State Information], page 159).

IMPORTANT: These functions have side-effects; they change the internal match state which is accessed using the match state functions (see Section 12.16.3 [Match State Information], page 159). This hidden state is provided for your convenience, to spare you the trouble of having to pass explicit parameters between the matching operations and the match state functions. All match state information is maintained on an internal stack, so that you can start a new search with the `regmatch` function while another one is still in progress. (This also implies that you can recursively invoke `regex` inside the `EXPR` argument of a `regex` call.)

The `OPTS` argument has the same meaning as for the `regex` function, and error handling also works in the same fashion. That is, in case of an abnormal error condition the `regmatch/regnext` functions return a `regerr MSG` expression instead of a truth value. See Section 12.16.1 [High-Level Interface], page 157.

Note that the `regnext` function will always return `false`, unless one of the `g/G` (global search) options is specified in the preceding `regmatch` call.

To terminate a (global) search which is still in progress (i.e., has not failed yet), you can call the `regdone` function, which always returns `()`. Subsequent invokations of the match state functions will then behave just like after a failing `regmatch/regnext` call. (After a failed match, `regdone` is not needed; it will be invoked automatically.)

Some care is needed to ensure proper operation of the low-level routines with multiple nested searches. As a general rule, each search successfully started with `regmatch` *must* be terminated either with a failing `regnext` call, or by an invokation of the `regdone` function. In both cases, after termination of the nested search, a subsequent `regnext` will continue where `regmatch/regnext` left off when the nested search was started.

It is important to note that these rules also apply if the search is *non-global*. The main rationale behind this behaviour is that, even if one is only interested in the first match, one must still be able to obtain information about any trailing unmatched text using the `regstart/regskip` functions (see Section 12.16.3 [Match State Information], page 159), and this information is only available *after* a match has failed (or was aborted using `regdone`). Thus the interface functions should always behave consistently, no matter whether the `g/G` option was specified or not. (In fact, the only difference `g/G` makes is that, if it is omitted, then `regnext` will pretend that no next match is available even if there is one.)

If this sounds confusing to you, just stick with the high-level `regex` function which takes care of all those messy details. A nested matching example using `regex` is discussed in Section 12.16.9 [Nested Searches], page 165.

12.16.3 Match State Information

These functions are typically invoked after `regmatch`, `regnext`, or in the `EXPR` argument of `regex`, to report information about the current match.

```
public extern regstart, regskip, reg N, regpos N, regend N, regs;
```

The `reg` function returns the text of a match, `regpos` its beginning in the target string (first character position), and `regend` its end (first position *behind* the match). These functions take an integer argument denoting the *group* (a.k.a. parenthesized subexpression) of the regular expression being matched. An argument of `0` always refers to the whole match, `N > 0` to the text matched by the `N`th group (counting opening parentheses from the left). The match reported for a given group will always extend to the right as far as possible ("longest match"), subject to the constraint that enclosing groups will also prefer the longest match, and they always take priority. Thus `reg 0` will extend to the right as far as possible, `reg 1` will be the longest possible match for group 1 inside this match, etc.

If a group belongs to a repetitive pattern (patterns involving the `*`, `+` operators, etc.), then it may be matched more than once. In this case, the text *last* matched by the group is reported by the `reg/regpos/regend` functions.

If the group belongs to an optional part of the pattern (`*`, `?`, different alternatives with `|`), then the group might not be matched at all. For such unmatched groups, `regpos` and `regend` both return `-1` and `reg` returns the empty string. These values are also returned for *all* groups (including `0`) after a match has failed or has been aborted using the `regdone` function.

The `regs` function returns a list with the indices of all matched groups (except `0`).

The parts of the target string which were *not* matched can also be accessed. This is done by means of the `regstart` and `regskip` functions, which return, respectively, the position where the last search for a match started, and the text from this position up to the following match (or to the end of the string if the match failed). These values can also be accessed after a failing or aborted match (e.g., after `regex` has finished), which is useful to determine a trailing unmatched portion in the input string.

12.16.4 Basic Examples

Please refer to your UNIX manual for a description of the regular expression syntax. (To get a start, take a look at the `egrep`(1) manual page; most of what it has to say about (extended) regular expressions should be applicable here too. There are also some good books on the subject.)

So let's consider some elementary examples first. Here's how we can find all occurrences of "identifiers" (defined here as sequences of letters and digits, starting with a letter) in a string:

```
==> regex "g" "[A-Za-z][A-Za-z0-9]*" "1var foo 99 BAR $%&" (reg 0)
["var","foo","BAR"]
```

The first argument to `regex` is always the option string; the `"g"` in our example indicates a *global* search, i.e., a search for all (non-overlapping) matches of the given pattern. The second argument is the regular expression pattern to be matched, and the third argument is the string to be searched for matches against the pattern. The fourth argument is the expression to be evaluated for each match; in our case, `reg 0` is used to simply retrieve the match. Actually, we could provide any appropriate expression here, e.g., we might use `sprintf` to add some fancy formatting:

```
==> regex "g" "[A-Za-z][A-Za-z0-9]*" "1var foo 99 BAR $%&" \
(sprintf "MATCH: %s" (reg 0))
["MATCH: var","MATCH: foo","MATCH: BAR"]
```

What happens if the pattern contains an error, such as an unmatched bracket? Let's see:

```
==> regex "g" "[A-Za-z][A-Za-z0-9*" "1var foo 99 BAR $%&" (reg 0)
regerr "Unmatched [ or [^"
```

Reasonably enough, `regex` returns an error message.

The `regex` function always enumerates matches from left to right, and only reports at most one match for each position in the input string. If more than one substring matches at a given position, the longest match is preferred. Thus in the above example, e.g., `foo` is matched, and not the individual characters `f` and `o` which by themselves also match the given pattern.

Note that with the `g` option only non-overlapping matches are reported; if we want *all* occurrences (even overlapping ones) then we specify the `G` option instead:

```
==> regex "G" "[A-Za-z][A-Za-z0-9]*" "1var foo 99 BAR $%&" (reg 0)
["var","ar","r","foo","oo","o","BAR","AR","R"]
```

And if we are only interested in the first match, we simply omit the `g`/`G` option character:

```
==> regex "" "[A-Za-z][A-Za-z0-9]*" "1var foo 99 BAR $%&" (reg 0)
["var"]
```

So far, so good. After these basics, let's reconsider our original search expression:

```
==> regex "g" "[A-Za-z][A-Za-z0-9]*" "1var foo 99 BAR $%&" (reg 0)
["var","foo","BAR"]
```

We can simplify the pattern somewhat if we use the `i` option to specify a case-independent search:

```
==> regex "gi" "[a-z][a-z0-9]*" "1var foo 99 BAR $%&" (reg 0)
["var","foo","BAR"]
```

Another way to write this pattern uses the predefined *character classes* `[:alpha:]` and `[:alnum:]`; these constructs have the advantage that they are portable, i.e., they work in different character sets and locales.

```
==> regex "g" "[[:alpha:]][[:alnum:]]*" "1var foo 99 BAR $%&" \
(reg 0)
["var","foo","BAR"]
```

So now we know how we can find identifiers in a string. But what if we have to check whether a given string *is* an identifier? For this purpose, we must match the string *as a whole* against the pattern. We can do this by tying our pattern to the beginning and end of the string using the `^` and `$` "anchors". (The `g` option is not required here, since we are only looking for a single match.)

```
==> regex "" "^[[:alpha:]][[:alnum:]]*$" "foo" (reg 0)
["foo"]

==> regex "" "^[[:alpha:]][[:alnum:]]*$" "1var" (reg 0)
[]
```

If we only want a truth value, we can simply test the size of the result list; in this case, any expression argument will do:

```
==> 1=#regex "" "^[[:alpha:]][[:alnum:]]*$" "foo" ()
true

==> 1=#regex "" "^[[:alpha:]][[:alnum:]]*$" "1var" ()
false
```

12.16.5 Empty and Overlapping Matches

Empty matches are permitted as well, subject to the constraint that at most one match is reported for each position (which also prevents looping). And of course an empty match will only be reported if nothing else matches. For instance:

```
==> regex "g" "" "foo" (regpos 0)
[0,1,2,3]

==> regex "g" "o*" "foo" (regpos 0)
[0,1,3]
```

(The usefulness of such constructs might be questionable, but see Section 12.16.7 [Performing Replacements], page 163, for a reasonable example.)

As already mentioned, only non-overlapping matches are reported with the `g` option. However, occasionally it might be useful to also determine overlapping matches, and for this purpose the `G` option is provided. For instance, suppose that we want to produce a table showing which letters follow another given letter in a text. The first step is to produce a list of all pairs of adjacent letters, and since we really need *all* pairs here, we employ the `G` option:

```
==> regex "G" "[[:alpha:]]{2}" "silly" (reg 0!0,reg 0!1)
[("s","i"),("i","l"),("l","l"),("l","y")]
```

Now it is an easy matter to collect the desired information using, e.g., a dictionary and a little bit of “list voodoo:”

```
==> def add = lambda D (lambda (X,Y) (update D X (insert (D!X) Y)))

==> def D = foldl add (mkdict emptyset (map fst _)) _

==> zip (keys D) (map list (vals D))
[("i",["l"]),("l",["l","y"]),("s",["i"])]
```

12.16.6 Splitting

In previous examples we saw how we can tokenize a string by matching its constituents (see Section 12.16.4 [Basic Examples], page 160). Sometimes one also has to go the other way round, i.e., split a string into arbitrary tokens separated by “non-tokens”. E.g., we might want to split a string into tokens delimited with whitespace, which can be described by an expression like `"[ \t\n]+"`, or, in a locale-independent fashion, using `"[[:space:]]+"`. To do this with the `regex` function, we must find all text that is *between* matches, and any trailing text after the last matched delimiter. The `regskip` function lets us do this as follows:

```
==> regex "g" "[[:space:]]+" "The little\t brown\n fox." regskip \
++ [regskip]
["The","little","brown","fox."]
```

Here the regskip expression argument to regex is used to access the intervening tokens, and the final ++ [regskip] appends the last token.

We can use this method to define our own little regular expression tokenizing function as follows:

```
regsplit OPTS REGEX S = regex OPTS REGEX S regskip ++ [regskip];
```

With this definition, the above example works as follows:

```
==> regsplit "g" "[[:space:]]+" "The little\t brown\n fox."
["The","little","brown","fox."]
```

We could also omit the g option, in which case only the first delimiter match is used as a splitting point:

```
==> regsplit "" "[[:space:]]+" "The little\t brown\n fox."
["The","little\t brown\n fox."]
```

Splitting with an empty or optional pattern also works as expected:

```
==> regsplit "g" "" "some text"
["","s","o","m","e"," ","t","e","x","t",""]

==> regsplit "g" " ?" "some text"
["","s","o","m","e","","t","e","x","t",""]
```

12.16.7 Performing Replacements

A similar idea is also used to replace matches with new text. You can define a substitution function, which replaces each match with the result of an expression, as follows:

```
special regsub ~OPTS ~REGEX ~S EXPR;
regsub OPTS REGEX S EXPR = strcat (regex OPTS REGEX S (regskip++EXPR))
                           ++ regskip;
```

For instance:

```
==> regsub "g" "[[:alpha:]][[:alnum:]]*" "1var foo 99 BAR $%&" \
(sprintf "-*-%s-*-" (reg 0))
"1-*-var-*- -*-foo-*- 99 -*-BAR-*- $%&"
```

As usual, if we only want to replace the first match, we can omit the g option:

```
==> regsub "" "[[:alpha:]][[:alnum:]]*" "1var foo 99 BAR $%&" \
(sprintf "-*-%s-*-" (reg 0))
"1-*-var-*- foo 99 BAR $%&"
```

And here's an example showing empty and optional patterns at work:

```
==> regsub "g" "" "some text" " "
" s o m e   t e x t "
```

```
==> regsub "g" " ?" "some text" ":"
":s:o:m:e::t:e:x:t:"
```

12.16.8 Submatches

It is also possible to access the parts of matches, called "groups", which are defined by parenthesized subexpressions of the regular expression. Groups are numbered starting at 1, counting opening parentheses from the left. The text matched by a group can be accessed using `reg N` where `N` is the group number.

For instance, suppose we want to parse environment lines, such as those returned by the shell's `set` command. We assume that each line with a definition looks like `VARIABLE=VALUE`, which can be described by the pattern `"^([^=]+)=(.*)$"`, in which group 1 denotes the variable and group 2 the value. Thus we can obtain the parts of a definition using a `regex` search like the following:

```
==> regex "" "^([^=]+)=(.*)$" "VARIABLE=VALUE" (reg 1,reg 2)
[("VARIABLE","VALUE")]
```

We can turn this into an `env` function which returns the current environment as a list of (variable,value) pairs as follows. Here, we use the `fget` function to read in the whole environment as a single string from a pipe opened on the `set` command. The environment string is then parsed using the `n` option of regex, which makes the `^` and `$` anchors match the beginning and end of each line, respectively.

```
env         = regex "gn" "^([^=]+)=(.*)$" envget (reg 1,reg 2);

envget      = fget (popen "set" "r");
```

Given these definitions, we can now get hold of the whole environment and, e.g., turn it into a dictionary which can be accessed in a convenient fashion using the functions from the `dict.q` standard library module:

```
==> def envdict = dict env

==> envdict!"SHELL"
"/bin/bash"
```

When a pattern consists of multiple alternatives, such as `"(foo)|(bar)"`, then some of the groups might participate in a match, while others don't. You can check whether a group matched by testing the corresponding `regpos` value. If the value is nonnegative, then the group matched; otherwise (the value is `-1`) it didn't. You can also use the `regs` function to determine the list of all groups that matched. For instance:

```
==> regex "" "(foo)|(bar)" "foo" (regpos 1,regpos 2)
[(0,-1)]

==> regex "" "(foo)|(bar)" "bar" regs
[[2]]
```

So here's a simple way to implement a lexical analyzer which distinguishes between different kinds of tokens. For example, let us tokenize identifiers and integers, skip whitespace, and return all remaining characters as literals.

```
def SYN = "([[:alpha:]][[:alnum:]]*)|(-?[[:digit:]]+)|([^[:space:]])";
def TOK = (none,ident,num,id);

toks S  = regex "g" SYN S ((TOK!(hd regs)) (reg (hd regs)));
```

Note that the literal characters are mapped to the standard library `id` function, the identity operation. Hence these characters are represented by themselves, whereas other tokens are denoted by appropriate constructor terms, `ident S` and `num S` in our example, which contain the actual text of the token as arguments. Let's see this in action:

```
==> toks "foo -99; bar 99;"
[ident "foo",num "-99",";",ident "bar",num "99",";"]
```

The token sequence is now ready to be processed by a higher-level routine such as a parser, but this is another story which will be told another time ...

Yet another way to employ submatches are the *back references*. You can specify that a group is to be repeated by using the notation `\N`, where `N` is a single digit between 1 and 9, in the regular expression argument. For instance, `regex "g" "([ab]+)\\1"` will match all substrings which have two consecutive identical sequences of `a`'s and `b`'s.

Back references add considerably to the matching complexity (some matches may take exponential time) and hence should be avoided. In fact, it can be shown that regular expression matching with back references is NP-complete.

12.16.9 Nested Searches

Regular expression searches can also be nested, by invoking the `regex` function recursively in the expression argument of another `regex` call. This is useful, e.g., if `regex` is used to tokenize a string, and we want to further analyze the individual tokens. We could do this by mapping a `regex` call to the finished token list, but it may be more efficient to perform the second search right away on each individual token. Here's an abstract example, which finds all substrings delimited by `c`'s, and counts the number of consecutive `a`'s and `b`'s in them:

```
==> regex "g" "[^c]+" "abbacbaaca" \
(regex "g" "a+|b+" (reg 0) (reg 0!0,#reg 0))
[[("a",1),("b",2),("a",1)],[("b",1),("a",2)],[("a",1)]]
```

12.17 Additional Integer Functions

The Q interpreter implements basic integer arithmetic using the GMP (GNU multiprecision) library. With the `clib` module you also get some of the more advanced GMP routines, namely integer powers and roots, and various number-theoretic functions.

12.17.1 Powers and Roots

```
public extern pow M N, root M N, intsqrt M;
```

These functions compute (exact) powers and (integer parts of) roots of integers. The results are always integers; roots are truncated to the largest integer below the exact root value. The `pow` function computes the `N`th power of `M` (`N >= 0`), `root` the integer part of the `N`th root of `M` (`M >= 0`, `N > 0`), and `intsqrt` the integer part of the square root of `M >= 0`.

```
public extern powmod K M N, invmod K M;
```

The `powmod` function returns the (smallest nonnegative) `N`th power of `M` modulo `K`, where `K` must be nonzero. This function is much more efficient than `pow M N mod K` if `N` is large. The `invmod` function computes the (smallest positive) multiplicative inverse of `M` modulo `K` (if it exists, i.e. if `M` is nonzero and relatively prime to `K`). In other words, `invmod K M` returns a number `N` in the range `1..K-1` s.t. `M*N = N*M = 1` modulo `K`.

12.17.2 Prime Test

```
public extern isprime N;
```

The `isprime` function implements the Miller-Rabin algorithm for probabilistic prime testing. If `isprime N` (where `N > 1` is an integer) returns `true` or `false`, then `N` is surely prime or composite, respectively. If the function fails, then the number is composite with a probability of only `(1/4)^ISPRIME_REP`, where the `ISPRIME_REP` variable denotes the number of repetitions of the probabilistic prime test.

To avoid the overhead of checking `ISPRIME_REP` each time the `isprime` function is invoked, `isprime` determines the value of the variable once, at the time of its first invokation. Thus, if you set this value, make sure to set it *before* you call `isprime` for the first time. Reasonable `ISPRIME_REP` values vary from 5 to 10; if the variable is not set when `isprime` is first called, a default value of 5 is used, which means that the probability of `isprime` failing on a composite will be 1/1024.

When using this function, your script should provide an appropriate default definition which handles the case that the `isprime` operation of this module fails. There are basically three different ways to cope with a failing `isprime` call. First, you can make your program abort, e.g., using an "error rule" like the following:

```
isprime N:Int = error "Prime test failed!";
```

Given that the test will probably not fail very often, this might be a viable option – you can always run your script again and hope that it finishes successfully, since the probabilistic test will choose different random parameters each time it is invoked.

Second, you can pretend that the prime test succeeded by providing the definition:

```
isprime N:Int = true;
```

With this definition there will be a small probability that you mistake a composite for a prime, but for some algorithms this might be tolerable.

Third, you can play safe by providing your own primality test which catches those few cases when the probabilistic test fails. For instance, employing the usual trial division method:

```
isprime N:Int = true if N = 2;
              = false if N and 1 = 0; // even number
              = try_div 3 (intsqrt N) N otherwise;

try_div L M N = true if L > M;
              = false if N mod L = 0;
              = try_div (L+2) M N otherwise;
```

Of course such an exhaustive search will take a *long* time for big numbers, but this might be acceptable if the default rule is not invoked very often.

12.17.3 Other Number-Theoretic Functions

```
public extern gcd M N, lcm M N;
```

These are the usual greatest-common-divisor and least-common-multiple functions. The gcd will always be positive, and the lcm will have the same sign as the product.

```
public extern fact M N, rem M N;
```

The `fact` and `rem` functions are used to help in factoring. The `fact` function returns the multiplicity of `N` in `M` (i.e., the maximum `L` s.t. `pow N L` divides `M`), and `rem` removes all `N` factors from `M`, i.e., `rem M N = M div pow N (fact M N)`. Both functions require that `M` is nonzero and `N` positive.

```
public extern jacobi M N;
```

This function computes the *Jacobi symbol* `M` over `N`, which can be used to find quadratic residues of an odd prime. `M` can be an arbitrary integer, but `N` must be positive; the value of the function is always 1, 0 or -1, where the value 0 indicates that `gcd M N > 1`. In the case of an odd prime `N`, we have that `jacobi M N = 1` iff `M` is a nonzero quadratic residue modulo `N`.

12.17.4 Examples

Determine all invertible numbers modulo 9 and their corresponding inverses:

```
==> filter (lambda (X,Y) (isnum Y)) \
(zip (nums 1 8) (map (invmod 9) (nums 1 8)))
[(1,1),(2,5),(4,7),(5,2),(7,4),(8,8)]
```

Check the result:

```
==> map (lambda (X,Y) (X*Y mod 9)) _
[1,1,1,1,1,1]
```

Find all (nonzero) quadratic residues modulo 7:

```
==> filter (lambda X (jacobi X 7 = 1)) (nums 1 6)
[1,2,4]
```

Decompose 858330 into prime factors (brute force):

```
==> def N = 858330, P = filter isprime (nums 2 N)

==> filter (lambda (X,Y) (Y>0)) (zip P (map (fact N) P))
[(2,1),(3,3),(5,1),(11,1),(17,2)]
```

Well, that's probably not the right way to do it. So here's a reasonable factorization algorithm, using both `fact` and `rem`. This method is a *lot* faster – but still not fast enough for code breaking.

```
factor 0            = [];
factor N:Int        = factor (-N) if N<0;
                    = [(2,fact N 2)|factor_from 3 (rem N 2)]
                         if N and 1 = 0; // even number
                    = factor_from 3 N
                         otherwise;

factor_from P N     = [] if P > N;
                    = [(P,fact N P)|factor_from (P+2) (rem N P)]
                         if N mod P = 0; // P divides N (must be prime)
                    = factor_from (P+2) N
                         otherwise; // not a factor, try the next one
```

For instance, try the following:

```
==> factor 807699854836875
[(3,1),(5,4),(7,2),(11,5),(13,2),(17,1),(19,1)]
```

Check the result:

```
==> prd (map (lambda (X,Y) (pow X Y)) _)
807699854836875
```

12.18 C Replacements for Common Standard Library Functions

Last not least, the `clib` module also provides "builtin" replacements for various standard library functions:

```
public extern stdlib::append Xs Y, stdlib::cat Xs, stdlib::mklist X N,
  stdlib::nums N M, stdlib::numsby K N M, stdlib::reverse Xs,
  stdlib::tuplecat Xs;

public extern string::chars S, string::join DELIM Xs,
  string::split DELIM Xs, string::strcat Xs;
```

These functions are *much* more efficient, both in running time and memory requirements, than the "standard" definitions in `stdlib.q` and `string.q`, and improve the performance of many common list and string processing tasks. In some cases, the provided operations are even several orders of magnitude faster. In particular, the standard definitions of the `cat`, `strcat` and `tuplecat` operations are *very* slow in comparison, because they are implemented using repeated concatenation and hence take quadratic running time. In contrast, all functions provided here are guaranteed to run in linear time.

A faster sorting routine is provided as well:

```
public extern sort P Xs;
```

This function works like the `msort` and `qsort` operations provided by the `sort.q` module, but is implemented using the quicksort routine from the C library. Note that, in difference to `msort`/`qsort`, the `sort` function does not perform stable sorting, so you still have to use `msort` or `qsort` if this feature is required.

Appendix A Q Language Grammar

This appendix summarizes the syntactical rules of the Q language. Please refer to Chapter 3 [Lexical Matters], page 19, for a detailed discussion of lexical matters. The syntax rules are given in BNF with the following extensions:

- `{ }` denotes repetition; the enclosed elements may be repeated zero or more times.
- `[ ]` denotes optional parts; the enclosed elements may be omitted.

The left-hand and right-hand sides of syntax rules are separated with a colon, and the `|` symbol denotes different alternatives. Identifiers stand for nonterminal grammar symbols, and terminal symbols are enclosed in single quotes.

In order to keep the grammar to a reasonable size, it does *not* specify the precedence of operator symbols. Expressions are parsed according to the precedence and associativity of operator symbols specified in Section 6.4 [Built-In Operators], page 33. A '`=`' symbol occuring unparenthesized in an equation separates left- and right-hand side.

```
script                   : {declaration|definition}

declaration              : 'import' module-spec {',' module-spec} ';'
                         | 'include' module-spec {',' module-spec} ';'

                         | prefix headers ';'
                         | [scope] 'type' unqualified-identifier
                           [':' identifier] ['=' sections] ';'
                         | [scope] 'extern' 'type' unqualified-identifier
                           [':' identifier] ';'
                         | [scope] 'type' qualified-identifier
                           ['as' unqualified-identifier] ';'

                         | '@' ['+'|'-'] unsigned-number

module-spec              : module-name ['as' unqualified-identifier]
module-name              : unqualified-identifier
                         | string

prefix                   : scope
                         | [scope] modifier {modifier}

scope                    : 'private'|'public'

modifier                 : 'const'|'special'|'extern'|'var'

headers                  : header {',' header}

header                   : unqualified-identifier
                           {['~'] variable-identifier}
                         | qualified-identifier
                           {['~'] variable-identifier}
```

```
                         'as' unqualified-identifier ';'

sections               : section {'|' section}

section                : [prefix] headers

definition             : expression '=' expression qualifiers ';'
                         {'=' expression qualifiers ';'}
                       | 'def' expression '=' expression
                         {',' expression '=' expression} ';'
                       | 'undef' identifier {',' identifier} ';'

qualifiers             : {qualifier}

qualifier              : condition
                       | where-clause

condition              : 'if' expression
                       | 'otherwise'

where-clause           : 'where' expression '=' expression
                         {',' expression '=' expression}

expression             : identifier
                       | variable-identifier ':' identifier
                       | number
                       | string
                       | expression expression
                       | unary-op expression
                       | expression binary-op expression
                       | '(' [element-list] ')'
                       | '[' [element-list] ']'
                       | '(' op ')'
                       | '(' expression binary-op ')'
                       | '(' binary-op expression ')'

element-list           : expression-list ['|' expression]

expression-list        : expression {',' expression}

op                     : unary-op|binary-op

unary-op               : '-'|'#'|'not'|'''|'`'|'~'

binary-op              : '^'|'!'|'++'|'+'|'-'|'\'|'*'|'/'|'div'|'mod'
                       | 'and'|'or'|'and' 'then'|'or' 'else'
                       |'<'|'>'|'='|'<='|'>='|'<>'|'||'

identifier             : unqualified-identifier
```

```
                        | qualified-identifier

qualified-identifier    : unqualified-identifier '::'
                          unqualified-identifier
unqualified-identifier  : variable-identifier
                        | function-identifier

variable-identifier     : uppercase-letter {letter|digit}
                        | '_'
function-identifier     : lowercase-letter {letter|digit}

number                  : ['-'] unsigned-number
unsigned-number         : '0' octdigitseq
                        | '0x' hexdigitseq
                        | '0X' hexdigitseq
                        | digitseq ['.' [digitseq]] [scalefact]
                        | [digitseq] '.' digitseq [scalefact]

digitseq                : digit {digit}
octdigitseq             : octdigit {octdigit}
hexdigitseq             : hexdigit {hexdigit}

scalefact               : 'E' ['-'] digitseq
                        | 'e' ['-'] digitseq

string                  : '"' {char} '"'

letter                  : uppercase-letter|lowercase-letter
uppercase-letter        : 'A'|...|'Z'
lowercase-letter        : 'a'|...|'z'|'_'

digit                   : '0'|...|'9'
octdigit                : '0'|...|'7'
hexdigit                : '0'|...|'9'|'a'|...|'f'|'A'|...|'F'

char                    : any character but newline and "
```

Appendix B Using Q

The following discussion assumes that you have already installed the Q programming system on your machine. In particular, you should make sure that the programs `q` and `qc` have been copied to a directory which is searched for executables, and that the standard library scripts have been installed properly.

You may also refer to the UNIX manual page q(1) for a brief description of the Q programs.

The Q programming system consists of two main parts: the compiler and the interpreter. The compiler is used to translate Q scripts into a binary format, a so-called "bytecode" file, which is then executed by the interpreter. Since the interpreter invokes the compiler automatically when you invoke it with a source script, you usually do not have to care about the compiler yourself. But if you wish you can also run compiler and interpreter separately.

B.1 Running Compiler and Interpreter

Here is how you normally run a script with the interpreter:

`q [`*options*`] [` *source-or-code-file* `[`*argument* `...] ]`

The interpreter will then start up, display its prompt, and wait for you to type some expressions or other commands (see Section B.2 [Command Language], page 181). You can exit the interpreter by typing `quit` (which invokes the built-in `quit` function), or by entering the end-of-file character at the beginning of the input line.

At most one source or code file may be specified; the remaining command line arguments can be accessed in the interpreter by means of the built-in `ARGS` variable. If the given file is in bytecode format, it is loaded immediately. Otherwise, the interpreter invokes the compiler to translate the source script to a temporary code file. Then, if the source script was translated successfully, the interpreter loads the generated code file and you can start typing in expressions and commands. (After the code file has been loaded by the interpreter, it is no longer needed and is discarded immediately. If you want to keep the code file, you must invoke the compiler separately, see below.)

Scripts and code files which have no absolute path specified are searched for on Q's library path (as given by the `QPATH` shell environment variable or with the `--path` option, see below). As with the `PATH` environment variable, the individual directories are separated with a colon '`:`' (semicolon '`;`' on DOS/Windows systems). The directories are searched in the indicated order; if the file cannot be found on the search path, an attempt is made to open it by its name. You should include the '`.`' directory in the search path if you wish to search the current directory prior to other locations. The default search path is usually something like `.:/usr/share/q/lib:/usr/lib/q` which causes compiler and interpreter to first search the current directory, and then other (system-dependent) locations where "library scripts" are kept.

By default, the interpreter automatically includes the `prelude.q` script (which will be searched for on the library path as usual) as the first module in your program. The prelude provides the definitions which are available without explicit import or include; normally it is used to include the standard library, see Chapter 4 [Scripts and Modules], page 23. You can also suppress the automatic inclusion of the prelude with the `--no-prelude` compiler option, see below, or provide your own, see Section B.3 [Setting up your Environment], page 187.

The interpreter can also be invoked with no arguments at all, in which case an empty script is assumed. This means that only the built-in operations of the Q language will be available, as well as the definitions in the `prelude.q` script (unless the `--no-prelude` option has been specified). (If you want to specify an empty script while still supplying arguments, specify `""` as the script name.)

If you want to avoid recompilations of large scripts, you can also run compiler and interpreter separately. For this purpose, you first invoke the compiler as described below to translate your script(s) to a bytecode file. Then you can invoke the interpreter, specifying the name of the bytecode file (normally `q.out`, unless another code file name has been specified with the `-o` compiler option). The interpreter can then load the bytecode file and start execution immediately.

The compiler is executed as follows:

`qc [`*options*`] [`*source-file* `...]`

It compiles the specified files and writes the resulting bytecode to the code file, normally `q.out`. Note that multiple source files may be specified. The first file denotes the main script, whose namespace defines the global scope; the name of the main script can also be missing or be specified as the empty string `""`, to indicate an empty main script. The remaining source files are additional scripts to be imported into the global scope.

The compiler recognizes the following options:

`--help, -h`
: Print a short help message.

`--list=`*list-file*`, -l` *list-file*
: When this option is used, error messages are written to the specified file instead of the standard error device.

`--no-code, -n`
: This option suppresses code file generation, which is useful if you only want to check the given scripts for syntax errors.

`--no-prelude`
: Suppresses automatic inclusion of the prelude.

`--output=`*output-file*`, -o` *output-file*
: This option allows you to change the default name of the generated code file.

`--path=`*path*`, -p` *path*
: This option sets the library path used to search for source and code files (default: system-dependent hardcoded default or value of the `QPATH` shell variable). If the *path* argument starts or ends with the path delimiter (‘`:`’ on UNIX) then the given path is appended or prepended to the current path, respectively.

`--prelude=`*file*
: Specify the name of the prelude script (default: `prelude.q`).

`--tablesize=`*size*`, -t` *size*
: This option allows you to determine the size of the runtime hash table used for the hashing of identifiers. For best performance this should be a prime number; the default size is 5711. With this option, you can change the size of the hash table if you are compiling a script with a very large number of function symbols, or a script which dynamically creates a lot of symbols. Inreasing the size of the hash table will reduce the "load" of the table and thereby improve the performance of symbol hashing at runtime.

`--verbose, -v`
: Display processed source files and statistics.

`--version, -V`
: This option causes version number and copyright information to be displayed, after which the program terminates.

Options are generally parsed using `getopt` which means that multiple single-letter options may be combined (e.g., `-vn`) and a parameterized option can be followed immediately by the corresponding argument (e.g., `-ofoo.out`); see `getopt` in Section 3 of the UNIX manual for details. All programs also accept long option names following the GNU conventions, e.g., you can use `--output` instead of `-o`. Note that in the case of parameterized long options the parameter must either follow in the next command line argument, or it must be separated from the option name by a `=` character. Long options may be abbreviated by an unambigious prefix of the option name. For instance, you may use `--out` instead of `--output`. Option parsing ends as soon as the `--` option is found; the remaining command line arguments are taken as normal parameters, even if they start with '`-`'.

After a script has been compiled, you can invoke the interpreter on the generated code file as already discussed above (in this case, the code file is *not* discarded). Below we describe the options which are accepted by the interpreter. Note that the interpreter also recognizes all of the compilation options discussed above, and will pass them on when it invokes the compiler.

`--break`
: This option causes invokation of the debugger in case of an exception like `Ctl-C`, invalid rule qualifier, user-raised exception (`throw`), or a call to the builtin `break` function. See Appendix D [Debugging], page 199, for details.

`--command=`*cmd*`, -c` *cmd*
: This option instructs the interpreter to execute the given command in "batch mode" (see below).

`--debug, -d`
: This option causes activation of the debugger. See Appendix D [Debugging], page 199, for details.

`--debug-options`
: Set various options which control how much detail the debugger prints when showing a reduction or rule. See Appendix D [Debugging], page 199, for details.

`--dec, --hex, --oct`
: Sets the integer output format to decimal (default), hexadecimal or octal, respectively.

`--echo, -e`
: Enable the echoing of commands when running in batch mode.

`--exitrc=`*file*
: This option allows you to specify the name of the file which is executed when the interpreter is exited; the default is usually `~/.qexitrc`.

`--fix[=`*prec*`], --sci[=`*prec*`], --std[=`*prec*`]`
: Sets the floating point output format to fixed point, scientific (always using exponents) or standard (only using exponents if it results in a shorter notation; this is the default). For the scientific and standard formats, the given precision denotes the number of significant digits to be printed, and defaults to `15`. For the fixed point format, the precision specifies the number of digits following the decimal point (`2` by default).

`--gc[=`*tolerance*`]`
: Enables the built-in garbage collector which defragments the heap after each evaluation (see comments on memory management below).

`--gnuclient`
: Lets the interpreter run as a client of gnuserv(1). This allows the interpreter to interact with an emacs process driving the interpreter, e.g., in "comint" mode. In particular, the `edit` and `help` commands of the interpreter (see Section B.2 [Command Language], page 181) will then be handled by sending corresponding Emacs Lisp commands via the gnuclient command instead of processing them directly. The actual name of the gnuclient program can be set using the `GNUCLIENT_PROGRAM` environment variable. Note that this option also disables the interpreter's own history processing as it is assumed that emacs takes care of it.

`--help, -h`
: Print a short help message.

`--histfile=`*file*
: Specify the name of the file in which the command history is stored (default: `~/.q_history`).

`--histsize=`*size*
: Specify the size of the command history (default: 500).

`--initrc=`*file*
: Specify the name of the file which is executed at startup (default: `~/.qinitrc`).

`--interactive, -i`
: Indicates that the interpreter is running interactively.

`--memsize=`*size*
: Specify the maximum size of the expression heap (default: 4096000; see comments on memory management below).

`--no-editing`
: Disable command line editing using readline.

`--no-exitrc`
: Suppress execution of the exitrc file.

`--no-initrc`
: Suppress execution of the initrc file.

`--path=`*path*`, -p` *path*
: As with `qc`, sets the library path used to search for source and code files.

`--prompt=`*str*
: Set the prompt string (default: '`\n==>` ').

`--quiet, -q`
: This option causes a quiet startup; it suppresses the sign-on message.

`--source=`*file*`, -s` *file*
: Source file with interpreter commands (batch mode).

`--stacksize=`*size*
: Specify the maximum size of the internal stacks (default: 1024000; see comments on memory management below).

`--version, -V`
: As with `qc`, display version number and copyright information.

Options for Interactive and Batch Usage

Unless one of the `-c` and `-s` options is specified, the interpreter starts up in *interactive mode*, in which the user is repeatedly prompted to enter an expression, and the interpreter answers with the corresponding normal form. The interpreter also understands some other special commands, which are discussed in Section B.2 [Command Language], page 181. With the `-c` and `-s` options, the interpreter runs in *batch mode*, in which it executes the commands and command files specified on the command line, prints the results on the standard output device, and then exits immediately. The `-d` option invokes the symbolic debugger built into the interpreter which allows you to trace evaluations, cf. Appendix D [Debugging], page 199.

When the interpreter starts up in interactive mode, and is connected to a terminal, it also provides a command history through the GNU readline library. (See section "Command Line Editing" in *The GNU Readline Library*.) That is, the lines you type are memorized, and you can search for and edit previous commands. The command history is also saved to a file when the interpreter is exited, and completion of filenames, keywords, command names and defined function and variable symbols is supported as well. You can disable this feature with the `--no-editing` option.

You can redirect the interpreter's input and output streams to a file or a pipe as usual. The Q interpreter checks whether it is connected to a terminal in order to prevent garbled output when input or output is redirected. If you're feeling adventurous, try something like the following:

```
q -d <<EOF
def Xs = nums 1 50
msort (<) (reverse Xs)
EOF
```

The `-i` option can be used to force the interpreter to interactive mode even when input or output is redirected. This causes sign-on and prompt to be printed, and the standard output buffers to be flushed after each evaluation, which is useful when another program is driving the interpreter. (If you still want to suppress the sign-on message, use `-i` in combination with `-q`.) Furthermore, the `-i` option also suppresses the batch mode execution of `-c` and `-s` options.

The `--initrc` and `--exitrc` options allow you to specify command files which are executed when the interpreter starts and exits, which is commonly used to customize the environment when running the interpreter interactively, see Section B.3 [Setting up your Environment], page 187. The execution of these files will be suppressed when the interpreter is run in batch mode, or when using the `--no-initrc` and `--no-exitrc` options.

Memory Management

The Q interpreter has a fully dynamic memory management system. In particular, it automatically resizes both the expression heap and the evaluation stack as required during the course of a computation. Thus you normally do not have to worry about memory management issues.

However, the `--stacksize` option allows you to specify a maximum size for the evaluation stack (more precisely, it sets the limit for both the expression and the rule stack), which keeps the stack from growing to infinity. This is useful, e.g., to catch infinite recursion. Note that as the stack is also used internally for the purpose of parsing expressions, a reasonable minimum size is required for the interpreter to function properly. Therefore the interpreter enforces that the given size is not less than a certain minimum value (usually 100); otherwise the option is ignored and the stack size is reset to its default value.

Similarly, the `--memsize` option limits the number of expression nodes on the heap and thereby prevents the interpreter from using up all available main memory on your system. You can disable both limits by specifying `0` as the argument to the corresponding option.

The interpreter manages expression memory using a reference counting scheme, thus no disruptive "garbage collection" process has to be invoked during an evaluation. However, the interpreter *does* have a built-in garbage collector for the purpose of defragmenting the expression heap and returning unused expression memory to the system after each evaluation. This functionality is not required for proper operation of the interpreter and is disabled by default. You might wish to enable it with the `--gc` option if conservative memory usage is required.

The `--gc[=`*tolerance*`]` option causes the garbage collector to be called after an expression evaluation if the percentage of free expression cells exceeds the given tolerance. For instance, a tolerance value of 50 means that unused expression memory will be returned to the system when the interpreter detects that it uses more than twice the memory required to store all expressions on the heap. The default tolerance value of zero forces garbage collection after each evaluation.

Please note that in the current implementation garbage collection is only performed after an evaluation has been finished, and only if no secondary threads are still executed in the background. Moreover, whether the freed memory actually becomes available to other applications depends on your system's memory manager. We also remark that in order to defragment the expression heap, the garbage collector has to copy the current expression memory to a fresh memory arena, which temporarily needs extra memory and may take some time if a lot of big expressions are currently on the heap. Allocating the new memory arena may fail if not enough main memory is available; in this case garbage collection is abandoned and the interpreter continues to use the old fragmented heap until the garbage collector is started the next time.

B.2 Command Language

The commands understood by the interpreter make up a simple *command language* which is described in the following. Commands can also be passed to the interpreter by means of the `-c` option, and you can put a sequence of commands into a file and execute them using the `-s` option (see Section B.1 [Running Compiler and Interpreter], page 175). Command files can also be executed in the interpreter using the `source` command (see below). Moreover, when the interpreter starts up and is exited in interactive mode, it reads and executes commands from its initialization and termination files, which allows you to customize your environment according to your taste (see Section B.3 [Setting up your Environment], page 187).

The command language is line-oriented, i.e., commands are terminated by the newline character. As usual, line ends can be escaped by putting the `\` character immediately before the end of the line; this also works inside of string literals. A line may contain multiple expressions and commands, which are separated by the `;` delimiter:

```
==> def X = foo Y; X; X/2
```

(You may also terminate a command line with an extra `;` delimiter, but this is not required.)

The three commands used most frequently are the following:

expression or `?` *expression*
: Evaluate the given expression and print the resulting normal form.

`def` *expression* `=` *expression*`, ...`
: Define free variable values.

`undef` *variable*`, ...`
: Undefine the given variables.

After a variable has been assigned a value, it will be replaced by its associated value whenever it is evaluated. This allows you to store intermediate results in a computation, set the values of free variables occuring in a script, and to define new functions interactively in the interpreter. For instance:

```
==> def double = (*) 2, X = double 4; X
```

```
8
```

A subsequent `undef` removes the variable definition. For instance:

```
==> undef X; X
X
```

As discussed in Section 7.3 [Free Variables], page 46, the left-hand side of a variable definition may actually be an arbitrary pattern to be matched against the right-hand side value.

We remark that, when starting up, the interpreter always defines the variables `INPUT`, `OUTPUT`, `ERROR` and `ARGS`. `INPUT` and `OUTPUT` are set to the standard input/output devices used by the terminal I/O functions (see Section 10.5.1 [Terminal I/O], page 82), while `ERROR` denotes the standard error device normally used to display error messages. You can use these symbols when passing a file to one of the built-in file I/O functions (see Section 10.5.2 [File I/O], page 83). As already mentioned, the `ARGS` variable is assigned a string list which contains the additional command line arguments the Q interpreter was invoked with (starting with the script name). These variable symbols are "read-only"; they cannot be re- or undefined.

There is one other special built-in variable, the "anonymous" variable '`_`', which, when used in an expression or the right-hand side of a variable definition, refers to the result of the most recent expression evaluation:

```
==> 1/3
0.333333333333333

==> 1/_
3.0
```

Note that expressions and definitions have the same syntax as in the Q language. Besides this, the interpreter allows you to escape a command to the shell:

`!` *command*

The given command is executed using the shell. Unlike the other types of commands, the shell escape must be on a line by itself.

The interpreter also provides some other special commands which extend the expression and definition syntax of the Q language. They consist of a command name and possibly some parameters. Unlike the `def` and `undef` keywords, which are also reserved words in the Q language itself, the command names are only treated specially when occuring at the beginning of a command; anywhere else they are treated as ordinary Q identifiers. You can escape a command name at the beginning of a command as an ordinary identifier (and thereby force the remainder of the command to be treated as an ordinary expression) by prefixing it with '`?`'. Thus,

```
==> ? echo "some string"
```

will evaluate an expression involving some `echo` function instead of executing the `echo` command.

The parameters of special commands are simply string literals which may be quoted or unquoted, and which are delimited with whitespace, very much like in the UNIX command shell. In particular, the interpreter will not try to evaluate these parameters. However, the usual escapes for special characters are supported. The following table lists the available commands, where `[...]` is used to denote optional arguments.

`copying` Show the GNU license conditions.

`help` [*S*] Start the GNU info reader (as specified by the `INFO_PROGRAM` environment variable, `info` by default) with the Q manual; if a keyword *S* is given, perform an index search. When the interpreter has been started with the `--gnuclient` option (see Section B.1 [Running Compiler and Interpreter], page 175), then the command submits a request to invoke the info reader of a driving emacs process using the gnuclient program.

`path` [*S*] Set the library search path (`QPATH`) to *S* (print the current setting if *S* is omitted). If the *S* argument starts or ends with the path delimiter (‘`:`’ on UNIX) then the given path is appended or prepended to the current path, respectively.

`prompt` [*S* [*S2* [*S3*]]]
: Set the prompt string (print the current setting if *S* is omitted). The `\\v`, `\\s`, `\\w`, `\\W`, `\\m` and `\\M` escape sequences may be used in the prompt string to denote the Q version number, host system information, current working directory, basename of the current directory, full pathname of the current script, and the script’s basename, respectively. (Note the double escapes; these are required to make one \ get through to the command.) The optional second and third argument allow you to change the default continuation (‘`> `’) and debugger prompt (‘`: `’); no special escape sequences are substituted in these strings.

`dec, hex, oct`
: Sets the integer output format to decimal, hexadecimal and octal, respectively.

`fix` [*prec*]`, sci` [*prec*]`, std` [*prec*]
: Sets the floating point output format to fixed point, scientific and standard, respectively (see also the corresponding interpreter command line options in Section B.1 [Running Compiler and Interpreter], page 175). If the precision is omitted, it defaults to `2` for the fixed point format, and to `15` otherwise.

`histfile` [*S*]`, histsize` [*N*]
: Change the history filename and set the size of the history (print the current setting if no argument).

`stacksize` [*N*]
: Set the maximum stack size (print the current setting if no argument); see the discussion of the `--stacksize` option in Section B.1 [Running Compiler and Interpreter], page 175.

`memsize` [*N*]
: Set the maximum number of expression nodes on the heap (print the current setting if no argument); see the discussion of the `--memsize` option in Section B.1 [Running Compiler and Interpreter], page 175.

`stats [all]`
: Print statistics about the most recent expression evaluation in the main thread. If the optional `all` parameter is given then the command also lists finished background threads which have not been recycled yet. The given figures are the cpu time needed to complete the evaluation, the total number of reductions performed by the interpreter, and the number of expression nodes (heap cells) created during the course of the computation. The application of each builtin rule counts as a single reduction. Note that since different builtin rules may have different execution times (in fact, the time needed by many of the builtin operations varies with the size of the input), the number of reductions usually only provides a rough estimate of the actual running time. But this value is still quite useful to perform asymptotic analysis (up to linear factors, if one wants to be picky), and can also be used to compare profiling results obtained on different machines with different processors. The cell count is the *maximum* number of cells required at any time during the evaluation, which does *not* include the size of the input expression. This value is used to measure space requirements, and is proportional to the "real" memory requirements of the interpreter (each expression node requires some 24 bytes in the current implementation). But note that subexpressions may be "shared" between different expressions, and thus the cell count will usually be less than the tree size of the constructed expressions.

`debug [on|off|`*options*`]`
: When invoked with arguments `on` or `off`, activate or deactivate debugging; print the current setting if no argument is specified. Alternatively, you can also use this command to set various options which control how much detail the debugger prints when showing a reduction or rule. See Appendix D [Debugging], page 199, for details.

`break [on|off]`
: This command controls how the interpreter responds to an exception like `Ctl-C`, invalid rule qualifier, user-raised exception (`throw`), or a call to the builtin `break` function; see Appendix D [Debugging], page 199, for details. If no argument is given, the current setting is printed.

`echo [on|off], echo` *S*
: Enable or disable command echoing (print current setting if omitted), or echo the given string to standard output (with a newline character appended). When echoing is enabled, all command lines executed in batch mode (using the `-s` or `-c` option, or the `source` command) are echoed to standard output before they are executed. (You can inhibit command echoing by prefixing a command line with the `@` character.)

`chdir [`*S*`], cd [`*S*`]`
: Change to the given directory (your home directory if none given).

`pwd`
: Print the current working directory.

`ls [`*S* `...]`
: List the contents of the current directory, or the files matching the given patterns. This command simply invokes the corresponding UNIX command with the given arguments.

`which [`*S*`]`
: Print the full name of the script or command file (as given by a search on the library path) that would be run when the relative pathname *S* is given to the `run` or `source` command; if invoked without arguments, print the full pathname of the running script.

`edit [`*S*`]`
: Edit the given file (current script if none specified), using the editor specified by the `EDITOR` environment variable, or `vi` by default. When the interpreter has been started with the `--gnuclient` option (see Section B.1 [Running Compiler and Interpreter], page 175), then the command causes an edit request to be submitted to a driving `emacs` process using the `gnuclient` program.

`run [`*S* `[`*args* `...]]`
: Run the given script with the given arguments. If no script is specified, rerun the current script with the current arguments. (Note that to force the interpreter back to an empty script, you can specify `""` for the script name.) When in interactive mode, the `exitrc` and `initrc` files are sourced as usual. Note that this operation also clears the variable memory, so you might wish to save your variables *before* invoking this command, unless your `exitrc` file already takes care of this.

`import [[+|-]`*S* `...]`
: Rerun the current script with the current arguments, as with `run`, but also add or remove the given scripts to/from the current "import list". A '`+`' prefix (or no prefix at all) causes the module to be added to the import list and '`-`' removes the module, if possible. If no arguments are given then the import list is cleared. Modules on the import list are imported into the global scope (i.e., the namespace of the main script), in addition to the prelude and the imports of the main script. The current import list can be shown with the `imports` command, see below.

  Note that you can also add modules which are already included in the running script, but which are not currently visible in the global scope. On the other hand, you can only remove those modules which have actually been added to the import list, not any modules which are only included in the running script. The main script can be removed as well, which has the same effect as doing `run` with an empty script name.

  IMPORTANT: This command will only work as advertised when running a source script. If a bytecode file is run, the import list will be cleared, and subsequent `import` commands will have no effect.

`source` *S*`, .` *S*
: Source a file with interpreter commands (may be nested). As indicated, this command may be abbreviated as '`.`'. The given command file is searched on the Q library path.

`save [S], load [S]`
: Save and restore variable values to/from the given file (if no file specified, use the file last specified with `save` or `load`, or `.q_vars` by default).

  The `save` command stores all user-defined variables in the form of variable definitions, i.e., `def` commands. Floating point values are always stored with maximum internal precision, no matter what the current precision set with the `std`, `sci` or `fixed` commands is. The built-in variables are *not* saved with this command; thus, if you want to also save the value of the '_' variable, you must explicitly assign it to a user variable.

  The `load` command is very much like `source`, so the file may actually contain arbitrary interpreter commands, but no library path search is performed, so a path must be given unless the file is contained in the current directory.

  Putting a pair of `load` and `save` commands in your `initrc`/`exitrc` files (see Section B.3 [Setting up your Environment], page 187) is a convenient method to have the interpreter automatically remember the variable environment from a previous session. You should note, however, that the `load` command reevaluates all values in the context of the current script, and hence the results may be quite different from the original values if the script has been changed during invokations of the interpreter. Moreover, on many systems floating point values cannot always be reconstructed exactly from their textual representations. (On some systems the interpreter can be built with support for ISO C99 format conversions which eliminate this problem; see the README file included in the distribution for details.) Another potential obstacle is that the `load` command cannot reconstruct "external" objects (<<*typeid*>>, see Section 10.5.2 [File I/O], page 83, and Appendix C [C Language Interface], page 191), since these objects do not have a parseable representation at all.

`clear [`*variable* `...]`
: Undefine the given variables, or all user-defined variables if no variable is specified. In the latter case, the command also purges the value of the '_' variable, but the variables defined by the loaded scripts are *not* affected.

`modules`
: List all loaded modules, i.e., all scripts imported or included in the running script (including the prelude). External modules are indicated by a trailing '*'.

`imports`
: Like the `modules` command, but only list the modules visible in the global scope. This includes the main script itself, the prelude and all the modules it includes (normally the standard library), the imports and includes of the main script, and the extra imports specified with the `import` command (see above). The latter modules and the main script are indicated by an initial '+' character. (These are the modules which can be removed with the `import` command.)

`who [all]`
: List all user-defined variable symbols. This only lists variables which have been defined in the interpreter. If the optional `all` parameter is specified then all variables are listed.

`whos` *symbol* ...
: Describe the given (function, variable or operator) symbols. Prints useful information about each symbol, such as whether it is a function symbol or a

variable, name of the script file in which the symbol is defined, `const` and `special` attributes, etc.

`completion_matches S`
: List the possible completions of a token prefix. This command allows programs like `emacs` (see Appendix E [Running Scripts in Emacs], page 203) to implement completion when driving the interpreter as an inferior process.

B.3 Setting up your Environment

There are basically two ways in which you can tailor the Q interpreter to your needs: you can provide your own prelude providing "preloaded" function and variable definitions; and you can use the `qinitrc` and `qexitrc` command files to execute a sequence of interpreter commands when an interactive instance of the interpreter starts up and exits. We describe each of these in turn.

The prelude, which has already been discussed in Chapter 4 [Scripts and Modules], page 23, is just an ordinary Q script which is by default imported in all your scripts. Normally the prelude includes the standard library modules, but you can also provide your own prelude into which you can put any "standard" definitions you want to have available in all your scripts. To customize the prelude, you can either modify the version provided with the standard library (to provide your definitions system-wide), or put your private version somewhere on your library path where the compiler will find it prior to the standard one. In the latter case you can still have your version of the prelude include the standard prelude by renaming the latter using an '`as`' clause (see Chapter 4 [Scripts and Modules], page 23). E.g., your `prelude.q` script may contain something like the following (assuming the standard prelude is in `/usr/share/q/lib`):

```
// include the standard prelude as 'stdprelude'
include "/usr/share/q/lib/prelude.q" as stdprelude;
// my own definitions here ...
```

Let us now turn to the interpreter's initialization and termination files. As already mentioned, when the interpreter runs in interactive mode, it automatically sources a startup file (usually named `~/.qinitrc`, but you can change this with the `--initrc` option) before entering the command loop. Thus you can put some commands in this file which set up your initial environment, e.g., initialize some variables and load variable definitions from a previous session. A typical startup file may look as follows:

```
// sample init file

// common constants
//def inf = 1e999
//def e = exp 1.0
//def pi = 4.0*atan 1.0
//def i = (0,1)

// set your preferred defaults here
//path .:~/q:/usr/share/q/lib:/usr/lib/q
//fix 2
```

```
//stacksize 1024000
//memsize 4096000
//break on
//debug pathnames=y detail=all maxchars=80
prompt "\n\\M>> "

// read variable definitions from .q_vars file
load
```

Similarly, when the interpreter is exited, it normally sources the `~/.qexitrc` file (or the file named with the `--exitrc` option), which might take care of saving the values of currently defined variables:

```
// sample exit file

// autosave variables
save

// want your daily epigram?
//! fortune
```

B.4 Running Scripts from the Shell

The Q programming system has been designed primarily to facilitate interactive usage. However, with a little additional effort it can also be used to create standalone application programs which are invoked directly from the shell. For this purpose, your script should be equipped with some "main" function which acts as the entry point to your application. On UNIX systems, you can then use the `#!` ("shebang") notation to specify the Q interpreter as a language processor to be invoked on the script instead of the shell:

```
#!/usr/bin/q -cmain
```

If you include this line (which will be treated as a comment by the Q interpreter) at the top of your script and give the script execute permissions, most UNIX shells will be able to execute the script just like any other program. Another typical example is the following, which passes arguments to the `main` function:

```
#!/usr/bin/q -cmain ARGS
```

Most shells only accept a single argument with `#!`. This means that you *must* pass all compiler and interpreter options in one argument. For instance:

```
#!/usr/bin/q -dcmain ARGS
```

Obviously, this can be cumbersome and it also does not allow you to have more than one parameterized option. As a remedy, the Q compiler and interpreter let you pass options by including any number of additional "shebang" lines in the format

`#!` *option*

at the beginning of the main script. (This even works if there is no `#!/usr/bin/q` line at the top of the script.) Note that this is not a shell feature, but an extension provided by the

Q programming tools. Also note that here the `#!` symbol and the option *must* be separated with whitespace, like so:

```
#!/usr/bin/q
#! -ofoo
#! -cmain ARGS
```

Using these facilities, fairly elaborate application setups can be handled which require setting the environment for a certain installation directory with additional modules and data files used by the main script. For instance, you can modify the Q path and set a `PROGDIR` variable to the installation directory as follows:

```
#!/usr/bin/q
#! -p/usr/share/myprog:
#! -cdef PROGDIR="/usr/share/myprog"
#! -cmain ARGS
```

By these means, all the necessary setup information is collected at the beginning of your main script where it can be changed easily at installation time.

Appendix C C Language Interface

The Q interpreter provides an interface to the C programming language, which allows you to extend the interpreter with your own collections of "built-ins" in C modules, in order to access functions in C libraries and to take advantage of the higher processing speed of C functions for time-critical applications. We also refer to such C modules interfacing to the Q interpreter as *external modules*. On most systems supporting "shared" libraries (a.k.a. "dll"s), external modules are loaded dynamically at runtime, otherwise they have to be linked statically with the interpreter's main program to create a new interpreter executable. The Q interpreter has its own external module support for MS Windows; on UNIX systems, it uses the `libtool` package to implement this functionality. See section "Shared library support for GNU" in *GNU Libtool*.

C.1 Compiling a Module

To provide for a platform-independent way of compiling and linking external modules, the Q programming system includes two tiny additional utilities which take care of the necessary and sometimes messy compilation details: `qcc` and `qld`.

`Qcc` is the module compiler. After you have prepared an external module (as explained in Section C.2 [Writing a Module], page 192), you run `qcc` to compile your module to a *library*, a binary file which can be loaded by or linked into the interpreter.

The `qcc` program will compile each given source file and then invoke the linker on all object files to produce the output library. The synopsis of the `qcc` program is as follows:

`qcc [`*options*`] [`*source-or-object-file*` ...] [-- `*cc-options*` ... [--link `*ld-options*`...]]`

As indicated, `qcc` can process both C source and object files, and extra options can be passed on to compiler and linker after the `--` option. All options understood by the `qcc` utility are listed below:

`--dry-run, -n`
: Only print compilation commands, do not actually execute them.

`--ext`
: Print the default output file extension and exit. This is usually '`.la`' on UNIX systems which denotes a `libtool` library, or '`.dll`' on Windows.

`--help, -h`
: Print a short help message and exit.

`--keep, -k`
: Do *not* delete intermediate files (object files and such) produced during the compilation process.

`--mingw`
: Select the mingw compiler driver. Mingw is the native Windows port of the well-known GNU C compiler, gcc. This option is the default for dll compilation on MS Windows.

`--msc`
: Select the MS Visual C compiler driver. Use this if you want to compile your module with the Visual C compiler on Windows.

`--output=`*file*`, -o` *file*
: Specify the name of the library output file.

`--verbose, -v`
: Echo compilation commands as they are processed.

`--version, -V`
: As with the other Q programming utilities, this option causes version number and copyright information to be displayed, after which the program terminates.

The Q module linker, `qld`, is implemented as a shell script (thus it is not supported on Windows) which simply invokes `libtool` with the specified `-dlopen` and `-dlpreopen` options to create a new interpreter executable. This is only necessary if your system does not support shared libraries. `Qld` is invoked as follows:

```
qld --help | --version | -o progname [-dlopen module ...]
```

The `--help` and `--version` options (which may also be abbreviated as `-h` and `-V`, respectively) work as usual. All other options are simply passed on to the `libtool` program, thus you can actually use any option recognized in libtool's "link" mode. (See section "Invoking libtool" in *GNU Libtool*, for details.) The interpreter executable is written to the specified output file *progname* (there is no default for the output file name, so the '`-o`' option must *always* be specified).

For a closer description of how `qld` is used see Section C.3 [Linking and Debugging a Module], page 197, below.

C.2 Writing a Module

Writing a real external module can be a complicated task, just like any bit of C programming which goes beyond mere exercising. We cannot cover this process in detail here, although we hopefully provide enough information to get you started. You should take a look at the external modules bundled with the Q distribution for more substantial examples.

The procedure for making a C function callable from the Q interpreter is fairly straightforward. You first need a Q script (called a *stub*) which declares the external C functions you would like to use in your Q program. Next you have to write a C module which implements these functions. Finally, you run `qcc` to translate the C module to a library file which can be loaded by or linked into the interpreter. We discuss each of these steps in turn.

As a running example, let us implement list reversal in C. Our C version of this function will be called `creverse`. We first create a Q script `creverse.q`, which declares the `creverse` function as `extern`:

```
// creverse stub
public extern creverse Xs;
```

This tells the interpreter that when the `creverse` function is applied to a single argument, it should invoke the corresponding C function, which is assumed to be found in a shared library named after the stub script.

To implement the `creverse` function, we create a C module `cerverse.c`. The Q programming system comes with an interface library called `libq` which provides the necessary operations to inspect Q expressions given as argument values, and to construct new expressions which can be returned as the result of the function invokation. The interface operations are declared in the `libq.h` header file which we include in our C module. The C module must provide the necessary code for being initialized by the interpreter, which is done by putting a `MODULE` "header" at the beginning of the source file. The `MODULE` macro takes one argument, the name of the module. The external functions are then declared with the `FUNCTION` macro, which takes four arguments: the name of the module, the name of the external function (as given in the stub script), the name of the argument count variable, and the name of the argument vector variable. The `FUNCTION` macro is followed by the C block giving the definition of the function. In our example, the `creverse.c` module contains the following C code:

```
#include <libq.h>

MODULE(creverse)

FUNCTION(creverse,creverse,argc,argv)
{
  /* to be sure, check number of arguments */
  if (argc == 1) {
    /* expr is the data type used by libq to represent Q expressions;
       x is set to the argument expression, y is initialized to the empty
       Q list (mknil value) */
    expr x = argv[0], y = mknil, hd, tl;
    /* iscons(x,...) checks that x is a [|] expression and returns its
       head element and tail list */
    while (y && iscons(x, &hd, &tl)) {
      /* use mkcons to prepend the head element to the list y constructed
         so far */
      expr z = mkcons(hd, y);
      y = z; x = tl;
    }
    if (!y)
      /* signal error condition */
      return __ERROR;
    else if (isnil(x))
      /* well-formed list argument, return the constructed reversal */
      return y;
    else {
      /* argument was not a well-formed list; throw away the constructed
         value and indicate failure */
      dispose(y);
      return __FAIL;
    }
  } else
    return __FAIL;
}
```

A description of the various macros and functions provided by the libq library can be found in the libq.h header file. We remark here that extern functions are treated very much like the built-in functions provided by the interpreter. The external definition may be thought of as an "external rule" being applied to a Q expression. The definition may either return a Q expression, in which case the rule applies, or __FAIL, in which case the rule fails, and the interpreter goes on to try other equations supplied by the programmer. As a third alternative, the external definition may also return __ERROR, which causes evaluation to be aborted with an error message. Also note that built-in functions always take priority. Thus the interpreter first checks for a built-in rule, then for an external definition, and finally considers the equations in the Q script. Therefore it is possible to override an equational function definition with an external function. For instance, we might rename the creverse function in the declaration in creverse.q to stdlib::reverse, and to reverse in creverse.c. This makes our definition override the definition of reverse in stdlib.q. The latter definition will then only be applied when the external definition fails. (This is what the clib module actually does to override the definition of reverse as well as other operations in stdlib.q, see Chapter 12 [Clib], page 119.)

We also remark that in order to prevent name clashes between external functions and ordinary C function names, the external function names are stropped with a special prefix (this is taken care of by the FUNCTION macro). To call an external function declared with FUNCTION from within your C module, you must therefore use the FUNCALL macro, which is invoked with the module name, function name and the argc and argv parameters as arguments. For instance, you would call the creverse function defined above as follows:

```
FUNCALL(creverse,creverse,argc,argv)
```

Back to our example. Before we can actually use the above function in the interpreter, we have to run qcc to translate the C module to a library object. In our example the process is fairly simple:

```
$ qcc creverse.c
```

If all went well, we now have a libtool library named creverse.la which is accompanied by some other libtool-generated files. (If you are trying this on a Windows system, you will see a dll file named creverse.dll instead.) If your system supports shared libraries then you can now simply run the creverse.q script with the Q interpreter as usual:

```
$ q creverse.q
```

(If at startup the interpreter complains that it could not load the module then your system probably does not support shared libraries. In this case you will have to create a new interpreter executable which links the module into the interpreter, as explained in Section C.3 [Linking and Debugging a Module], page 197.)

Let's try it: You probably want to compare the running time of our list reversal function against a Q version of the same operation, which can be defined as follows (include this in the creverse.q script):

```
public qreverse Xs;
qreverse Xs:List        = foldl push [] Xs;
```

The following results were obtained on an Intel PIII-800 PC running Linux:

```
==> def l = nums 1 50000

==> def r = creverse l; stats
0.033 secs, 1 reduction, 50000 cells

==> def r = qreverse l; stats
0.471 secs, 100002 reductions, 50004 cells
```

Not very surprisingly, the C function is indeed much faster, which is due to the extra pattern matching, value extraction and function call overhead of the interpreter.

It is also possible to declare a Q *type* as `extern`, and realize the type in C. Such a type must always be abstract (i.e., it must not have any constructor symbols, cf. Chapter 8 [Types], page 61), and the only way to get a value of such a type is by means of corresponding extern functions. For this purpose, the `libq` library provides the `mkobj` operation, which takes as its argument a pointer to the corresponding C object. For instance, consider an extern type `Bar` and a corresponding construction function `bar` declared as:

```
public extern type Bar;
public extern bar I J; // I and J integer
```

We might implement this type as a C struct containing a pair of integers as follows:

```
#include <libq.h>

MODULE(ctype)

typedef struct { long i, j; } Bar;

FUNCTION(ctype,bar,argc,argv)
{
  long i, j;
  if (argc != 2 || !isint(argv[0], &i) || !isint(argv[1], &j))
    return __FAIL;
  else {
    Bar *v = malloc(sizeof(Bar));
    expr x;
    if (!v) return __ERROR;
    v->i = i; v->j = j;
    return mkobj(type(Bar), v);
  }
}
```

Note that objects of an external type are completely "opaque" as far as the interpreter is concerned (just like file values). In particular, they will be printed using the notation <<*typeid*>> in the interpreter:

```
==> bar 1 2
<<Bar>>
```

The only way to access the contents of an external object is by means of corresponding C functions. For instance, an extern function which converts a `Bar` object to a tuple may be implemented as follows. The declaration:

```
public extern bartuple B;
```

The corresponding definition in the C module:

```
FUNCTION(ctype,bartuple,argc,argv)
{
  Bar *v;
  if (argc == 1 && isobj(argv[0], type(Bar), (void**)&v))
    return mktuplel(2, mkint(v->i), mkint(v->j));
  else
    return __FAIL;
}
```

Compile and load the script, using the same procedure as above, and try the following:

```
==> bartuple (bar 1 2)
(1,2)
```

(It is worth noting here that the `isint` and `mkint` functions used above only allow you to access and create integer values fitting into a machine integer. Integers of arbitrary sizes, which are represented as GMP `mpz_t` values, can be dealt with using the `ismpz` and `mkmpz` functions, see the `libq.h` header files for details.)

As indicated in our example, external objects are usually allocated dynamically, and will be freed automatically by the interpreter when they are no longer needed. This default behaviour is appropriate in most cases. However, you can also explicitly define a *destructor* function for objects of an external type in your C module as follows:

```
DESTRUCTOR(ctype,Bar,v)
{
  /* we could perform any other necessary cleanup here */
  free(v);
}
```

If such a destructor function is present, it will be used instead of the interpreter's default action to call `free()` when it disposes a Q expression of the corresponding type.

Occasionally, a module will also have some global internal data structures which have to be initialized and/or finalized. For this purpose, you can declare parameterless functions using the `INIT` and `FINI` macros, which will be executed before the script's initialization code, and just before the interpreter exits, respectively. Both macros take the module name as their single argument:

```
INIT(name)
{
  /* any code to be executed at startup goes here */
}

FINI(name)
{
```

```
  /* any code to be executed at termination goes here */
}
```

Note that the actual order in which the initialization/finalization routines of different modules are executed is unspecified. Furthermore, initialization routines will be executed before any of the script's initialization code (`def` and `undef`). Thus these routines should only be used to perform initializations and cleanup of private data structures of the module. Other initializations can be performed using appropriate `def` statements in the stub script.

C.3 Linking and Debugging a Module

If your system does not support shared libraries, or provides no means to dynamically load a shared library at runtime, you must run the `qld` program to produce a new interpreter executable which links in the required module. For instance, we could link the `creverse` module from the preceding section into the interpreter as follows:

```
$ qld -o myq -dlopen creverse.la
```

(You can also use the `-dlpreopen` option instead of `-dlopen` to force the module to be linked at load time, even if your system supports runtime loading. See section "Link mode" in *GNU Libtool*, for more information.)

You then run the script as usual, but using the custom interpreter executable built with `qld` instead of the standard '`q`' program:

```
$ ./myq creverse.q
```

Building such a "preloaded" interpreter is also required when you want to debug a module, in which case the module usually *must* be linked statically into the interpreter. You can do this as follows (assuming that your C compiler understands the `-g` debugging flag):

```
$ qcc creverse.c -- -g

$ qld -o myq -static -g -dlopen creverse.la
```

If you now let your debugger execute '`myq creverse.q`', you should be able to debug `creverse.c` at the source level.

Appendix D Debugging

The Q interpreter includes a simple symbolic debugger which can be used to trace the reductions performed during an expression evaluation. In order to use this tool successfully, you should be familiar with the way the Q interpreter evaluates expressions; see, in particular, Section 7.9 [Performing Reductions on a Stack], page 54.

The debugger is subject to activation when one of the following conditions arises:

- The interpreter has been invoked with the `-d` option, or the user has activated debugging using the `debug on` command (see Appendix B [Using Q], page 175).
- The interpreter executes the built-in `break` function on the right-hand side of a rule (see Section 10.6 [Exception Handling], page 86).
- The user interrupts an evaluation with `Ctl-C` (see Section 7.11 [Error Handling], page 59).
- A qualifying condition of a rule does not not evaluate to a truth value (cf. Section 7.7 [Conditional Rules], page 52).
- An exception is raised by the running script, either by using `throw` or through a trapped signal, see Section 10.6 [Exception Handling], page 86.

In the first case, the debugger is always invoked as soon as an evaluation is started. In the remaining cases the debugger is *only* invoked if the `break` flag is `on`. The `break` flag is an internal flag of the interpreter which can be controlled with the `break` command, cf. Section B.2 [Command Language], page 181. If `break` is `off` then all kinds of exceptions normally cause evaluation to be aborted immediately with an appropriate error message, and invokations of the `break` function are simply ignored. (The `break` flag is `off` by default, so you must set it to `on` before you can catch any break points with the debugger.) Also note that if there is a pending `catch` (see Section 10.6 [Exception Handling], page 86) then the debugger will only be invoked for a call to the `break` function, if the `break` flag is `on`.

When active, the debugger prints each reduction by a built-in or user-defined rule performed during evaluation in the following format:

`**` *left-hand-side* `==>` *right-hand-side*

"Tail reductions" (see Section 7.10 [Tail Recursion], page 57) are signaled by a leading '`++`' instead of '`**`', global and local variable bindings (cf. Section 7.3 [Free Variables], page 46, and Section 7.4 [Local Variables], page 47) with a leading `--`.

Whenever a new rule is activated, and also after processing a qualifier, the debugger interrupts the evaluation and displays the current rule together with the corresponding source file name and line number in the format:

level`>` *source-file*`, line` *line-number*`:` *left-hand-side* `==>` *right-hand-side qualifier*

(To remove clutter, the debugger only prints one qualifier at a time, namely the condition or local definition which is currently being processed.)

The level number printed in front of the rule indicates the position of the printed rule on the reduction stack (the topmost rule is at level 0). At the '`:`' prompt, you may then enter one of the following commands:

`?, help` Print a short help message which lists the debugger commands.

`options` Print a help message which briefly describes debugger options that can be set with the '`.`' command.

`?` *expr* Evaluate the given expression, with local variables bound to their current values.

`.` [*arg* ...]
Reprint the current rule (if invoked without arguments), or set debugger options.

`l` [*offs*] [*lines*]
List source lines of the current rule. You may specify the number of lines to be listed (the default is 1), as well as an offset taken relative to the line number of the current rule (this must be a signed number, with the default being `+0`).

`p` [*arg*] Print stacked rules.

`m` Print memory usage.

`v` List the local variables of the current rule.

`u` [*arg*]`, d` [*arg*]
Move up or down on the rule stack.

`t, b` Move to the top or bottom of the stack.

`<CR>` Step into the current reduction.

`n` Next: step over the current reduction.

`c` Continue: resume evaluation.

`h` Halt: abort evaluation.

`q, <EOF>` Quit: exit from the interpreter.

With the debugger being activated, you can simply keep on hitting the carriage return key to go through an evaluation step by step. The '`n`' ("next" a.k.a. "step over") command causes the interpreter to finish the current rule and also step over any tail reductions; it then leaves you in the context from which the current rule was invoked (which might be the interpreter's command prompt, if you just stepped over the toplevel rule).

The '`p`' command can be used to print the stack of active rules. The optional integer argument of this command sets the maximum number of stacked rules to print. The '`u`', '`d`', '`t`' and '`b`' commands let you move around on the reduction stack; '`u`' and '`d`' move up and down the given number of levels (default: 1), while '`t`' and '`b`' take you to the top (i.e., topmost rule) and bottom (active rule) of the stack, respectively.

The '`l`' command lists the source of the current rule. By default, only the line at which the rule starts is printed. You can specify the number of lines to be listed; e.g., '`l 5`' causes five lines to be printed. To have some context of the rule printed, you can also specify a signed integer denoting the relative offset from the source line; e.g., '`l -2 5`' causes five lines to be printed, starting two lines above the current rule. (Both the line count and the offset are remembered across different invocations of the '`l`' command.)

The 'v' command lists the local (i.e., "bound") variables of a rule. This comprises the variables which are bound by the left-hand side or have already been bound in a `where` clause while the rule is being processed. (Note that in the current implementation only those variables will be listed whose values are actually *used* somewhere on the right-hand side or in the qualifiers of the rule.)

The '?' command, when followed by an expression, causes the expression to be evaluated in the context of the current rule, i.e., local variables are bound to their current values. This lets you inspect the values of local variables, and perform arbitrary calculations with these values. Note that debugging mode is suspended while the expression is evaluated, so only the result (or an error message) will be printed.

At any point, you can use the 'c', 'h' and 'q' commands to continue evaluation without the debugger, abort an evaluation and return to the interpreter's prompt, or quit the interpreter, respectively. The 'm' command prints the current memory usage, i.e., the total number of allocated stack and heap expression cells, along with the corresponding numbers of cells which are actually in use. For the heap, it also prints the number of expression cells in the free list, i.e., temporarily unused expression cells below the current heap top. If limits are set on the maximum number of stack and heap cells, these figures are printed as well.

The current rule can be reprinted using the '.' command. Because Q expressions can get very large, the debugger usually only prints expression "outlines"; otherwise you could end up scrolling through pages and pages of printouts for each reduction and rule. The '.' command accepts some options which allow you to set the level of detail you want to see in the printed reductions and rules; these options will be remembered during an interpreter session. The options are generally specified in the format *option*=*value* and must not contain any whitespace (blanks or tabs). Multiple options can be specified with a single '.' command, separating different options with whitespace. Option values can also be specified in the same format on the interpreter's command line, using the `debug` command (see Section B.2 [Command Language], page 181), or with the interpreter's `--debug-options` option (see Section B.1 [Running Compiler and Interpreter], page 175). The following options are provided:

`. detail=`*n*
: Set the maximum depth (a.k.a. levels of nested parentheses, 2 by default) up to which expressions are displayed. The debugger will omit all subexpressions below the current level. Level 1 ('`. detail=1`') means to just print the toplevel expression, level 2 adds the first-level parentheses, level 3 all second-level parentheses, etc. All omitted parts in the expression outline will be represented using an ellipsis '`...`'. If you set the level to zero, or specify the symbolic value `all`, all expressions will be printed to arbitrary depth.

`. maxitems=`*n*
: Set the maximum number of elements to print for each list or tuple. The default is 3. A value of `0` or `all` causes all list and tuple elements to be printed.

`. maxchars=`*n*
: Set the maximum number of characters to print for each string value. The default is 33. A value of `0` or `all` causes all characters to be printed.

- `maxstack=`*n*
  Set the number of stack levels to print with the '`p`' command. The default is 6. A value of `0` or `all` causes the entire rule stack to be printed. This value is also modified when an argument is specified with '`p`'.
- `pathnames=y|n`
  Print full pathnames of scripts ('`y`' = yes, '`n`' = no). The default value of this option is '`n`', i.e., only the basenames of scripts are printed. Change this value to '`y`' if you want to see the complete path of a script file when a rule is printed.
- `options` Print the current settings. This option does not modify any settings, but simply prints all current option values on a single line.

Let us now take a look at a typical session with the debugger. For an example, we take the definition of the `fac` function from Section 2.3 [Writing a Script], page 12:

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

The commented session with the debugger follows.

```
==> fac 1                                                    evaluate fac 1
  0>  fac.q, line 1: fac 1  ==>  1*fac (1-1) if 1>0          try the first rule ...
(type ? for help)
: <CR>
**  1>0  ==>  true                                           evaluate the qualifier
  0>  fac.q, line 1: fac 1  ==>  1*fac (1-1)
: <CR>
**  1-1  ==>  0                                              evaluate 1-1 argument
  1>  fac.q, line 1: fac 0  ==>  0*fac (0-1) if 0>0          try the first rule on fac 0
: p
stack size: 2                                                now two rules are on the stack
  0>  fac.q, line 1: fac 1  ==>  1*fac (1-1)
  1>  fac.q, line 1: fac 0  ==>  0*fac (0-1) if 0>0
: <CR>
**  0>0  ==>  false                                          qualifier fails
  1>  fac.q, line 2: fac 0  ==>  1                           try the second rule ...
: <CR>
**  fac 0  ==>  1                                            bingo! reduce fac 0 to 1
  0>  fac.q, line 1: fac 1  ==>  1*fac (1-1)                 return to the suspended rule
: <CR>
**  1*1  ==>  1
**  fac 1  ==>  1                                            final reduction ...
1                                                            ... and the result printed by the
                                                             interpreter

==>
```

Appendix E Running Scripts in Emacs

The Q programming system also includes an Emacs Lisp program which lets you edit and run Q scripts in GNU Emacs or XEmacs. The program implements two major modes `q-mode` and `q-eval-mode` for Q source scripts and Q interpreter processes, respectively. The following discussion is rather terse and incomplete, but if you know Emacs then it should provide enough information to get you started.

To install Q mode on your system, copy the file `q-mode.el` from *prefix*`/share/q/etc` (where *prefix* is the installation prefix you selected at installation time) to your Emacs `site-lisp` directory and paste the following lines into your Emacs startup file:

```
(require 'q-mode)
(setq auto-mode-alist (cons '("\\.q$" . q-mode) auto-mode-alist))
```

If you want syntactic fontification (`font-lock` mode), you also have to add the following lines:

```
(add-hook 'q-mode-hook 'turn-on-font-lock)
(add-hook 'q-eval-mode-hook 'turn-on-font-lock)
```

More installation options are described at the beginning of the `q-mode.el` file. Once Q mode is installed and loaded, you can also customize it in Emacs; see the description of the `Customize` option below.

Q mode is used for editing Q scripts. With the startup configuration described above, it is invoked automatically on all files having the `.q` extension. Q mode provides auto-indentation and filling, as well as syntactic fontification of keywords, strings, comments, and variable and type symbols using the `font-lock` mode which is part of the Emacs library.

The indentation rules for Q scripts are somewhat unusual for a programming language, so we briefly summarize them here:

- Declarations start in the first column, with continuation lines being indented by a certain (configurable) amount. Moreover, initial '`=`', '`|`' and '`;`' symbols are all aligned in data type declarations.
- Inside rules, an initial '`=`' is aligned with the most recent equation. The default amount of '`=`' indentation can be configured. Auto indentation is also provided for the right-hand side and qualifier part of a rule. Qualifiers (introduced with `if`, `otherwise` and `where`) use an extra amount of indentation which can be configured as well.
- In addition, expressions are indented according to their parenthetical structure, as in Lisp mode.

As usual, indentation of a line is performed with the *`Tab`* key. There also is a menu command for indenting a selected region. Moreover, you can use *`Esc-Tab`* at the end of a line to move the cursor to the indentation position for the `=` symbol in an equation, and certain "electric" delimiter symbols like '`=`' will automatically perform indentation when typed at the beginning of a line.

The `q-run-script` command (usually bound to *`C-c C-c`*) compiles and runs the script in the current buffer; you can also use the `run-q` command (which has no keybinding by default) to invoke the interpreter without a script. Both commands create a new buffer named `*q-eval*` (if that buffer does not already exist) in which it runs the interpreter. The `*q-eval*` buffer is also used to display error messages from the compiler, if any. The interpreter is used in this buffer as usual, with the only difference that input and output is done through the buffer. Since this is an ordinary Emacs buffer, you can also save the buffer to some file to obtain a transcript of your interpreter session.

The `*q-eval*` buffer uses `q-eval-mode` as its major mode which is based on `comint-mode`, a generic command language interpreter mode which is also part of the Emacs library. In addition to the facilities provided by Comint mode (such as cycling through a history of recent commands), Q-Eval mode provides fontification of some special items (strings, comments and source file references printed by the interpreter), as well as some commands for locating the source lines referenced by compiler or debugger messages from the interpreter. In particular, pressing the return key (or the middle mouse button, when running under X-Windows) on such a message visits the corresponding source file with the cursor positioned at the line indicated by the message. You can also scan through the messages found in the buffer with the following commands:

`q-next-msg (C-c C-n), q-prev-msg (C-c C-p)`
: Show the next/previous compiler/debugger message and visit the corresponding source line in another window. An optional prefix argument may be used to specify the number of messages to advance.

`q-first-msg (C-c C-a), q-last-msg (C-c C-e)`
: Show the first/last line in a contiguous sequence of compiler/debugger messages above/below the current message and visit the corresponding source line.

The above commands can be invoked in both Q and Q-Eval mode, and they are also accessible from the corresponding Q and Q-Eval menus in the menubar (or the buffer popup menu obtained with the right mouse button). Some other commands are provided as well, but you can easily find out about these using the online help facilities of Emacs. In particular, you should try the `describe-mode` (*C-h m*) command which describes the currently active mode. Another useful command is `describe-key` (*C-h k*). You can also invoke the online version of this manual with *C-c C-h*.

The `Customize` option in the `Q`/`Q-Eval` menus can be used to set various options of the interpreter in the Q customization group, which belongs to Emacs' "Programming Languages" group. In particular, you should enable the `q-gnuclient` option in the "Options" subgroup which synchronizes Emacs with the Q interpreter; e.g., this option allows you to edit files and read online help using the interpreter's `edit` and `help` commands directly in Emacs. (For this to work, you must have the `gnuserv` package, which can be obtained from the usual elisp archives; see the `q-mode.el` file for details.)

References

H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

Adobe Systems Incorporated. *PostScript Language Reference Manual. Second Edition.* Addison-Wesley, Reading, Mass., 1990.

R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, New York, 1988.

N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. B: Formal Models and Semantics,* pages 243-320. North-Holland, Amsterdam, 1990.

A. Gräf. Left-to-right tree pattern matching. In *Proceedings Rewriting Techniques and Applications '91,* LNCS 488, pages 323-334. Springer, Berlin, 1991.

M.C. Henson. *Elements of Functional Languages.* Blackwell Scientific Publications, Oxford, 1987.

M.J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, Cambridge, Mass., 1985.

G.L. Steele, Jr. and G.J. Sussman. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory, 1975.

Index

.

. ... 185
.qexitrc 188
.qinitrc 187

@

@ 53

_

_ 42, 182

\

\" 21
\\ 21
\0 30
\b 21
\f 21
\n 21
\r 21
\t 21

A

abs 94, 109
abstract data type (ADT) 61
abstract type 49, 65
accept 145
access 129
acos 109
acosh 109
active 148
active rules 55
all 94
and 36
anonymous variable 42, 182
any 94
append 94, 99, 168
applicable equation 50
application 30
applicative order evaluation 52
apply 94
arc 113
arct 113
arg 109
ARGS 182
argument 30
argument list 26
arithmetic operators 35
arity 26
array 99
Array 99
array.q 99
array2 99
arrays 99
as 24
asctime 155
asin 109
asinh 109
assert 117
assert.q 117
assertions 117
associative arrays 102
atan 78
atan2 78
atanh 109
AVL tree 65
await 150

B

bag 101
Bag 101
bag.q 101
bags 101
basic expressions 29
batch mode 179
bcat 120
bcmp 120
bfloat 120
binary search tree 61
bind 144
binding clause 106
bint 120
bit operations 37, 77
bit shift functions 77
bitsets 77
bitwise logical operators 37
Bool 63
bound variable 41, 46
bounded_sem 151
bounding box (EPSF) 117
bread 135
break 86, 184

breakpoint 86
broadcast 150
bsize 120
bstr 120
bsub 120
built-in constants 30
built-in equations 50
built-in special forms 72
built-in types 63
bwrite 135
byte 120
byte order 122
bytecode 175
bytecode file 23
bytes 120
bytestr 120
ByteStr 120

C

C interface 191
C programming language 191
c_cc ATTR; 139
c_cflag 139
c_iflag 139
c_ispeed 139
c_lflag 139
c_oflag 139
c_ospeed 139
call by name 69
call by value 69
cancel 148
canceled 148
cat 94, 168
catch 86
cd 184
Char 63
character conversion 80
character sequences 20, 30
charpath 114
chars 96, 168
chdir 129, 184
chmod 129
chown 129
chr 80
clear 186
clib.q 94, 119
clip 114
clippath 113
clipping path 112
clock 156
close 134
closepath 113
closesocket 144
closing a file 84
closing a path 111
Color 115
combinatorial calculus 44
combinators 104
command 85
command language 181
command line arguments 175
commands 183
comment 19
comp.q 97
comparison functions (comp.q) 97
completion_matches 187
complex numbers (complex.q) 109
complex.q 109
compose 94
compound expressions 29
concatenation 38
concrete type 65
cond 150
Cond 150
cond.q 108
condition 41, 150
conditional clause 106
conditional equation 41
conditional expression 69
conj 109
connect 145
cons 94
const 26
constant symbol 26
constructors 43
container data structures 98
conversion functions 79
coordinate system 111
copies 116
copying 183
copypage 116
cos 78
cosh 109
creverse 192
critical section 150

crypt . . . 142
cst . . . 94
ctermid . . . 135
ctime . . . 155
currying . . . 31
curveto . . . 113

D

dash pattern . . . 115
data structure . . . 61
daylight . . . 155
debug . . . 184
debugger . . . 199
debugger commands . . . 199
debugger session . . . 202
debugger, activation of . . . 199
dec . . . 183
declarations . . . 25
declarations, consistency . . . 26
def . . . 181
default case . . . 41
default declaration . . . 26
delete . . . 101, 102
destructor function . . . 196
diagnostics . . . 117
dict . . . 101
Dict . . . 101
dict.q . . . 101
dictionaries . . . 101
dictionaries, hashed . . . 102
digit sequence . . . 20
disambiguating rules . . . 52
div . . . 35
do . . . 94
document structuring conventions . . . 116
double quotes . . . 20
drawing . . . 111
drop . . . 94
dropwhile . . . 94
DSC . . . 116
dup . . . 134
dup2 . . . 134

E

echo . . . 184
edit . . . 185
emacs . . . 203
empty list . . . 30
empty tuple . . . 30
emptyarray . . . 99
emptybag . . . 101
emptydict . . . 101
emptyhdict . . . 102
emptyheap . . . 100
emptyset . . . 100
encapsulated PostScript . . . 116
enumeration type . . . 63
eoclip . . . 114
eof . . . 81
eofill . . . 114
EPSF . . . 116
eq . . . 94
equality operator, at toplevel of equation . . . 43
equality, container data structures . . . 98
equality, lists . . . 97
equality, streams . . . 106
equality, truth values . . . 97
equality, tuples . . . 97
equation . . . 41
erasepage . . . 116
errno . . . 140
error . . . 117
ERROR . . . 182
error conditions . . . 59
error message . . . 117
error.q . . . 117
errorchecking_mutex . . . 150
escape sequences . . . 20
evaluation stack . . . 55
exception . . . 86
exceptions . . . 87
exec . . . 130
exists . . . 83
exit . . . 130
exitstatus . . . 131
exp . . . 78
exponential function . . . 78
expression . . . 29
expression value . . . 51
extern . . . 26, 192
extern symbol . . . 26

extern type . . . 195
external C function . . . 192
external rule . . . 193

F

fac . . . 12
fact . . . 167
factorial function . . . 12
factorial function, tail-recursive implementation . . . 57
fail . . . 86
fchdir . . . 134
fchmod . . . 134
fchown . . . 134
fclose . . . 81, 84
fcntl . . . 135
fcopy . . . 84, 135
fdatasync . . . 134
fdopen . . . 123
feof . . . 81
fflush . . . 81
fget . . . 123
fgetc . . . 124
fgets . . . 123
fib . . . 42, 58, 154
Fibonacci function . . . 42
Fibonacci function, hashed implementation . . . 154
Fibonacci function, tail-recursive implementation . . . 58
File . . . 63
file I/O . . . 83
file object . . . 83
file object, flushing output . . . 85
file object, unparsing . . . 84
file, binary . . . 83
filedev . . . 110
filename . . . 83
fileno . . . 123
files, binary . . . 85
fill . . . 114
filling . . . 111
filter . . . 94
first . . . 99, 100, 101, 102
fix . . . 183
flip . . . 90
float . . . 79
Float . . . 63
floating point number . . . 29
flush . . . 81
flushpage . . . 116
fnmatch . . . 156
foldl . . . 94
foldl1 . . . 94
foldr . . . 95
foldr1 . . . 95
font . . . 115
fopen . . . 81, 83, 123
force operator . . . 70
fork . . . 130
forkpty . . . 135
formal parameters . . . 41
fprintf . . . 125
fputc . . . 124
fputs . . . 124
frac . . . 79
fread . . . 81
freadc . . . 81
freadq . . . 81
freads . . . 81
free variable . . . 46
freopen . . . 123
fscanf . . . 125
fseek . . . 123
fst . . . 95
fstat . . . 134
fsync . . . 134
ftell . . . 123
ftruncate . . . 134
function . . . 30
function symbol . . . 19
fungetc . . . 123
fwrite . . . 81
fwritec . . . 81
fwriteq . . . 81
fwrites . . . 81

G

gcd . . . 167
get . . . 151, 153
get_bound . . . 151
get_size . . . 151
getc . . . 124
getcwd . . . 129
getegid . . . 132

getenv 131
geteuid 132
getgid 132
getgrent 141
getgrgid 141
getgrnam 141
getgroups 132
gethostbyaddr 142
gethostbyname 142
gethostent 142
gethostname 141
getlogin 132
getpeername 145
getpgid 132
getpgrp 132
getpid 130
getppid 130
getprotobyname 142
getprotobynumber 142
getprotoent 142
getpwent 141
getpwnam 141
getpwuid UID 141
gets 123
getsched 148
getservbyname 142
getservbyport 142
getservent 142
getsid 132
getsockname 145
getsockopt 145
getuid 132
Ghostscript 110
glob 156
gmtime 155
GNU emacs 203
gr_gid 141
gr_members 141
gr_name 141
gr_passwd 141
graphics 110
GRAPHICS 110
graphics state 112
graphics.q 110
grestore 114
gsave 114
gsdev 110
gvdev 110

H

h_addr_list 142
h_addr_type 142
h_aliases 142
h_name 142
halt 86
hash 80
hash codes 80
hashes 102
hd 95
hdict 102
HDict 102
hdict.q 102
hds 95
header 26
heap 100
Heap 100
heap.q 100
heaps 100
help 183
hex 183
higher-order functions 44
histfile 183
histsize 183

I

I/O functions 81
id 95
identifier 19
identifier, qualified 19, 23
ifelse 108
im 109
import 23, 185
import, cyclic 24
imports 186
in 36, 106
include 23
indexing 38
infix operator symbols 29
information hiding 61
inheritance 64
init 95
INPUT 84, 182
input and output 81
insert 99, 100, 101, 102
int 79
Int 63

integer 29
interactive mode 179
interrupt key 59
intsqrt 166
invmod 166
isactive 131
isalnum 119
isalpha 119
isarray 99
isascii 119
isatty 135
isbag 101
isbool 97
isbytestr 120
ischar 97
iscntrl 119
iscomplex 109
iscond 150
isconst 90
isdef 90
isdict 101
isdigit 119
isexcept 97
isexited 131
isfile 97
isfloat 97
isfun 90
isgraph 119
ishdict 102
isheap 100
isint 97
islist 97
islower 119
ismutex 150
isnum 97
isprime 166
isprint 119
ispunct 119
isref 153
issem 151
isset 100
issignaled 131
isspace 119
isspecial 90
isstopped 131
isstr 97
isstream 106
issym 97
isthread 147
istuple 97
isupper 119
isvar 90
isxdigit 119
iter 95
iterate 107
iterative algorithm 58

J

jacobi 167
join 96, 168

K

keys 102
kill 130

L

lambda 103
lambda calculus 44, 103
lambda.q 103
last 95, 99, 101, 102
lcm 167
left-hand side 41
leftmost-innermost evaluation 52
lexicographic order, lists 97
lexicographic order, strings 35
lg 109
libq library 192
libq, DESTRUCTOR macro 196
libq, FINI macro 196
libq, FUNCALL macro 193
libq, FUNCTION macro 192
libq, INIT macro 196
libq, module initialization and finalization 196
libq, MODULE macro 192
libq.h header file 192
lineto 113
link 129
list 32
list 80, 99, 100, 101, 102, 106
List 63
list comprehensions 106
list functions 79
list operators 38

list.q 106
list2 99
listen 145
listof 106
literal expression 73
ln 78
load 186
local variable 47
localtime 155
lock 150
log 109
logarithm 78
logical operators 36
lpdev 110
ls 85, 185
lseek 135
lstat 129

M

map 95
match 108
matchp 108
math.q 109
mathematical functions 78
mathematical functions (math.q) 109
max 95
member 101, 102
members 99, 100, 101, 102
members2 99
memsize 183
mergesort 97
min 95
mkarray 99
mkarray2 99
mkdict 101
mkdir 129
mkfifo 129
mkhdict 102
mklist 95, 168
mkstream 107
mktime 155
mod 35
mode argument (fopen) 83
mode argument (popen) 85
modifier 26
module 23
module name 24
modules 186
more 86
moveto 113
msort 97
multiple declarations 26
multisets 101
multithreading 147
mutex 150
Mutex 150

N

namespace 23
narc 113
neg 95
negative numbers 34
neq 95
newline character 21
newpath 113
Newton's algorithm 13
nice 130
non-linear equation 45
non-special argument 71
normal form 51
not 36
null 95, 99, 100, 101, 102
null character 30
nulldev 110
Num 63
numeric constants 20
numeric constants, range and precision 29
numeric functions 78
nums 95, 168
numsby 95, 168
numstream 106
numstreamby 107

O

oct 183
open 134
openpty 135
operator associativity 34
operator precedence 33
operator section 34
operator symbols 33
operator, as prefix function 34
operators, on left-hand side of equation 43

options 177
or 36
ord 80
ordinal number 80
otherwise 41
OUTPUT 84, 182

P

p_aliases 142
p_name 142
p_proto 142
page description 110
painting 111
pair 95
parsing expressions 80
path 111
path 183
path search 91
pattern 42
pause 130
perror 140
pipe 85
pipe 134
point unit 111
polymorphic operation 65
pop 95
popen 81
pos 79
POSIX threads 147
post 151
PostScript 110
pow 166
powmod 166
prd 95
pred 77, 96
predefined type symbols 27, 49
prefix operator symbols 29
prelude 24, 175, 187
prelude.q 24, 93, 175, 187
primary expression 34
primitive operation 50
printf 125
priority declaration 53
priority queues 100
private 25
private symbol 25
prompt 183
ps 116
psfile 116
psheader 116
psstr 116
public 25
public symbol 25
push 95
put 153
putc 124
puts 124
pw_dir 141
pw_gecos 141
pw_gid 141
pw_name 141
pw_passwd 141
pw_shell 141
pw_uid 141
pwd 184

Q

q 175
q.out 176
qc 176
qcc 191
qexitrc 188
qinitrc 187
qld 192
QPATH 175
qsort 98
qualifier 41
quicksort 97
quit 86, 175
quotation operators 35, 74
quote operator 73

R

raise 130
random 78
random numbers 78
rcurveto 113
re 109
read 81
readc 81
readdir 129
readlink 129
readq 81

reads 81
realtime thread 148
recursive definition 57
recursive evaluation 55
recursive_mutex 150
recv 145
recvfrom 145
redeclaration 26
redex 51
reduction 50
reduction stack 55
reduction strategy 51
reductions, tracing of 199
reexport 26
ref 153
Ref 153
reg 159
regdone 158
regend 159
regex 157
regex_next 157
regmatch 158
regnext 158
regpos 159
regs 159
regskip 159
regstart 159
regular expressions 157
relational operators 35
rem 167
rename 129
reserved words 20
restorematrix 114
result 148
return 148
reverse 95, 168, 192
rewriting model 55
right-hand side 41
rlineto 113
rmdir 129
rmfirst 99, 100, 101, 102
rmlast 99, 101, 102
rmoveto 113
root 166
rotate 115
round 79
rule 41
rule priority 53
run 185
runtime error 59

S

s_aliases 142
s_name 142
s_port 142
s_proto 142
save 186
savematrix 114
scale 115
scaling factor 20
scan 95
scan1 95
scanf 125
scheduling 148
sci 183
scope 25
scope, symbol in type declaration 27
scope-modifier prefix 26
script 23
search path 175
search tree operations 62
seed 78
select 137
sem 151
Sem 151
semaphore 151
send 145
sendto 145
sequence operator 39
sequence operator, used with I/O operations 83
set 100
Set 100
set.q 100
set/bag comparison 98
set/bag difference 98
set/bag intersection 98
set/bag union 98
setcmykcolor 115
setcolor 115
setdash 115
setegid 132
setenv 131
seterrno 140
seteuid 132
setfont 115

setgid 132
setgray 115
setgroups 132
sethsbcolor 115
setlinecap 115
setlinejoin 115
setlinewidth 115
setpgid 132
setpgrp 132
setregid 132
setreuid 132
setrgbcolor 115
sets 100
setsched 148
setsid 132
setsockopt 145
setuid 132
setvbuf 123
sgn 95
shared library 191
shell escape 182
shl 77
short circuit evaluation 36
short-circuit mode 72
show 114
showpage 116
shr 77
shutdown 144
side-effects, in I/O operations 81
sieve of Erathosthenes 107
signal 90
signal 150
signals, in multithreaded scripts 151
sin 78
sinh 109
size operator 38
size operator, container data structures 98
sleep 90
slice 79
snd 95
socket 144
socketpair 144
sort 169
sort.q 97
sorting algorithms 97
source 185
spawn 130
special 26
special argument 69
special case rule 52
special constructor 72
special form 69
splice operator 73
split 97, 168
sprintf 125
sqrt 78
square root 78
sscanf 125
st_atime 130
st_ctime 130
st_dev 130
st_gid 130
st_ino 130
st_mode 130
st_mtime 130
st_nlink 130
st_rdev 130
st_size 130
st_uid 130
stacksize 183
standard error device 182
standard functions 94
standard input/output devices 182
standard library scripts 93
standard types 98
stat 129
state variables 57
stats 184
std 183
stddecl.q 98
stddecls.q 93
stdlib.q 93, 94
stdtypes.q 93, 98
stopsig 131
str 80
strcat 97, 168
stream 72
stream 106
Stream 106
stream comprehensions 107
stream.q 106
streamcat 107
streamof 107
streams 106
streams, overloaded list operations 106
strerror 140

strftime 155
string 30
String 63
string comparison 35
string constants 20
string functions 79
string functions (string.q) 96
string operators 38
string.q 96
stroke 114
strq 80
stub 192
sub 79
subpath 111
subscript operator 38
subsequence 79
substr 79
subtype 64
succ 77, 96
sum 96
supertype 64
switch 108
symlink 129
syntactic equality 45
syntactic identity 45
sysinfo 90
system 130
system information 90

T

tab character 21
tail call elimination 58
tail recursion 57
tail recursion optimization 58
tail recursion, with sequences 59
tail reduction 58
tail-recursive programming 57
take 96
takewhile 96
tan 109
tanh 109
task 147
tcdrain 139
tcflow 139
tcflush 139
tcgetattr 139
tcgetpgrp 139
tcsendbreak 139
tcsetattr 139
tcsetpgrp 139
terminal I/O 82
terminal input 82
terminal output 82
termsig 131
text output (graphics) 112
this_thread 147
thread 147
thread 148
Thread 147
thread cancellation 148
thread handle 147
thread_no 147
throw 86
time 90
times 156
timezone 155
timing functions 91
tl 96
tls 96
tm_day 155
tm_hour 155
tm_isdst 155
tm_min 155
tm_month 155
tm_sec 155
tm_wday 155
tm_yday 155
tm_year 155
tmpfile 123
tmpnam 123
tolower 119
top 96
toupper 119
translate 115
transpose 96
trap 86
trd 96
trigonometric functions 78
triple 96
trunc 79
truncate 129
truth values 30
try 150, 151
ttyname 135
tuple 32

tuple 80
Tuple 63
tuple functions 79
tuple operators 38
tuplecat 96, 168
tuples and parenthesized expressions 34
type 61
type declaration 27, 61
type guard 49, 61
type hierarchy 49
type identifier 27
type-checking predicates (typec.q) 97
typec.q 97
tzname 155

U

umask 129
un_machine 141
un_nodename 141
un_release 141
un_sysname 141
un_version 141
uname 141
unary minus 34
undef 181
ungetc 123
unit length 111
universal quantification 41
unlink 129
unlock 150
unparsing expressions 80
until 96
unzip 96
unzip3 96
update 100, 102
update2 100
utime 129

V

val 80
valq 80
vals 102
var 26
variable bindings 55
variable definition 46
variable definition, local 41
variable symbol 19
variable symbol, bound 30
variable symbol, free 26, 30
variable, multiple occurrence on left-hand side 45
variadic functions 31
version 90
version information 90

W

wait 130
waitpid 130
where 47
which 90, 185
while 96
who 186
whos 186
write 81
writec 81
writeq 81
writes 81

X

XEmacs 203

Y

yield 148

Z

zip 96
zip3 96
zipwith 96
zipwith3 96

Table of Contents

1 Introduction **1**

2 Getting Started **5**

- 2.1 Using the Interpreter 5
- 2.2 Using the Standard Library 8
- 2.3 Writing a Script 12
- 2.4 Definitions 14
- 2.5 Runtime Errors 16

3 Lexical Matters **19**

4 Scripts and Modules **23**

5 Declarations **25**

6 Expressions **29**

- 6.1 Constants and Variables 29
- 6.2 Applications 30
- 6.3 Lists and Tuples 32
- 6.4 Built-In Operators 33
  - 6.4.1 Quotation Operators 35
  - 6.4.2 Arithmetic Operators 35
  - 6.4.3 Relational Operators 35
  - 6.4.4 Logical and Bit Operators 36
  - 6.4.5 String/List/Tuple Operators 38
  - 6.4.6 The Sequence Operator 39

7 Equations and Expression Evaluation **41**

- 7.1 Equations 41
- 7.2 Non-Linear Equations 45
- 7.3 Free Variables 46
- 7.4 Local Variables 47
- 7.5 Type Guards 49
- 7.6 Normal Forms and Reduction Strategy 50
- 7.7 Conditional Rules 52
- 7.8 Rule Priorities 53
- 7.9 Performing Reductions on a Stack 54
- 7.10 Tail Recursion 57
- 7.11 Error Handling 59

8 Types 61
8.1 Using Type Guards 61
8.2 Built-In and Enumeration Types 63
8.3 Sub- and Supertypes 64

9 Special Forms 69
9.1 Basic Concepts 69
9.2 Special Constructors 72
9.3 Built-In Special Forms 72
9.4 The Quote Operator 73

10 Built-In Functions 77
10.1 Arithmetic Functions 77
10.2 Numeric Functions 78
10.3 String/List/Tuple Functions 79
10.4 Conversion Functions 79
10.5 I/O Functions 81
10.5.1 Terminal I/O 82
10.5.2 File I/O 83
10.5.3 Pipes 85
10.6 Exception Handling 86
10.7 Miscellaneous Functions 90

11 The Standard Library 93
11.1 Standard Functions 94
11.2 String Functions 96
11.3 Comparison Functions 97
11.4 Type-Checking Predicates 97
11.5 Sorting Algorithms 97
11.6 Standard Types 98
11.6.1 Arrays 99
11.6.2 Heaps 100
11.6.3 Sets 100
11.6.4 Bags 101
11.6.5 Dictionaries 101
11.6.6 Hashed Dictionaries 102
11.7 Lambda Calculus 103
11.8 List Comprehensions 106
11.9 Streams 106
11.10 Conditional Expressions 108
11.11 Mathematical Functions 109
11.12 Complex Numbers 109
11.13 Graphics 110
11.13.1 Coordinate System 111
11.13.2 Overview of Graphics Operations 111
11.13.3 Path Construction 113
11.13.4 Painting 114

11.13.5 Clipping 114
11.13.6 Graphics State 114
11.13.7 Miscellaneous Operations 116
11.13.8 DSC and EPSF Comments 116
11.14 Diagnostics and Error Messages 117

12 Clib 119

12.1 Manifest Constants 119
12.2 Additional String Functions 119
12.3 Byte Strings 120
12.4 Extended File Functions 123
12.5 C-Style Formatted I/O 125
12.6 File and Directory Functions 129
12.7 Process Control 130
12.8 Low-Level I/O 134
12.9 Terminal Operations 139
12.10 System Information 140
12.11 Sockets 143
12.12 POSIX Threads 147
12.12.1 Thread Creation and Management 147
12.12.2 Realtime Scheduling 148
12.12.3 Mutexes 150
12.12.4 Conditions 150
12.12.5 Semaphores 151
12.12.6 Threads and Signals 151
12.12.7 Thread Examples 152
12.13 Expression References 153
12.14 Time Functions 154
12.15 Filename Globbing 156
12.16 Regular Expression Matching 157
12.16.1 High-Level Interface 157
12.16.2 Low-Level Interface 158
12.16.3 Match State Information 159
12.16.4 Basic Examples 160
12.16.5 Empty and Overlapping Matches 162
12.16.6 Splitting 162
12.16.7 Performing Replacements 163
12.16.8 Submatches 164
12.16.9 Nested Searches 165
12.17 Additional Integer Functions 165
12.17.1 Powers and Roots 166
12.17.2 Prime Test 166
12.17.3 Other Number-Theoretic Functions 167
12.17.4 Examples 167
12.18 C Replacements for Common Standard Library Functions 168

Appendix A Q Language Grammar **171**

Appendix B Using Q **175**
B.1 Running Compiler and Interpreter 175
B.2 Command Language 181
B.3 Setting up your Environment 187
B.4 Running Scripts from the Shell 188

Appendix C C Language Interface **191**
C.1 Compiling a Module 191
C.2 Writing a Module...................................... 192
C.3 Linking and Debugging a Module 197

Appendix D Debugging **199**

Appendix E Running Scripts in Emacs **203**

References **205**

Index **207**