

# Q in a Nutshell

Albert Gräf<sup>1</sup>  
Department of Music Informatics  
Johannes Gutenberg University  
Mainz, Germany

June 30, 2007

<sup>1</sup>Email: [Dr.Graef@t-online.de](mailto:Dr.Graef@t-online.de). This is draft version 0.5 for Q version 7.7. Copyright © 2007 by Albert Gräf. Permission is granted to make and distribute verbatim copies of this document provided the copyright notice and this permission notice are preserved on all copies.

# Preface

The purpose of this tutorial is to provide computer scientists, mathematicians, programmers, software engineers, students, teachers and anyone else interested in functional programming with an informal introduction to the Q programming language, which can be used to get a glance of the main ideas behind the language and its major features as quickly and painlessly as possible.

So what is Q? In a nutshell, it is a kind of functional programming language with a syntax similar to Miranda and Haskell, but with general term rewriting instead of the lambda calculus as the underlying computational model. It is also one of my pet projects, on which I have been working, on and off, for almost sixteen years now. Q has some fairly unique features, in particular its user-definable special forms and algebraic types with inheritance, which makes it a somewhat exotic but powerful member of the FP language zoo. Q is also supposed to be a practical language (hence all the interfaces to third-party libraries and software), so I hope that some people may find it as useful as I do.

## Acknowledgements

Q surely wouldn't exist without the support of many kind people who provided help in various stages of the project. Special thanks are due to Frank Drewes and Klaus Barthelmann who helped to get it all started, John Cowan for his work on the Chicken interface and helping with Q's Unicode support, Rob Hubbard who wrote Q's rational number and polynomial libraries, Andrew Berg for his work on an SDL interface (yet to be released), Jonas Joebges and Milen Velikov who did most of the OpenGL, OpenAL and Xine interfaces, and Yann Orlarey for his support and many interesting discussions about how functional programming can be applied in the realm of computer music. Last but not least, I'd also like to thank John Cowan, Rob Hubbard and Keith Trenton for their thorough proofreading (of course I'm the one to blame for all remaining typos and other errors).

Albert Gräf  
Mainz, June 2007

# Contents

<b>Preface</b>	<b>1</b>
<b>1 Introduction</b>	<b>4</b>
<b>2 The Language</b>	<b>5</b>
2.1 Expressions, Equations and Rewriting . . . . .	6
2.2 Conditional Equations . . . . .	11
2.3 Variable Definitions . . . . .	12
2.4 Recursion . . . . .	15
2.5 Pattern Matching . . . . .	16
2.6 Higher-Order Functions . . . . .	17
2.7 Symbolic Rewriting . . . . .	18
2.8 Special Forms and Lazy Evaluation . . . . .	20
2.9 Metaprogramming . . . . .	22
2.10 Algebraic Types . . . . .	25
2.11 Sub- and Supertypes . . . . .	27
2.12 Abstract Types and Views . . . . .	28
2.13 Enumeration Types . . . . .	30
2.14 Imperative Features . . . . .	31
2.15 Modules and Namespaces . . . . .	33
<b>3 The Library</b>	<b>34</b>
3.1 Core Libraries . . . . .	34
3.2 Multimedia Library . . . . .	35
<b>4 The Implementation</b>	<b>37</b>
4.1 The Q Virtual Machine . . . . .	37
4.2 QVM Bytecode Example . . . . .	40
<b>5 Examples</b>	<b>43</b>
5.1 Basic Examples . . . . .	43
5.2 Search Trees . . . . .	46
5.3 Huffman Trees . . . . .	49

5.4	Streams . . . . .	52
5.5	Eight Queens . . . . .	55
5.6	Fixed Points . . . . .	57
5.7	Symbolic Rewriting . . . . .	58
5.8	System Programming . . . . .	59
<b>6</b>	<b>Conclusion</b>	<b>61</b>
	<b>Bibliography</b>	<b>63</b>

# Chapter 1

## Introduction

Q, the “equational programming language”, was born at some time in 1991 as an experiment to show that a general-purpose language based on the term rewriting calculus would be useful as a practical programming language. Since its humble beginnings as a toy language, which mainly served as a testbed for the author’s ideas about efficient term pattern matching [7], Q has evolved to a point where the initial kinks have been ironed out and the language is now considered stable. Q works on a variety of operating systems, including Linux, OS X and Windows, and should be portable to other POSIX-compatible systems with moderate effort. Adequate programming tools are available, including an interpreter with integrated symbolic debugger, an Emacs mode featuring syntactic fontification, which also lets you run Q scripts right inside the editor, and a graphical front end to the interpreter for all recent Windows versions. Q also offers a fairly comprehensive library, which makes it useful as a kind of “functional scripting language” in many application areas. After its move to SourceForge in December 2003, the Q project has attracted a growing user community and people have contributed to the project through source code, ideas, discussions, feature requests, bug reports and general support.

The rest of the tutorial is organized into several chapters dealing with the most important aspects of the language, its library and its implementation, respectively, followed by a chapter with a few typical programming examples. We hope that you will find this material informative, or at least entertaining, and maybe helpful to get a first impression of Q and decide whether you might want to give it a try. Please note that this document is *not* a general introduction to functional programming; it will be helpful if you already have at least a passing familiarity with this style of programming (see, e.g., [11], or [22] if you’re short on time). For more details on the Q language, you may consult the documentation at <http://q-lang.sf.net>. There you can also find the Q interpreter and additional related software as free downloads (distributed under the GNU Public License).

# Chapter 2

## The Language

At its core, Q is a fairly simple language which is based entirely on the notions of *reductions* and *normal forms* pertaining to the term rewriting calculus. A Q program or *script* is simply a collection of equations, which are interpreted as rewriting rules used to reduce expressions to normal form.

Languages based on term rewriting and equational semantics were a fashionable research topic during the 1980s and the beginning of the 1990s, two notable examples being the OBJ family of languages and OPAL [6, 5]. However, most of these languages were designed for specific applications such as algebraic specification, theorem proving, computer algebra and program transformation; a recent example of the latter kind is Stratego [4]. Practical systems for general-purpose programming like Aardappel [20] and Q are still more the exception than the rule, at least if we don't count modern functional languages such as Clean, Haskell, Miranda and ML [17, 12, 19, 13], which also employ equational logic and rewriting for reasoning about programs, but are in fact based on the lambda calculus, which is just one very specific rewriting system.

While the term rewriting semantics of Q were largely inspired by O'Donnell's pioneering work on equational programming [14], the syntax of the language was based on the original edition of Bird and Wadler's book [3] and thus bears more similarities to Miranda and Haskell (a notable difference being that Q is free-format instead of layout-based). Like other modern-style functional languages, Q has curried function applications, equational function definitions, algebraic types and a module system. The interpreter provides a reasonably efficient implementation of term rewriting, featuring automatic memory management, tail call optimization and both eager and lazy evaluation strategies. Q is an "impure" functional language, i.e., it supports operations with side-effects such as imperative I/O, mutable data, exception handling and multithreading. Moreover, Q also offers some features that make it more attractive to Lisp programmers, in particular: dynamic typing and metaprogramming capabilities. In a sense Q bridges the gap between modern-style and traditional functional languages, and adds in extensive symbolic computing capabilities which constitute Q's heritage from the term rewriting calculus.

In the following, we take you through a rather informal tour of Q's most important features. A much more detailed description of the language can be found in [8].

## 2.1 Expressions, Equations and Rewriting

Let's start out with a trivial example. Here is how you define a function `square` which squares its argument by multiplying it by itself:

```
square X = X*X;
```

As indicated, the simplest form of an equation just consists of two expressions, the left-hand side and the right-hand side of the equation, separated by a '=' symbol.<sup>1</sup> Capitalized identifiers are used to indicate the variables in an equation, which are bound to the actual values when an equation is applied. This convention, which was borrowed from Prolog, ensures that the interpreter can distinguish between function and variable symbols even without any explicit declarations.

Equations are always applied from left to right. In this process the variables occurring on the left-hand side are bound to their actual values in the target expression to be evaluated, and the right-hand side is then evaluated in the context of these variable bindings. Such a single rewriting step is called a *reduction* and written  $L \Rightarrow R$ , where the involved instances  $L$  and  $R$  of both sides of the equation are also called the *redex* and the *reduct*, respectively. For instance, when the above equation is applied to evaluate an expression like `square 2`, the interpreter performs the following reduction sequence by first invoking the user-defined rule for `square` (employing `square 2` as the redex, which yields `2*2` as the corresponding reduct) and then the built-in rule for integer multiplication:

```
square 2  $\Rightarrow$  2*2  $\Rightarrow$  4
```

The resulting expression `4` is in *normal form* (meaning that no more equations or built-in rules can be applied to it) and denotes the “value” of the original expression `square 2`. Thus *evaluation* of an expression in Q simply means that the interpreter keeps rewriting the target expression to be evaluated, applying equations as well as built-in rules implementing the primitive operations such as arithmetic, until it reaches a normal form. Expressions are normally evaluated from left to right, innermost expressions first, so that the value of an expression like `square (2+3)` is computed as follows:

```
square (2+3)  $\Rightarrow$  square 5  $\Rightarrow$  5*5  $\Rightarrow$  25
```

This is also known as “call by value” or “eager” evaluation. As will be discussed later, the Q language also provides so-called “special forms” as a means to change the default evaluation order.

---

<sup>1</sup>Also note the mandatory ';' delimiter at the end of equations (and declarations). This is necessary because Q, unlike layout-based languages like Haskell, is a *free format* language in which whitespace is not significant, except if it serves as a delimiter. The same applies to (C/C++ style) comments.

## Expression Syntax

A few words about Q’s expression syntax are in order here. The basic expression types are summarized in Figure 2.1. All the “usual suspects” are there: *numbers*, which are either multiprecision integers or double precision floating point values; *strings*, which are internally encoded in UTF-8, providing full Unicode support while retaining backward compatibility with 7 bit ASCII; *lists*, *streams* and *tuples*, which provide different means to represent sequences of expressions; function, variable and operator *symbols*, consisting of arbitrary alphanumeric and punctuation Unicode characters, respectively; and, last but not least, function and operator *applications*.

Note that, unlike Haskell, Q provides both *eager* and *lazy* (or, more precisely, *call-by-value* and *call-by-name*) lists. Following common terminology, these are simply called *lists* and *streams* in Q. Both lists and streams use the customary right-recursive representation, whereas *tuples* are internally encoded as vectors permitting constant-time access to their elements. Lists may also be written using the Prolog-like syntax `[X|Xs]`, denoting a list with head element `X` and tail (list of remaining elements) `Xs` (likewise for streams and tuples, using curly braces and parentheses instead of brackets). Also note that, as in Python, trailing commas are generally permitted at the end of a list, stream or tuple, and the special case of a 1-tuple (which doesn’t exist in Haskell) *must* be denoted with a trailing comma to distinguish it from a parenthesized expression. There is also a special *grouping syntax* that allows lists (as well as streams and tuples) of tuples to be written without extra parentheses, as in `[a,1;b,2;...]`, which provides a convenient notation, e.g., for lists of key-value pairs.

As will be explained in much more detail later, Q doesn’t actually distinguish between “defined” and “constructor” function symbols, and thus a symbol like `foo` may act as *both* a function and a constructor, depending on the context. Q provides a few built-in function symbols which are in fact constants and hence stand for themselves, such as `false` and `true`, which denote the truth values.

Like most modern functional languages, Q has curried function applications, written simply as juxtaposition, as in `max 2 3`, which denotes the application of a function `max` to two parameters `2` and `3`. This makes it possible to have partial function applications like `max 0`, which again denotes a function, namely the function that computes the maximum of `0` and the next parameter. Parentheses are used to group expressions and to denote tuples, but are *not* part of the function application syntax. Hence an expression like `foo(X,Y)` denotes an application of the `foo` function, but to *one* structured argument `(X,Y)` rather than two separate arguments.

For convenience, the Q language also supports the usual infix notation for operators such as `+` and `*`. As in other modern functional languages, these are syntactic sugar for function applications, and enclosing an operator in parentheses turns it into an ordinary prefix function. For example, `X*X` is a shorthand for the function application `(*) X X`. So-called operator *sections*, which are just special syntax for partial operator applications, are also supported; e.g., `(+1)` denotes the function that adds 1 to its argument, `(1/)` the reciprocal function.

Simple expressions		Compound expressions	
Type	Examples	Type	Examples
Integer (decimal) (hexadecimal) (octal)	12345678 0xa0 033	List	[] [a] [a,b,c]
Float	0. -1.0 1.2345e-78	Stream	{} {a} {a,b,c}
String	" " "abc" "Hello!\n"	Tuple	() (a,) (a,b,c)
Symbol (function) (variable) (operator)	foo BAR (+)	Application	sin 0.5 map (*2) Xs X+Y

Figure 2.1: Expression types.

Group	Operators	Meaning	Example
Quotation (unary)	' ' ~ &	quote, splice, force, memoize	'X 'X ~X &X
Application		function application	X Y
Composition	.	function composition	X.Y
Script (right-associative)	^ !	exponentiation indexing	X^Y X!Y
Prefix (unary)	- # not	unary minus length/size logical/bitwise negation	-X #X not X
Multiplication	* / div mod and and then	multiplication division integer division remainder of integer division logical/bitwise conjunction short-circuit logical conjunction	X*Y X/Y X div Y X mod Y X and Y X and then Y
Addition	+ - ++ or or else	addition subtraction concatenation logical/bitwise disjunction short-circuit logical disjunction	X+Y X-Y X++Y X or Y X or else Y
Relations (non-associative)	< > <= >= = <> ==	less than greater than less than or equal greater than or equal equal not equal syntactic equality	X<Y X>Y X<=Y X>=Y X=Y X<>Y X==Y
Infix application (right-associative)	\$	function application	X\$Y
Sequence		execute expressions in sequence	X  Y

Figure 2.2: Built-in operators.

Figure 2.2 shows the built-in operators, in order of decreasing precedence. All binary operators are left-associative unless explicitly mentioned otherwise. In addition, Q provides two predefined mixfix operators for conditional expressions and lambda abstractions, using the customary syntax ‘`if X then Y else Z`’ and ‘`\X.Y`’, respectively. These are at lower precedence than all the other operators (except sequences, which take precedence over conditional expressions). Likewise, the usual syntactic sugar is provided for list and stream *enumerations* and *comprehensions*, although the latter use a slightly different syntax than Haskell. More precisely, the notations `[A..B]` and `[A,B..C]` indicate arithmetic sequences and lists of consecutive elements of “enumeration type” members, see Section 2.13, and syntax like `[2*X+Y : (X,Y) in Zs, X>0]` can be used as a shorthand to construct lists by iterating over other lists and filtering elements with predicates. Similar syntax is provided for streams, using curly braces instead of brackets, see Section 2.8. All of these are just syntactic sugar for built-in or standard library functions which are actually implemented as so-called “special forms”; we will have to say more about that later.

Note that the operator system, if you look at the the precedence levels between the unary prefix and the relational operators, has many similarities with that of Pascal, which arguably has its deficiencies. In particular, relations have to be parenthesized when combined with logical connectives. However, this system was chosen because it is simple enough so that operator precedences can be remembered easily. Also, the ‘=’ and ‘-’ symbols play somewhat awkward double roles in the syntax. The former is used both as an operator and a delimiter to separate the left-hand and right-hand side of an equation, while the latter is used to denote both unary and binary minus. Finally, note that Figure 2.2 only lists the *built-in* operators. It is also possible to declare your own prefix or infix operator symbols (e.g., the relational ‘`in`’ operator used in list and stream comprehensions is not a builtin, but declared in the standard library). For the sake of brevity, we omit the discussion of these technicalities here; please refer to [8, Section 6] for details.

## Using the Interpreter

Using Q is supposed to be simple; essentially you just throw together some equations, start the interpreter and type in the expressions you want to evaluate. You can run the Q interpreter with the ‘`q`’ command, optionally specifying a source script containing your equations on the command line, e.g.:

```
ag@obelix> q square.q
```

The interpreter provides an interactive read-eval-print loop which takes the expressions to be evaluated as input and responds with the computed normal forms:

```
==> square (2+3)
25
```

On Unix-like systems, it is also possible to run Q scripts as standalone programs, using “shebangs” like ‘`#!/usr/bin/q -c main`’ as usual; on Windows, the same effect can be

achieved with a “batch file” wrapper or by translating the script to an executable using the `qcwrap` program. Of course, you can also run the interpreter without any source script and just use it as a sophisticated kind of desktop calculator. The input language understood by the interpreter is a somewhat “shell-like” line-oriented version of the Q language that encompasses only the expression syntax and some special commands tailored for interactive usage (try `help commands` sometime to learn more about this). The default `==>` prompt can be changed with the `prompt` command. For instance, `prompt "\n\M>> "` sets a Hugs-like prompt showing the basename of the main script. You can also add commands like this to the interpreter’s interactive startup file, `~/.qinitrc`, to configure the system to your liking.

The interpreter has a built-in symbolic debugger which allows you to execute a reduction sequence step by step:

```
==> debug on; square (2+3)
** 2+3 ==> 5
  0> test.q, line 1: square 5 ==> 5*5
  (type ? for help)
  :
** 5*5 ==> 25
** square 5 ==> 25
25
```

The lines starting with `**` indicate the reductions as they are performed by the interpreter during a recursive evaluation of the target expression. First the expression `2+3` is reduced to `5`, then the debugger stops at the user-defined rule for `square`. Hitting the carriage return key at the debugger’s `:` prompt continues execution with the next reduction, `5*5 ⇒ 25`. Having computed the value of the right-hand side of `square`’s definition, the interpreter now performs the reduction `square 5 ⇒ 25` and finally prints the end result, `25`.

The debugger offers all the operations you would expect in such a facility, including commands to set breakpoints, walk around the stack of active rule invocations, step over rules, inspect the values of variables in a rule and evaluate arbitrary expressions in the context of the current variable bindings, etc. While the output of the debugger can look a bit daunting at times, it will show you exactly what happens during a lengthy computation and where it happens, and thus is an indispensable tool for the experienced Q programmer.

The interpreter provides another useful facility which allows you to print the CPU time, number of reductions performed and memory cells (a.k.a. expression nodes) needed during an evaluation:

```
==> debug off; square (square (2+3))
625

==> stats
0 secs, 5 reductions, 3 cells
```

You can also obtain profiles of individual functions, counting the number of reductions involving a given function as the head symbol:

```
==> profile square (+) (*)

==> square (square (2+3)); profile
625
    2 (*)
    2 square
    1 (+)
```

Of course the results (2 invocations of the ‘\*’ operator and `square`, 1 invocation of ‘+’) are quite obvious in this simple example, but with more complicated definitions the `profile` command quickly becomes an important tool for identifying bottleneck operations in a program.

## 2.2 Conditional Equations

Equations may also include a condition part, called a “qualifier” in Q parlance, as in the following recursive definition of the well-known factorial function:

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

Note that this definition actually consists of *two* equations for the same left-hand side ‘`fact N`’; if an equation doesn’t have its own left-hand side, it uses the most recent one. The Q interpreter always picks the first *applicable* equation in the program text, meaning that the left-hand side must match the target expression and any conditions must evaluate to `true`. The interpreter also checks that the qualifying condition actually yields a truth value and raises a runtime error if the result is neither `true` nor `false`. The `otherwise` keyword denotes an empty condition which is always true. This allows you to point out the “default” case of a definition, but is really just syntactic sugar for the human eye and has no effect on the order in which equations are considered. In our example, the first equation will only be applicable if `N>0` holds; otherwise the second equation will be applied.

Qualifiers may also be written on the left-hand side of an equation, as follows (note the additional colon before the equals sign, which is needed to separate the left-hand qualifiers from the right-hand side of the equation):

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

The two forms are more or less equivalent and can be mixed. The main difference is that left-hand qualifiers may also be shared by multiple equations for the same left-hand side

expression. More precisely, the scope of a left-hand qualifier extends up to (but excluding) the next equation with either another left-hand qualifier or a new left-hand side expression. For instance, here is a definition of Ackerman's function (only try this with small values of  $M$  and  $N$ ; for values larger than about 3 it will take ages to compute):

```
ack M N if M>0:    = ack (M-1) (ack M (N-1)) if N>0;
                  = ack (M-1) 1 if N=0;
                  if M=0:    = N+1;
```

Here the qualifier  $M>0$  on the left-hand side of the first equation applies to both the first and the second equation. During evaluation, left-hand qualifiers will be considered first (before the right-hand qualifiers) and will only be evaluated once, at the time the first equation for the given left-hand side is tried.

## 2.3 Variable Definitions

The following constructs are somewhat non-standard in term rewriting, which does not have any kind of “variable assignments”, but they provide a convenient means to abbreviate the values of common subexpressions both interactively and in source scripts.

### Global Variables

*Free* occurrences of a variable (i.e., variables occurring only on the right-hand side of an equation) may be given a value with ‘def’. This is useful if a value is used repeatedly in different equations and you don't want to recalculate it each time it is needed. For instance, here is a simple function to calculate the area of a circle, which uses the definition of a free variable  $PI$ :

```
def PI = 4 * atan 1;
area R = PI * square R;
```

For convenience, global variables are bound *dynamically*, so that you can modify the value of a free variable interactively at any time during an interpreter session. Note, however, that a new value can be set for a global variable only from the command line of the interpreter. Thus a free variable will not change its value during an evaluation, although it make take on different values in subsequent evaluations of the same expression:

```
==> area 2
12.5663706143592

==> def PI = 3.14; area 2
12.56
```

Free variables may also be declared explicitly, and optionally initialized at the same time, as follows:

```
var PI = 3.1415926535897931;
```

If a free variable is used or defined without a prior explicit declaration, it is implicitly declared as a “private” symbol of the defining module. An explicit declaration is required if you want to change the scope of the variable symbol to “public” (i.e., accessible in other modules, cf. Section 2.15). You can also enforce that a free variable value remains constant, by adding the ‘const’ modifier to the declaration (this only works in scripts, not on the command line):

```
const var PI = 3.1415926535897931;
```

Now any attempt to redefine the value of PI provokes an error message:

```
==> def PI = 3.14
! Cannot redefine const variable
>>> def PI = 3.14
      ^
```

Instead of just a single variable, definitions may also involve a pattern on the left-hand side which is to be matched against the value of the corresponding right-hand side. Such a *pattern-matching definition* binds all variables on the left-hand side simultaneously. An error is reported if the match fails. For instance:

```
==> def (A,B,C) = (1,2,3)

==> def (A,B,C) = 99
! Value mismatch in definition
>>> def (A,B,C) = 99
      ^
```

Note that this only works with ‘def’, not with ‘var’. That is, the left-hand side of a variable declaration with initializer must always be a single variable symbol, but of course you can first declare the variables using ‘var’ and then assign values to them in a pattern-matching definition using ‘def’.

As in Prolog, the *anonymous* variable ‘\_’ may be used to indicate a component value that we don’t care about:

```
def (A,_,C) = (1,2,3); // only A and C are bound here
```

In fact, the anonymous variable has a double meaning in Q, since on the command line a free occurrence of ‘\_’ refers to the most recent result of an expression evaluation:

```
==> 127/13
9.76923076923077
```

```
==> _*13
127.0
```

A single ‘def’ clause may contain multiple definitions, separated by commas, and later definitions may refer to variables introduced earlier:

```
def A = 1, B = A+1, C = B+1;
```

It is sometimes convenient to use a function symbol as a variable, e.g., when it is to be bound to a lambda function (see Section 2.6). This requires an explicit variable declaration:

```
==> var fact = \N.if N>0 then N*fact(N-1) else 1
```

```
==> map fact [1..10]
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

## Local Variables

Besides global definitions of free variables, the Q language also provides local definitions in the form of “where clauses” which allow you to bind additional local variables in an equation. Like variables bound in the left-hand side of an equation, but unlike free variables, local variables have *lexical scope*, which is determined statically by the surrounding program text, and they are visible only in the hosting equation. For instance, consider the following definition to recursively compute pairs of consecutive Fibonacci numbers, which uses two local variables A and B for the result of the recursive invocation of `fibs` in the first equation:

```
fibs N = (B,A+B) where (A,B) = fibs (N-1) if N>0;
          = (0,1) otherwise;
```

Like conditions, `where` clauses can also be written on the left-hand side of an equation. This allows you to share local variables between different equations for the same left-hand side. For instance, here is a little quadratic equation solver which returns a special term indicating an error condition when the discriminant is negative. Note that the value of the discriminant D computed on the left-hand side of the first equation is used on the right-hand side of both the first and the second equation.

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

As in global definitions, you can have multiple definitions in a single **where** clause, which are processed from *left to right*, and each occurrence of a variable on the right-hand side of an equation or a local definition refers to its most recent definition. Thus, e.g., ‘**where**  $X = \text{foo } Y, X = \text{bar } X$ ’ effectively binds  $X$  to **bar** ( $\text{foo } Y$ ). This works, not because the variables are mutable (they aren’t), but because each local definition introduces fresh variables and a new nested scope in the local variable environment. (These rules also imply that local values cannot be defined in a recursive fashion, at least not directly – but see Section 2.9 for a technique to work around this limitation.)

It is also possible to mix several **if** and **where** qualifiers in a single equation, but note that in this case the individual qualifiers are processed from *right to left* (just as you would read them in a mathematical definition):

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

As already mentioned, left-hand qualifiers are processed before qualifiers on the right-hand side:

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

Failed matches in **where** clauses with a compound pattern on the left-hand side do *not* raise runtime errors; instead, the **where** clause acts as a qualifier that makes the entire equation fail if the match fails. For instance, given the definition of **solve** above, you can check for the existence of real solutions as follows:

```
check P Q = true where (X1,X2) = solve P Q;
           = false otherwise;
```

## 2.4 Recursion

As we already saw in the preceding examples of the factorial and the Ackerman and Fibonacci functions, iterative calculations can be expressed in Q by means of recursion. In fact, just as in other modern functional languages, *all* repetitive computations are done that way; there are no built-in “loop” control structures of any kind. Note that general recursive definitions may need a variable amount of space on the evaluation stack.<sup>2</sup> However, there is a form of recursion called *tail recursion*, which can still be executed in constant stack space just like iterative constructs in traditional programming languages, because the

---

<sup>2</sup>The Q interpreter performs most evaluations on its own stack, which is resized dynamically as needed, and the maximum stack size can be set with the **stacksize** command. It is also possible to set the maximum stack size to 0 to disable the limit, but it is generally a good idea to have a reasonably large limit so that the interpreter can catch infinite recursion.

recursive call comes “last” during leftmost-innermost evaluation. The Q interpreter does the usual automatic “tail call elimination” to achieve this. For instance, Euclid’s algorithm to compute the greatest common divisor of two nonnegative integers can be expressed quite naturally as a tail-recursive definition in Q as follows:

```
gcdiv X Y = gcdiv Y (X mod Y) if Y>0;
          = X otherwise;
```

Many simple kinds of recursive algorithms, including our definition of the factorial from Section 2.2, can readily be translated to a tail-recursive form by introducing an *accumulating parameter*:

```
fact N      = fact_loop 1 N;
fact_loop P N = fact_loop (P*N) (N-1) if N>0;
              = P otherwise;
```

Note that here the recursion actually happens in the `fact_loop` helper function, which always invokes itself as the last step in the right-hand side of the second equation. (This differs from our earlier definition of `fact` in Section 2.2 where the recursive invocation of `fact` was a proper subterm of the right-hand side.) Tail call elimination ensures that the interpreter actually implements this recursive call as a kind of “goto” instruction, so that no extra stack space is required during the evaluation of `fact_loop`.

## 2.5 Pattern Matching

It almost goes without saying that, as a language based on term rewriting, Q lets you define functions by pattern matching. In fact, we have already seen pattern matching at work in variable definitions, so this shouldn’t come as a big surprise. For instance, here is an alternative definition of the factorial which handles the ‘`fact 0`’ case separately, using a constant pattern:

```
fact 0 = 1;
fact N = N*fact (N-1) otherwise;
```

Patterns can also be used to decompose structured arguments like tuples and lists. As already mentioned, lists are written in Prolog-like syntax, thus `[]` denotes the empty list and `[X|Xs]` a list starting with the head element `X` and continuing with the list of remaining elements `Xs`. More generally, `[X1,X2,...|Xs]` denotes a list with head elements `X1,X2,...`. List concatenation is denoted using the ‘`++`’ operator. These ingredients can be used to implement another well-known example, the quicksort function, with the following two equations:

```
qsort []      = [];
qsort [X|Xs] = qsort (filter (<X) Xs) ++ [X] ++
               qsort (filter (>=X) Xs);
```

Repeated occurrences of the same variable on the left-hand side must be matched to the same value.<sup>3</sup> For instance, the following function `uniq` removes adjacent duplicates from a list:

```

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

```

We mention in passing that Q’s equational definitions actually go well beyond the simple “decompose-and-bind” kind of pattern-matching definitions employed above, since there is no “constructor discipline” and thus arbitrary symbolic rewriting rules are possible. We will have to say more about this in Section 2.7.

## 2.6 Higher-Order Functions

Just like in any other functional language, functions are first-class objects in Q, which can be computed dynamically, stored in data structures, passed around as parameters, returned as function results and applied to arguments. Hence higher-order functions can be programmed in a straightforward manner. For instance, the `filter` function used in the quicksort example of the previous section is defined in the standard library as follows. In this case, the function argument `P` is a predicate expected to return the value `true` if an element should be included in the result list, `false` otherwise.

```

filter P []      = [];
filter P [X|Xs] = [X|filter P Xs] if P X;
                = filter P Xs otherwise;

```

Functions can also return functions as results. In fact this happens implicitly whenever a function is partially applied. But you can also compute new functions from existing ones in more interesting ways. For instance, the following `ntimes` function returns a function which applies the given function a given number of times:

```

ntimes N F = foldl (.) id [F : I in [1..N]];

```

Here we employed some typical functional “list voodoo”: The `foldl` function is an operation provided by the standard library which accumulates the results of applying a binary function over a list, from left to right, starting from a given initial value. In this example the iterated function is the function composition operator `(.)` and we start out with the identity function. The list argument of `foldl` in this case is a list comprehension which yields a list consisting of `N` `F`s. Let’s see this in action:

---

<sup>3</sup>Here, “same” means “syntactically the same” in the sense that both values *print out the same* in the interpreter. Note that the constants ‘0’ and ‘0.0’ are thus not the “same”, although they are “equal” in the sense that `0=0.0 ⇒ true`.

```

==> ntimes 0 (+1)
id

==> ntimes 4 (+1)
id . (+1) . (+1) . (+1) . (+1)

==> _ 23
27

```

Lambda abstractions provide yet another way to obtain new functions at runtime. The notation ‘`\X.Y`’ used here and in the following is just syntactic sugar for an application ‘`lambda X Y`’ of the built-in special form `lambda`.<sup>4</sup> Also note that, for efficiency reasons, lambdas are actually compiled to an internal representation at runtime. The original names of the lambda variables are lost in this process, which accounts for the generic variable names `X1`, `X2` etc. when a lambda object is pretty-printed. For instance, here is how we can quickly define the factorial and assign it to a variable in the interpreter (the function `map` used in this example applies a function to each member of a list):

```

==> var fact = \N.if N>0 then N*fact(N-1) else 1

==> fact
\X1 . if X1>0 then X1*fact (X1-1) else 1

==> map fact [1..10]
[1,2,6,24,120,720,5040,40320,362880,3628800]

```

The standard library defines a number of other higher-order functions that can be used to construct pretty much any function at runtime, instead of hard-wiring it into the program. Such computed functions are not quite as efficient as the hard-wired ones, but are still useful to create anonymous functions of all sorts and sizes on the fly. It goes without saying that this is an important feature for artificial intelligence applications, and in fact lambda abstractions and other special forms provide a powerful framework for *metaprogramming*. This will be discussed in more detail in Section 2.9.

## 2.7 Symbolic Rewriting

Since Q’s definition syntax is superficially similar to Haskell’s, it is necessary to point out some important differences between the two. Haskell’s equations are really just syntactic sugar, a convenient shortcut for ordinary function definitions. Q’s equations, on the other

---

<sup>4</sup>Multi-argument lambdas can be written as, e.g., ‘`\X Y.Z`’, which is equivalent to ‘`\X.\Y.Z`’. Q also supports pattern-matching lambda abstractions such as ‘`\(X,Y).Z`’, which matches its argument against a pair `(X,Y)` and binds the variables `X` and `Y` accordingly.

hand, form a term rewriting system in the sense of equational logic. In fact, the Q interpreter doesn't even "know" what a function is (apart from the fact that it is some kind of expression applied to another one); neither does it care whether a function definition is "total" or "partial". It just faithfully keeps on applying term rewriting rules to the target expression until it reaches a normal form.

Q also doesn't have an a priori distinction between "defined" and "constructor" functions, whereas Haskell always enforces the so-called "constructor discipline", meaning that the head symbol on the left-hand side of a Haskell equation must always be a defined function, and only constructor symbols are allowed inside the parameters (this is not only true for Haskell, but for virtually all functional programming languages permitting pattern-matching function definitions). Thus an equation like the following, which expresses the idempotence of the function `h`, is illegal in Haskell:

```
h (h X) = h X;
```

In Q the above is perfectly acceptable and will have the intended consequences:

```
==> h (h (h 99))
h 99
```

The Q interpreter will even happily evaluate "non-ground terms", i.e., expressions containing unbound variables. For instance, taking the definition of the `square` function from the beginning of this chapter as an example:

```
==> square (A+B)
(A+B)*(A+B)
```

We might as well take this as an invitation to add some symbolic rules for associativity and distributivity:

```
X+(Y+Z) = (X+Y)+Z;   X*(Y*Z) = (X*Y)*Z;
X*(Y+Z) = X*Y+X*Z;   (X+Y)*Z = X*Z+Y*Z;
```

With these definitions, we now have:

```
==> square (A+B)
A*A+A*B+B*A+B*B
```

Note again that this kind of symbolic processing is impossible to do directly (without adding an extra level of interpretation) in languages like ML and Haskell because the associativity and distributivity equations violate the constructor discipline.

The only restriction that Q enforces is that the left-hand side of an equation may not be a constant by itself. Thus rules like the following are invalid and will provoke an error message from the compiler:

```
[X,Y] = [Y,X]; // WRONG!  
true  = false; // WRONG!
```

On the other hand, function applications with a constant or even a variable in the “head” position are perfectly legal. This allows you to pull off some neat little programming tricks like the following equations which tell the interpreter how to apply a list of functions to an argument:

```
[] X      = [];  
[F|Fs] X = [F X|Fs X];
```

Example:

```
==> [(+1),(*2),(1/)] 4  
[5,8,0.25]
```

So Q’s abilities to rewrite expressions using equations are considerably more general than those of most other functional languages. This might first look like an arcane feature, but it does add a great deal of power to the language, especially in conjunction with the notion of “special forms” to be discussed below, because it allows advanced features like list comprehensions to be defined in Q itself, rather than hardcoding those facilities into the interpreter. It also makes the language conceptually much simpler, because there are no arbitrary restrictions on the form of the equations. Essentially, all the Q interpreter does is algebraic manipulation of formulae, only much faster than you could ever do it by hand.

## 2.8 Special Forms and Lazy Evaluation

Q’s “special forms” are a fairly unique feature of the language which allows macro-like metaprogramming facilities and lazy data structures to be treated in a uniform manner. They are essentially like Lisp’s special forms (hence the name), which take their arguments unevaluated, using “call by name”, but can be freely defined by the programmer rather than being hardcoded into the interpreter.

A simple example is the conditional expression, which is implemented in the standard library by the special form ‘`ifelse`’ as follows:

```
public special ifelse ~P X Y;  
ifelse P:Bool X Y = X if P;  
                = Y otherwise;
```

The ‘`~`’ token before the P argument indicates that this parameter is actually passed by value (this is the truth value which determines whether the value of X or Y is to be returned, so this value will be needed anyway). The remaining parameters of `ifelse`

are passed by name and will only be evaluated if and when they are needed. Thus an expression like `ifelse true X Y` will immediately return the value of `X` even if `Y` denotes, say, a non-terminating computation.

Special arguments become so-called *deferred* values (also known as *thunks* or *suspensions* in the literature) which will be left unevaluated as long as they are “protected” by a special form. But as soon as the value of a thunk is referred to in a “non-special” context, the deferred value will be evaluated by the interpreter. This also applies to free variables, which are treated as literals if they occur as a special argument, even if they are bound to a value; when evaluated, the variable will yield its current value. All this happens automatically; no explicit “forcing” of the deferred value is necessary.

Special constructors are handled in the same fashion. Streams (lazy lists) are an important example for this. `Q` has syntactic sugar for these objects, even though most of the stream operations are actually implemented in the standard library. Streams are written using the same syntax as lists, using curly braces instead of brackets; unlike `[]`, the binary stream constructor `{|}` is a special form. For instance, let’s concatenate two streams:

```
==> def S = {a,b,c}++{x,y,z}; S
      {a|{b,c}++{x,y,z}}
```

You probably noticed that the tail of the resulting stream, `{b,c}++{x,y,z}`, has not been evaluated yet; it has been “thunked”. But that is all right, because it *will* be evaluated as soon as we need it:

```
==> S!4
y
```

This also works if we reach down into the thunked value during pattern matching:

```
==> def {X,Y,Z|_} = S; Z
c
```

As already mentioned, `Q` provides the usual syntactic sugar for arithmetic sequences as well as list and stream comprehensions. A comprehension consists of an expression to be evaluated along with one or more clauses of the form `X in Xs` (indicating that the variable `X` is to be bound to each member of the list or stream `Xs` in turn) and other expressions denoting predicates which are applied as filters to the resulting list or stream. Using these facilities, Erathosthenes’ sieve can be implemented in exactly the same way as in Haskell:

```
primes      = sieve {2..};
sieve {N|Ns} = {N|sieve {M : M in Ns, M mod N<>0}};
```

`Q`’s streams work pretty much like Haskell’s (lazy) lists, minus the memoization feature. This means that stream elements and tails will normally be reevaluated any time they are needed. However, if desired then memoization can easily be added by applying `Q`’s `&`

operator. In fact there is a standard library function, appropriately called `lazy`, which recursively applies the memoization operator to all heads and tails of a stream so that they can be accessed using “call by need” rather than just “call by name”.

The reason that special arguments are not memoized automatically at all times in Q is that some arguments may involve functions with side-effects, in which case memoization is often undesirable. For instance, we can compute an infinite stream of (pseudo-) random integers quite conveniently using a stream comprehension as follows (this example also shows the use of the standard library function `strict`, `lazy`’s counterpart, which forces an entire stream to be evaluated immediately.):

```
==> def R = {random : I in {1..}}

==> strict (take 5 R)
{1432755108,182897266,2463011318,1507076992,4006100417}
```

This works as expected because the invocation of the built-in `random` function is a special parameter, so it will be recomputed as each stream element is produced. However, if we memoize the `random` subterm then it will only ever be evaluated once, resulting in a stream consisting of an infinite number of copies of the *same* random value:

```
==> def S = {&random : I in {1..}}

==> strict (take 5 S)
{40902312,40902312,40902312,40902312,40902312}
```

## 2.9 Metaprogramming

Q also employs special forms to provide fairly advanced capabilities for symbolic expression manipulation at runtime. One example that has already been mentioned is the built-in `lambda` function. While Q’s lambda abstractions look superficially like those in languages like Haskell and ML, there is an important difference: Lambdas are just *ordinary expressions* in the Q language and can thus be inspected and manipulated just like any other expression. For instance, you can create a lambda object and then extract its variable and body parts using pattern matching as usual:

```
==> def F = \X.(2*X+1); F
\X1 . 2*X1+1

==> def \V.B = F; V; B
X1
2*X1+1
```

You can also manipulate lambdas in various ways before actually applying and thereby evaluating them, which offers great opportunities for “metaprogramming”. For instance, while Q does not provide any built-in ML-style ‘let’ expression that binds a variable to a value, you can easily define one yourself in terms of `lambda`:

```
special let C X;
let (A=B) X = (\A.X) B;
```

For a more substantial example, let us extend ‘let’ so that it lets us produce *recursive bindings*. That is, we want to implement a special form that mimics ML’s ‘let rec’, which is useful if we want to compute recursive functions like the factorial on the fly. Just a plain lambda will not do the trick here, since anonymous functions cannot be recursive. Or can they? Of course the old hands among you already know the answer, and that it involves the *fixed point combinator*. Here is how we can implement this little gem of functional programming in Q:

```
fix = Z Z where Z = \Z F X.F (Z Z F) X;
```

Again, the old hands will quickly point out that this is actually the “applicative order” fixed point combinator, but this will suffice for our purposes here. (See Section 5.6 for more on this topic, if you like.) Now we can define ‘letrec’ as follows:

```
special letrec C X;
letrec (A=B) X = (\A.X) (fix (\A.B));
```

Let’s try this with the factorial:

```
==> var fact = letrec (F=(\X.if X>0 then X*F(X-1) else 1)) F

==> fact
\X1 . (\X2 . \X3 . if X3>0 then X3*X2 (X3-1) else 1) ((\X2 . \X3 .
\X4 . X3 (X2 X2 X3) X4) (\X2 . \X3 . \X4 . X3 (X2 X2 X3) X4) (\X2 .
\X3 . if X3>0 then X3*X2 (X3-1) else 1)) X1
```

This looks a bit scary, but works all right:

```
==> map fact [1..10]
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

For more elaborate manipulations of literal expressions, the Q language provides the built-in special “quote” constructor, which just prevents its argument from being evaluated:

```
==> def X = '(5+4); X
'(5+4)
```

Note that this is different from Lisp’s quote in that `'` is indeed a *constructor* symbol, i.e., it becomes part of the quoted expression. This is essential because in Q a deferred expression only remains unevaluated as long as it is “protected” by a special form. Any expression outside a special context is always in normal form. Consider, for instance:

```
==> def 'Y = X; Y
9
```

Here the argument of the quote is extracted from its special context using a pattern-matching variable definition, and is thus evaluated. The same effect can also be achieved with the backquote operator ```, which unquotes its argument:

```
==> `X
9
```

The backquote operator can also be used to forcibly evaluate and “splice” a subexpression into a larger expression regardless of the context (special or not). Together, these operations provide the necessary toolset to freely manipulate literal expressions before they are actually evaluated. For instance, here is the complete definition of the special form `listof` from the standard library which implements the list comprehensions (you can find this code in the `cond.q` standard library module):

```
special listofx A Cs;

listof A Cs          = `(listofx A Cs);

listofx A ()         = '[A];
listofx A (X in Xs|Cs) = '(cat (map (\X. `(listofx A Cs)) Xs)
                             if isvar X; // will always match
                             = '(cat (map (\X. `(listofx A Cs))
                                           (filter (eq true. (\X.true)) Xs)));
listofx A (X in Xs)  = '(cat (map (\X.[A]) Xs) if isvar X;
                             = '(cat (map (\X.[A])
                                           (filter (eq true. (\X.true)) Xs)));
listofx A (P|Cs)     = '(ifelse P `(listofx A Cs) []);
listofx A P          = '(ifelse P [A] []);
```

Note how the `listofx` function “compiles” the target expression and the tuple of comprehension clauses to the actual “code” (a pile of nested lambdas, `cat`, `map`, `filter` and conditional expressions), which is returned as a quoted expression. For instance:

```
==> listofx (2*I) (I in [1..4])
'(cat (map (\I . [2*I]) [1..4]))
```

The result is then “executed” when it gets unquoted with the backquote operator in the definition of `listof`. The backquote operator is also used here to splice subexpressions into quoted expressions (as can be seen, e.g., in the third and fourth equations above).

It is worth noting here that, to make constructs like ‘`listof`’ work properly when they are embedded in other lambda-binding terms, we also have to provide a little interface to the built-in ‘`lambda`’ function. To these ends, the standard library implementation of `listof` declares the function as an instance of the predefined `Lambda` type, and defines a rule for the predefined `lambdax` function:

```
public type ListComp : Lambda = special listof A Cs;
lambdax (listof A Cs) = listofx A Cs;
```

The purpose of these definitions is to allow `lambda` to recognize list comprehensions embedded in lambda terms and expand them accordingly. Analogous code should be used to implement any user-defined special form which employs `lambda` to bind variables, such as our ‘`let`’ and ‘`letrec`’ constructs above; we omitted this here for the sake of brevity.

## A Bit of Reflection

The bottom line of this section is that expressions are really first-class objects in `Q`, which you can manipulate to your heart’s content before actually evaluating them. This makes `Q`’s special forms much more versatile than the kind of suspensions provided by other languages. They are more like Lisp macros, only fully dynamic. While `Q` is not a fully reflective language (in particular, equations, data types and modules are not first-class), first-class symbolic expressions provide enough reflectiveness so that you can easily do most Lisp-like metaprogramming. Other modern functional languages like ML and Haskell fall short in this respect because they cannot deal with arbitrary expressions on a symbolic level without introducing an extra level of interpretation.

## 2.10 Algebraic Types

*Algebraic types* are data types whose members form a free term algebra in the sense of universal algebra, hence the name. Members of the data type are created from constructors which are applied to arguments forming the components of the data element. The type may have different constructors, to distinguish between different kinds of values. Algebraic types essentially are “union of product types”, similar to “records with variants” in conventional programming languages. They are the primary device to create new data types in most modern functional languages.

As a term rewriting language, `Q` essentially gives you algebraic data types for free, since *any* function will act as a constructor if it occurs in a normal form (a pure constructor is just a function not being defined anywhere). However, `Q` also allows you to explicitly assign function symbols as constructors to a type symbol and make use of this typing

information in the pattern matching process. For instance, a simple binary tree data type with `nil` and `bin` constructors can be defined as follows:

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

Note that the ‘`const`’ keyword in front of the constructor declaration tells the compiler that these kinds of objects are to be treated as constants (just like the built-in types of constants such as integers, lists, etc.), and so cannot be the left-hand side of an equation, cf. Section 2.7.

We can now add, e.g., an insertion operation `insert`, operations to convert between lists and trees (`bintree`, `members`), and a tree sort routine `treesort` as follows:

```
// insertion
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) otherwise;

// construct a tree from a list
bintree Xs             = foldl insert nil Xs;

// list the members of a tree using inorder traversal
members nil            = [];
members (bin X T1 T2) = members T1 ++ [X] ++ members T2;

// binary tree sort
treesort               = members . bintree;
```

Types can be used as “guards” on variables on the left-hand side of an equation. The variable will then only match instances of the data type, which is convenient if you want to avoid spelling out all possible constructor patterns for the type. For example, we can now define a “tree union” operation as follows:

```
T1:BinTree + T2:BinTree = foldl insert T1 (members T2);
```

Note that these checks only happen at runtime, rather than being used for static type analysis (which Q doesn’t have). Nevertheless, the tree union operation thus defined is “safe”; it will only be applied to proper `BinTree` arguments:

```
==> def T1 = bintree [17,5,26,5], T2 = bintree [8,17]; T1; T2
bin 17 (bin 5 nil (bin 5 nil nil)) (bin 26 nil nil)
bin 8 nil (bin 17 nil nil)
```

```

==> T1+T2
bin 17 (bin 5 nil (bin 5 nil (bin 8 nil nil))) (bin 26 (bin 17 nil
nil) nil)

==> members (T1+T2)
[5,5,8,17,17,26]

==> 99+T2
99+bin 8 nil (bin 17 nil nil)

```

Note that the interpreter refused to evaluate the last expression because the first operand was not a `BinTree` object and thus the defining equation of `+` failed to match.

## 2.11 Sub- and Supertypes

Types can also be “derived” from each other, which gives you a kind of object-oriented type system with single inheritance. Each of the subtype’s constructor terms will then also be matched by its supertypes. For instance, an “abstract” supertype `Tree` with two subtypes `BinTree` and `AVLTree` can be defined as follows:

```

type Tree;
type BinTree : Tree = const nil, bin X T1 T2;
type AVLTree : Tree = const anil, abin X H T1 T2;

bintree Xs = foldl insert nil Xs;
avltree Xs = foldl insert anil Xs;

```

Now you can go ahead and, like above, implement the basic `insert` and `members` operations for each of the subtypes (we omit the implementation of `AVLTree` here for brevity, but the full source code can be found in Section 5.2). Then a polymorphic tree union operation can be implemented as:

```

T1:Tree + T2:Tree = foldl insert T1 (members T2);

```

This definition is now applicable to any mixture of `Tree` arguments. The way we defined it, the type of the result of `+` will match that of the first operand:

```

==> def T1 = avltree [17,5,26,5], T2 = bintree [8,17]

==> T1+T2
abin 3 17 (abin 2 5 (abin 1 5 anil anil) (abin 1 8 anil anil)) (abin
2 26 (abin 1 17 anil anil) anil)

==> members (T1+T2)
[5,5,8,17,17,26]

```

## 2.12 Abstract Types and Views

An algebraic type can be turned into an *abstract data type* (ADT) by “hiding” its internal representation, i.e., the constructors. This is desirable if we want to make sure that only a selected subset of operations is given access to the internal representation of the data type. To achieve this in Q, the type is simply made “public” while its constructors are declared “private”, as follows:

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

As will be explained in more detail in Section 2.15, this means that only the type symbol is available outside the module which defines the type, where it can be used in type guards as usual. The constructors, however, are only available in the defining module. This raises an “abstraction barrier” which provides more safety and enforces modularity, but has the drawback that, outside the defining module, values of the type cannot be matched against their constructors any more, and thus we lose one of the main conveniences of an algebraic data representation.

As a remedy, Q allows you to define *views*, which effectively provide you with a “virtual” algebraic type representation for the ADT.<sup>5</sup> This gives you the best of both worlds; you can use the virtual representation in pattern matching just like you would in the case of an ordinary algebraic type, but the real internal representation is still hidden outside the defining module of the data type and thus the abstraction barrier is unimpaired. Moreover, the view can be tailored to the kind of access operations that you want to support, independent from the internal representation.

In order to define a view for the `BinTree` ADT, we first have to declare one or more *virtual constructors* for the type. These are usually some functions which construct members of the type from some other data. In our example it makes sense to employ the `bintree` operation, which constructs a `BinTree` object from a list, for that purpose. We can turn `bintree` into a virtual constructor of the `BinTree` type as follows:

```
public type BinTree = public virtual bintree Xs
  | private const nil, bin X T1 T2;
```

The `public` keyword in front of the declaration of `bintree` could actually be omitted here, since the scope of a constructor defaults to the scope of its host type, but we included the scope modifier anyway for clarity. Also note that the `bintree` function is *not* declared as ‘`const`’, as in reality it is not a constructor but a “defined function”. In fact, the definition of `bintree` is just the same as before:

```
bintree Xs = foldl insert nil Xs;
```

---

<sup>5</sup>Q’s view implementation is based on the original proposal by Wadler [21], including the notion of a bidirectional view transformation, which has been criticized because of the problem it poses for equational reasoning [16]. In Q, we take a pragmatic standpoint and leave it up to the programmer to define the view in such a manner that it reconstructs something equivalent to the original object when evaluated.

The second step now is to provide a definition for the built-in `view` function which, given an object of the target type, computes its representation in terms of the virtual constructor. There is one technical hitch here – as virtual constructors usually are ordinary functions, we must be careful not to evaluate the result of `view`. This is achieved with the built-in special quote constructor `'` (see Section 2.9), which in a sense “casts” the view to a normal form:

```
view T:BinTree = '(bintree Xs) where Xs = members T;
```

Now we have:

```
==> def T1 = bintree [17,5,26,5], T2 = bintree [8,17]
```

```
==> T1+T2
bintree [5,5,8,17,17,26]
```

Note that the interpreter now actually uses the new virtual representation of the type for pretty-printing `BinTree` objects. Pattern matching against this representation also works as expected (in both cases the implicit quote in front of the view is omitted):

```
==> def bintree [X,Y,Z|_] = T1+T2; (X,Y,Z)
(5,5,8)
```

The above view for the `BinTree` type is rather inefficient since the entire list of members will be computed whenever the view is needed in pattern matching. Fortunately, we can do better than that. It is possible to define views in a recursive fashion so that the tree members are made available one at a time instead. For instance, we might want to create a list-like view which delivers the tree members in ascending order. This can be achieved as follows:

```
public type BinTree = public virtual empty, cons X T
                      | private const nil, bin X T1 T2;

empty                 = nil;
cons X T:BinTree     = insert T X;

view nil              = 'empty;
view T:BinTree       = '(cons X T) where X = first T, T = delete T X;
```

The new view is defined in terms of two virtual constructors `empty` (which returns the empty tree) and `cons` (which takes a value and a tree as parameters and inserts the value into the tree). To define the view, we need two new operations, `first`, which returns the smallest member in the tree, and `delete`, which removes the given element from the tree. These functions can be implemented as follows [3]:

```

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 otherwise;

join nil T2          = T2;
join T1:BinTree 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;

first (bin X nil _)  = X;
first (bin X T1 T2) = first T1 otherwise;

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

```

With these definitions we now have:

```

==> def T = bintree [5,1,9,3]; T
cons 1 (cons 3 (cons 5 (cons 9 empty)))

==> def (cons X (cons Y _)) = T; (X,Y)
(1,3)

```

## 2.13 Enumeration Types

Another special case of algebraic types is the so-called “enumeration type”, which only has nullary constructors, all declared as ‘const’:

```

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

```

Without any further ado, such a definition automatically enables comparisons as well as some arithmetic and enumeration operations on the type, e.g.:

```

==> sun < mon; sun+3; fri-sun
true
wed
5

==> [mon..fri]
[mon,tue,wed,thu,fri]

```

Examples of built-in enumeration types are `Bool` (comprising the truth values, `false` and `true`) and `Char` (the subtype of `String` denoting the single-character strings):

```
==> false < true
true

==> "a"<"b"; "0"+3; "7"-"0"
true
"3"
7

==> ["a".."k"]
["a","b","c","d","e","f","g","h","i","j","k"]
```

## 2.14 Imperative Features

In contrast to pure functional languages such as Haskell, Q takes the pragmatic route in that it also provides (monad-free) imperative programming features such as I/O operations and mutable data cells (“references”), similar to the corresponding facilities in the ML programming language. It also supports exception handling and non-local value returns, as well as POSIX threads. It goes without saying that these features break Q’s basic equational semantics, but they certainly make life easier when dealing with complex I/O situations. The ‘`||`’ operator can be employed to execute such actions in sequence. For instance, a simple prompt/input interaction can be written as follows:

```
prompt = puts "gimme: " || gets;
```

### References

References are a kind of mutable “memory cell” for expressions. Three operations are provided: `ref`, which creates a reference from its initial value, `put`, which changes the referenced value, and `get`, which returns the current value. With these facilities you can realize mutable data structures and maintain hidden state in a function. For instance, the following function `counter` returns the next integer at each invocation, starting at zero:

```
def R = ref 0;
counter = put R (N+1) || N where N = get R;
```

There is a second, lazy kind of reference, called a *sentinel*, whose value is evaluated at the time the sentinel is garbage-collected. Sentinels provide a means to trigger automatic cleanup of Q data structures in the same fashion as the cleanup performed by some built-in data types actually implemented as C objects, such as files. Example:

```
==> def X = sentinel (puts "cleaning up...\n")

==> undef X
cleaning up...
```

## Exceptions

Q's term rewriting semantics dictate that the language is mostly exception-free. In particular, “mismatch of arguments” is usually not an error but means that the expression is already in normal form. However, it is possible to make a function raise an exception if this is desired and you don't care about losing the equational semantics. For instance, if you'd like the standard library function `hd` (which returns the head element of a list) to raise an exception if it is applied to an empty list parameter, you can add a rule like the following to your program:

```
hd [] = throw "hd: empty list";
```

It is then possible to handle such exceptions (as well as a few “hard” exceptions such as stack or memory overflow which are generated by the runtime system) with the built-in `catch` function. This facility is also useful for implementing non-local value returns (see Section 5.5 for an example).

## Threads

*Threads*, a.k.a. “light-weighted processes”, allow you to write “multithreaded” programs 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 main loop. Q provides a complete implementation of POSIX 1003.1b threads along with the usual synchronization facilities, which enable tasks to communicate via mutexes, conditions and semaphores (the latter are actually implemented as “semaphore queues” in Q, so that you can send arbitrary data from one thread to another).

Q makes the creation of threads really easy. All you have to do is invoke the special form `thread` with the expression to evaluate. The most basic usage is to store the thread handle returned by the `thread` function in some variable and then later use the `result` function on the handle to obtain the computed normal form. For instance:

```
==> def TASK = thread (sum [1..1000000])

==> // do some other work ...

==> result TASK // get the result
500000500000
```

## 2.15 Modules and Namespaces

Last but not least, Q has a simple but effective module system, which gives the programmer control over which data types and operations are to be exported by each module. Q adopts the convention that each source file is a separate module with its own namespace. Symbols defined in a module must be declared explicitly using the ‘`public`’ keyword if they are to be used in other modules; symbols declared implicitly or with ‘`private`’ are only available in the defining module. Modules are imported by using an `import` or `include` declaration (the difference between these two is that `include` automatically reexports all imported symbols). Name clashes between different modules can be resolved using qualified identifiers (‘`moo::foo`’) or alias declarations, and symbols imported from other modules can also be selectively reexported.<sup>6</sup>

Also note that the standard library (or more precisely the standard prelude script `prelude.q`, which in turn includes the other standard library modules) is imported automatically in other modules so that its functions, variables and types are always available (similar to Haskell’s standard prelude).

Let us take the polymorphic search tree data structure from Section 2.11 as an example. We might describe the export interface of this module by the following series of Q declarations:

```
public type Tree;
public type BinTree : Tree = private const nil, bin X T1 T2;
public type AVLTree : Tree = private const anil, abin X H T1 T2;

public bintree Xs, avltree Xs;
public insert T X, delete T X, members T;
```

As the example shows, it is possible to have a public type with private constructors, to ensure that outside the defining module objects of the data type can only be created using the appropriate interface operations. This enables you to enforce data abstraction and information hiding, see Section 2.12 for details.

We can put the above declarations together with the equations defining the operations into a sourcefile named, say, `searchtree.q`, which can then be accessed from another module by means of the following import declaration:

```
import searchtree;

/* example: */

def T1 = avltree [17,5,26,5], T2 = bintree [8,17], T = T1+T2;
```

---

<sup>6</sup>What Q currently lacks, however, is a way to selectively *import* symbols from a given module. It’s either all or nothing.

# Chapter 3

## The Library

No modern programming language is complete without an extensive software library covering the more mundane programming tasks. Q's standard library alone already provides hundreds of functions, so we cannot go into any detail here and we just mention the most important parts of Q's library in passing. It is by no means accidental that the following list reads more like a who's who of popular open source libraries, as we've preferred to make use of tried and proven, freely available software components wherever possible, instead of trying to reinvent the wheel (and reinvent it badly). The emphasis is on cross-platform libraries so that the programming interface is mostly the same across all platforms supported by the Q interpreter.

While the APIs listed below are not as comprehensive as the facilities of other (much larger) language projects such as Perl and Python, they do make it possible to tackle many practical programming tasks with ease, and there are some areas where Q really shines. In particular, Q's support for multimedia and computer music is rather well-developed and goes well beyond what is currently being offered for its bigger cousins like ML and Haskell. Moreover, Q has an elaborate C/C++ interface including support for the SWIG wrapper generator ([www.swig.org](http://www.swig.org)), which allows you to interface to other C/C++ libraries as needed. Conversely, it is also possible to embed Q in C/C++ programs, and John Cowan has written a Chicken interface which allows Q code to be executed from Scheme ([www.call-with-current-continuation.org](http://www.call-with-current-continuation.org)).

### 3.1 Core Libraries

These libraries are mostly distributed with the interpreter.

**Standard Library.** The standard library, which is written in Q itself, implements a lot of useful Q types and functions, such as complex and rational numbers, generic list and stream processing functions and container data structures (sets, dictionaries, hash tables, etc.). Quite a lot of this code was originally adopted from [3]. In addition, Rob Hubbard's rational and polynomial libraries [9, 10] are available as separate packages.

**System Interface.** Along with the standard library comes a fairly complete POSIX interface, written in C, which provides useful system functions such as low-level I/O, process and thread management, sockets, regular expression matching and internationalization features. Q implementations of the compiler generator tools Lex and Yacc are also available as a separate package.

**GUI and Graphics.** In the GUI department, Q relies on Tcl/Tk ([www.tcl.tk](http://www.tcl.tk)). An interface to Gnocl ([gnocl.sf.net](http://gnocl.sf.net)), which provides a bridge to GNOME and GTK including a corresponding GUI builder, is available as a separate package. For basic 2D raster graphics, Q employs GGI, the “General Graphics Interface” ([www.ggi-project.org](http://www.ggi-project.org)) and ImageMagick ([www.imagemagick.org](http://www.imagemagick.org)). In addition, the standard library (see above) also comes with a PostScript interface, which provides basic vector graphic capabilities via external programs such as Ghostscript.

**Scientific Programming.** Q has interfaces to the numerical computation software Octave ([www.octave.org](http://www.octave.org)) and IBM’s comprehensive data visualization software “Open Data Explorer” ([www.opendx.org](http://www.opendx.org)). Moreover, a combinatorial graph library, which also includes a Tk-based graph editor, is available as a separate package.

**Web and Databases.** Some further modules provide the necessary facilities for web programming, such as an Apache module ([httpd.apache.org](http://httpd.apache.org)), a Curl interface ([curl.haxx.se](http://curl.haxx.se)) and an interface to the XML and XSLT libraries from the GNOME project ([xmlsoft.org](http://xmlsoft.org)). Database access is provided using either ODBC ([www.unixodbc.org](http://www.unixodbc.org)) or SQLite ([www.sqlite.org](http://www.sqlite.org)). GNU dbm ([www.gnu.org](http://www.gnu.org)) is also supported, which is suitable for basic applications that just need to store indexed data in a file.

## 3.2 Multimedia Library

Q has an extensive collection of multimedia-related “add-on” modules, which currently cover the application areas sketched out below. Most of these facilities include support for soft realtime processing. (While Q has not been designed as a realtime-capable language per se, it seems to do very well for soft realtime processing, which is probably due to the fact that the current interpreter uses a traditional reference counting and freelist scheme to reduce garbage collection overhead.)

**3D Graphics and Sound.** We provide interfaces to OpenGL ([www.opengl.org](http://www.opengl.org)) and OpenAL ([www.openal.org](http://www.openal.org)). An interface to SDL ([www.libsdl.org](http://www.libsdl.org)), which offers similar functionality but has better support on some embedded devices, is being worked on.

**Digital Audio.** Q’s interface to PortAudio ([www.portaudio.com](http://www.portaudio.com)) provides portable and near-realtime audio device access. Additional modules interface to the libsndfile and lib-

samplerate libraries ([www.mega-nerd.com](http://www.mega-nerd.com), sound file processing and samplerate conversion) and FFTW3 ([www.fftw.org](http://www.fftw.org), fast Fourier transforms).

**Digital Video.** Q’s video interface is currently a bit limited, but at least it is possible to play back multimedia files in different formats using the Xine library ([xinehq.de](http://xinehq.de)).

**DSP Programming.** For signal processing needs, Q relies on Faust, Yann Orlarey’s functional DSP programming language ([faust.grame.fr](http://faust.grame.fr)). Faust programs can be compiled to shared modules which can be loaded and executed in Q programs.

**MIDI Programming.** Q uses Grame’s cross-platform MidiShare library to interface to MIDI devices ([midishare.sf.net](http://midishare.sf.net)). Operations for reading and writing MIDI files are also provided, and the distribution of this module includes various useful tools such as a MIDI player and a MidiShare patchbay.

**Computer Music.** In addition to the audio, DSP and MIDI interfaces, Q also offers various libraries dedicated to computer music applications. This includes support for OSC, Berkeley’s Open Sound Control protocol for communicating with multimedia equipment and software ([opensoundcontrol.org](http://opensoundcontrol.org)), and an interface to James McCartney’s “SuperCollider”, arguably one of the most advanced realtime sound synthesis engines available right now ([supercollider.sf.net](http://supercollider.sf.net)). Q also has a plugin interface for Pd a.k.a. “Pure Data”, a popular visual programming environment for computer music applications written by Miller Puckette ([puredata.info](http://puredata.info)). This plugin enables you to program Pd control objects in Q. A similar module for the “Max/MSP” environment (<http://www.cycling74.com>) is also being worked on.

# Chapter 4

## The Implementation

Naive implementations of general term rewriting are unacceptably slow. Therefore it was deemed necessary to equip the Q interpreter with its own virtual machine designed from scratch and specifically tailored to the Q language, rather than employing one of the many existing VMs. This enabled us to provide a reasonably efficient implementation of Q's term rewriting machinery and its somewhat peculiar kind of runtime special forms.

The resulting VM, called the *Q Virtual Machine (QVM)*, is a kind of stack machine with added term pattern matching capabilities. Q scripts are compiled to QVM bytecode, which is then interpreted. It is impossible to give a detailed account on the virtual machine here, but we can at least sketch out some of the internals of the QVM and give a concrete example of QVM bytecode. (You can safely skip this chapter unless you want to learn a bit more about Q's implementation.)

### 4.1 The Q Virtual Machine

QVM output programs generated by the bytecode compiler essentially consist of three major data structures:

- The *symbol table*, which stores the print names and special attributes of all symbols in the source script and imported modules, so that these properties are available at runtime. The symbol table is consulted by the interpreter during the construction and pretty-printing of expressions. The interpreter also updates the symbol table at runtime when new function and variable symbols are created dynamically by the user or the running program.
- The *pattern-matching automaton*, which is used at runtime both to select the equations matching a target expression and to match the left-hand sides of local variable definitions. The current implementation employs a refinement of the algorithm discussed in [7], which allows patterns to be matched efficiently during a single left-to-right scan of the target expression. This usually works very well, although there are a

few pathological cases of overlapping patterns which lead to an exponential blow-up of the automaton; fortunately, they are rare.

- The QVM *bytecode program* implementing the right-hand sides of equations and variable definitions. This is code for a virtual machine which evaluates expressions on a stack. Most instructions performed by the QVM are actually just pushes of constants and function symbols. Expression nodes for function symbols are therefore preallocated to speed up the latter operation. A few special instructions do all the rest of the work; they are summarized in Figure 4.1.

As in most other modern functional language implementations, expressions are represented as *term DAGs*, i.e., as directed acyclic graphs which allow the sharing of subterms, so that deep copies of subterms can be avoided. Special support is provided for handling lists and tuples in an efficient manner, although these could also be represented as ordinary constructor applications. Thus the QVM knows about exactly three built-in binary symbols: “cons”-style list and tuple nodes, and applications. All other symbols are considered nullary. Evaluation happens when processing function symbols and applications in the bytecode, in which case the pattern matcher is invoked to determine matching equations. Conversely, the pattern matcher ties back into the evaluator in order to handle virtual constructors as well as call-by-pattern while matching “thunked” arguments.

The QVM also keeps track of the global and local variable environments. In the current implementation, global variable definitions, which use dynamic scoping, are stored directly in the symbol table, while local bindings (which use lexical scoping instead) are already resolved at compile time, employing the expression stack for temporary storage of local data. The QVM also handles the “thunking” and evaluation of special arguments, and manages expression memory using reference counting and a freelist, allocating memory from the system in larger chunks to reduce the overhead of dynamic memory management. Garbage is *always* collected immediately, as soon as a reference count drops to zero, which makes it possible to perform automatic cleanup on “external” data (such as closing files, freeing dynamically allocated C data, etc.) as soon as an object becomes inaccessible. Of course, reference counting has the drawback that cyclic structures cannot be reclaimed, but one really has to go to some lengths to create such structures in Q and for most purposes they can easily be avoided. Legitimate usage cases mostly occur in lazy data structures stored in the global variable environment, which presumably exist during the entire lifetime of a program anyway.

The QVM also provides the necessary hooks to call built-in as well as other external operations and to represent external data structures which are actually implemented in C or C++. External objects are wrapped up in a special kind of expression which provides a pointer to the C/C++ data structure. In addition, built-in constant types such as numbers and strings are also stored in a format compatible with C. Marshalling between Q data and C/C++ therefore incurs only very little overhead, which makes it possible to provide efficient interfaces to third-party libraries written in C or C++.

Name	Mnemonic	Example	Meaning
return	!RET	!RET	Return from rule. This also performs the necessary cleanup of arguments and temporaries on the stack.
lvalue	[ <i>offs</i> ] $\$p_1\$p_2\$\dots\$p_n$	[0] $\$1\$1\$0$	Push a left-hand-side ( <i>offs</i> = 0) or local value ( <i>offs</i> > 0). $p_1p_2\dots p_n$ , $n \geq 0$ , denotes the subterm path reaching inside applications, lists and tuples, where $p_i = 0$ means the left, $p_i = 1$ the right subterm.
query	? <i>addr/offs</i>	? 00032/0	Check that the top of the stack is a truth value. Skip rule and advance to the next one if the stack top is <b>false</b> , continue with the current rule otherwise. The start address of the next rule for the same left-hand side is given by <i>addr</i> in hexadecimal (-0001 if none), along with the stack offset <i>offs</i> in decimal, which indicates the offset from the stack base pointer to which the stack is unwound if the equation fails (this will be zero unless there are any local definitions in the left-hand side).
match	?MATCH # <i>m</i> <i>addr/offs</i>	?MATCH #1 00032/0	Match the top of the stack against the left-hand side of a local variable definition. Skip rule and advance to the next one if the match fails, continue with the current rule otherwise. <i>m</i> is an index into the <i>match table</i> , which contains, for each local definition, the corresponding start state in the matching automaton. The meaning of the <i>addr</i> and <i>offs</i> parameters is the same as for the query instruction.
info	!INFO <i>mod, line</i> ( <i>addr/offs</i> )	!INFO /test/fib.q, line 2 (00032/0)	Source module and line number information at the beginning of a rule. This is used in the symbolic debugger, and also provides the start address <i>addr</i> of the next rule for the same left-hand side and the stack offset <i>offs</i> for left-hand side definitions (see above).
pop	!POP	!POP	Pop expression from stack.
constant	<i>integer</i> <i>float</i> <i>string</i>	4711 471.1 "abc"	Push an integer, floating point or string constant on the stack.
nil/void	[] ()	[] ()	Push an empty list or tuple on the stack.
cons/pair	[ ] ( )	[ ] ( )	Construct a list or tuple node, using the topmost two values from the stack (stack top is the tail list/tuple).
symbol	<i>function</i> <i>variable</i>	foo BAR	Evaluate or push nullary function or variable symbols.
application	@	@	Apply a function to an argument, using the topmost two values from the stack (stack top is the argument). Evaluate or push the resulting application.

Figure 4.1: QVM instructions.

## 4.2 QVM Bytecode Example

For a simple example, consider the following program for computing Fibonacci numbers (cf. Section 2.3):

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

This program actually doesn't require any standard library functions, so it is convenient to compile it with the `'--no-prelude'` option, which precludes the auto-import of the standard library and enables us to have a look at the bytecode generated from just the three equations in the script above.

Below you find the listing produced with the `'-d'` option of the bytecode compiler. For an explanation of the different instruction types please refer to Figure 4.1. The listing illustrates how the evaluation of the right-hand sides is implemented using rather conventional stack-based code, but also includes some tie-ins to the pattern matching automaton, which in this example is also used to match against the left-hand sides of the local variable definitions in the first and the second equation.

```
00000: !INFO /test/fib.q, line 1 (-0001/0)
00001: fib::fibs *
00002: [0]$1 (N)
00003: @
00004: ?MATCH #0 -0001/0
00005: [1]$0 (A)
00006: !RET
00007: !INFO /test/fib.q, line 2 (00032/0)
00008: (>) *
00009: [0]$1 (N)
00010: @ *
00011: 0
00012: @
00013: ? 00032/0
00014: fib::fibs *
00015: (-) *
00016: [0]$1 (N)
00017: @ *
00018: 1
00019: @
00020: @
00021: ?MATCH #1 00032/0
00022: [1]$1$0 (B)
00023: (+) *
```

```

00024: [1]$0 (A)
00025: @ *
00026: [1]$1$0 (B)
00027: @
00028: ()
00029: (|)
00030: (|)
00031: !RET
00032: !INFO /test/fib.q, line 3 (-0001/0)
00033: 0
00034: 1
00035: ()
00036: (|)
00037: (|)
00038: !RET

```

The description of the pattern matching automaton is listed below. Each state has various transitions on function symbols acceptable in the state; the ‘\_’ symbol indicates variable transitions, and type tags on function and variable symbols are shown in angle brackets. The corresponding bytecode entry points for matched equations and definitions are shown in the final states of the automaton (these are only used when matching equations). State 0 is the main start state, which is used for matching against the left-hand sides of equations. The auxiliary sections labelled ‘?MATCH #*m*’ implement the matching automata for the different ‘?MATCH’ instructions binding local variables in the bytecode above.

```

0:
    @ : 1

1:
    fib::fib : 2
    fib::fibs : 4

2:
    _ : 3

3: 00000

4:
    _ : 5

5: 00007

```

\*\*\* ?MATCH #0 \*\*\*

6:           <Tuple>(l) : 7

7:           \_ : 8

8:           <Tuple>(l) : 9

9:           \_ : 10

10:          <Tuple>() : 11

11: 00000

\*\*\* ?MATCH #1 \*\*\*

12:          <Tuple>(l) : 13

13:          \_ : 14

14:          <Tuple>(l) : 15

15:          \_ : 16

16:          <Tuple>() : 17

17: 00007

# Chapter 5

## Examples

A little source code says more than a thousand words. In this chapter you can have a look at a few small but more or less typical examples to get a quick impression of how Q programs look. All these programs are also included in the core distribution; after installation you can find them in the `examples` folder of the Q installation directory. We present the programs here mostly without further annotations, as the sources are already commented.

### 5.1 Basic Examples

These can be found in the `basics.q` script included in the distribution.

```
/* Some basic equational definitions. Start out with these examples to get an
   idea how Q programs look like. */

// two simple function definitions

/* Note that variables are capitalized, to distinguish them from function
   names. Function application is just juxtaposition, no parentheses are
   required. Parentheses are used for grouping expressions and tuples only.
   E.g., foo(X) is just the same as foo X, bar X Y applies a function bar to
   two parameters X and Y, while bar (X,Y) applies bar to a single argument (a
   pair) (X,Y). */

double X          = 2*X;
square X          = X*X;

// global variable definition, gives a value to a "free" variable

def PI = 4*atan 1;

// area of a circle, uses the variable from above
```

```

area R                = PI* $\text{square } R$ ;

// conditional equations (sign of a number)

/* Note that the left-hand side of an equation can be omitted to just repeat
   the left-hand side from the preceding equation. */

sign X                = -1 if  $X < 0$ ;
                    =  0 if  $X = 0$ ;
                    =  1 otherwise;

// ditto, with recursion (the factorial)

fact N                =  $N * \text{fact } (N-1)$  if  $N > 0$ ;
                    =  1 otherwise;

// the same, but using tail recursion (a.k.a. "looping") with an "accumulating
// parameter"

fact_loop P N        = fact_loop (P*N) (N-1) if  $N > 0$ ;
                    = P otherwise;

fact2 N              = fact_loop 1 N;

// local variable definitions a.k.a. "where" clauses (Fibonacci numbers)

fib N                = A where (A,B) = fibs N;
fibs N               = (B,A+B) where (A,B) = fibs (N-1) if  $N > 0$ ;
                    = (0,1) otherwise;

/* Higher-order functions. *****/

/* The filter function used in the following (as well as the foldl function
   used in the algebraic data type example below) is an example of a
   higher-order function (HOF) which is already defined in the standard
   library. The standard library contains many more such generic list
   functions which have proven to be useful in many contexts. In particular,
   you should take a look at stdlib.q in which most of these functions are
   defined. */

/* Divisibility predicate (does M divide N?). */

divides M N          = (N mod M=0);

```

```

/* Employ Erathosthenes' sieve to compute all primes up to a given number N.
   Note that the predicate 'neg (divides N)' returns true iff its argument is
   _not_ divisible by N. The 'neg' function is also defined in stdlib.q. It
   negates a predicate, in this case a "curried" form of the divisibility
   predicate defined above. This is used to filter the list of remaining
   candidates. */

primes N          = sieve [2..N];
sieve [N|Ns]      = [N|sieve (filter (neg (divides N)) Ns)];
sieve []         = [];

/* Algebraic data types. *****/

/* A simple binary search tree data type with nil and bin constructors and an
   insertion operation. (See searchtree.q for a more elaborate example.) */

type BinTree = const nil, bin X T1 T2;

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) otherwise;

// construct a tree from a list (uses the foldl function from stdlib.q)

bintree Xs        = foldl insert nil Xs;

// enumerate the members of a tree as a list, using inorder traversal

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

/* Unlike languages like ML and Haskell, Q does _not_ have a rigid distinction
   between "constructor" and "defined" function symbols, and actually we don't
   have to explicitly declare the constructor symbols at all. However, if we
   do, then we can use the type name as a "guard" on variables, as in the
   following definition of a "tree union" operation: */

T1:BinTree + T2:BinTree = foldl insert T1 (members T2);

// Example:

def T1 = bintree [17,5,26,5], T2 = bintree [8,17], T = T1+T2;

```

## 5.2 Search Trees

This is the full source code of the example discussed in Section 2.10 ff. The implementation of the unbalanced and balanced (AVL) binary search tree types follows [3]. These data structures are useful for searching and sorting, as well as to implement container data structures such as sets and dictionaries. (Q's standard library contains a similar implementation of various container types in terms of AVL trees.)

```
/* searchtree.q: An example showing the use of algebraic data types with
   inheritance. */

/* We first define a generic search tree type (we call this an "abstract" type
   since it doesn't have any constructors of its own). This type is used as
   the supertype for two concrete subtypes BinTree and AVLTree below. NB: The
   'public' keyword causes the type to be exported so that it can be used in
   other modules which import this script. */

public type Tree;

/* Interface operations to be provided by all subtypes of the Tree type:
   insert/delete elements, compute the list of members (in ascending order),
   test membership. In this example we allow multiple instances of the same
   element in a tree. This means that the resulting trees will represent
   multisets. */

public insert T X, delete T X, members T, member T X;

/* Generic tree operations implemented in terms of the interface functions:
   size, union, difference, intersection, subset comparison. */

#T:Tree                = #members T;

T1:Tree + T2:Tree      = foldl insert T1 (members T2);
T1:Tree - T2:Tree      = foldl delete T1 (members T2);
T1:Tree * T2:Tree      = T1 - (T1 - T2);

T1:Tree <= T2:Tree     = all (member T2) (members T1);
T1:Tree >= T2:Tree     = all (member T1) (members T2);

T1:Tree < T2:Tree      = (T1<=T2) and then not (T2<=T1);
T1:Tree > T2:Tree      = (T1>=T2) and then not (T2>=T1);

(T1:Tree = T2:Tree)    = (T1<=T2) and then (T2<=T1);
T1:Tree <> T2:Tree      = not (T1=T2);

/* The binary tree and AVL tree subtypes, along with corresponding virtual
```

```

    constructor functions which create a tree from a list of its members. The
    real constructors are private so that they cannot be (ab)used outside this
    module. */

public type BinTree : Tree = virtual bintree Xs
    | private const nil, bin X T1 T2;
public type AVLTree : Tree = virtual avltree Xs
    | private const anil, abin H X T1 T2;

/* Define the corresponding views in terms of the virtual constructors. */

view T:BinTree          = '(bintree ~(members T));
view T:AVLTree         = '(avltree ~(members T));

/* Note that the above definitions of the generic tree operations apply to
   _all_ subtypes of Tree (provided that they implement the required interface
   operations). Examples: */

def T1 = avltree [17,5,26,5], T2 = bintree [8,17], S = T1+T2, T = T1-T2;

/* Also note that the views for BinTree and AVLTree objects we defined above
   let us use the virtual constructors in pattern-matching definitions just as
   if they were real constructors. The virtual representations of these
   objects will be constructed on the fly, as they are required during pattern
   matching. Since the virtual constructors are public, this will even work
   outside this module. Examples: */

def avltree Xs = S, avltree Ys = T;

mymembers (bintree Xs)          = Xs;
mymembers (avltree Xs)         = Xs;

/* Implementation of the BinTree operations. */

bintree Xs                      = foldl insert nil Xs;

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

member nil Y                    = false;
member (bin X T1 T2) Y          = member T1 Y if X>Y;
                                = member T2 Y if X<Y;
                                = true otherwise;

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) otherwise;

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 otherwise;

join nil T2                 = T2;
join T1:BinTree 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;

/* Implementation of the AVLTree operations (after Bird/Wadler). */

avltree Xs                  = foldl insert anil Xs;

members anil                = [];
members (abin H X T1 T2)    = members T1 ++ [X|members T2];

member anil Y               = false;
member (abin H X T1 T2) Y   = member T1 Y if X>Y;
                           = member T2 Y if X<Y;
                           = true otherwise;

insert anil Y                = abin 1 Y anil anil;
insert (abin H X T1 T2) Y    = rebal (mknode X (insert T1 Y) T2) if X>Y;
                           = rebal (mknode X T1 (insert T2 Y)) otherwise;

delete anil Y                = anil;
delete (abin 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 anil T2                 = T2;
join T1:AVLTree T2          = rebal (mknode (last T1) (init T1) T2)
                           otherwise;

init (abin H X T1 anil)     = T1;

```

```

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

last (abin H X T1 anil)        = X;
last (abin H X T1 T2)          = last T2 otherwise;

/* mknode constructs an AVL tree node, computing the height value */

mknode X T1 T2                  = abin (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 anil                     = 0;
height (abin H X T1 T2)         = H;

slope anil                      = 0;
slope (abin H X T1 T2)          = height T1 - height T2;

/* rebal rebalances after single insertions and deletions */

private shl T, shr T;

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

/* rotation operations */

rol (abin H X1 T1 (abin H2 X2 T2 T3))
                                = mknode X2 (mknode X1 T1 T2) T3;

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

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

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

```

## 5.3 Huffman Trees

Here is another interesting example of a tree data structure [1, Section 2.3.4].

```

/* huffman.q: Huffman encoding trees */

/* written by Albert Graef, 05-08-1993 (see also Abelson/Sussman: Structure
   and Interpretation of Computer Programs, MIT Press, 1985)
   revised 11-26-1993, 3-27-1997, 12-19-2000, 03-02-2002, 01-30-04 AG */

/*

```

NOTE: Alphabets should consist of two symbols at least. Otherwise the single symbol will be encoded as the empty list, which leads to infinite recursion when decoded. This could be considered a bug. ;-)

For instance, try the following:

```

==> def message = chars "Alice in Wonderland"
==> def H = huffman_tree (weights message)
==> def code = encode H message
==> strcat (decode H code)

```

A more amusing example is perhaps the following exercise from Abelson/Sussman 1985, p. 125:

"The following eight-symbol alphabet with associated relative frequencies was designed to efficiently encode the lyrics of 1950s rock songs. (Note that the "symbols" of an "alphabet" need not be individual letters.)

A	2	NA	16
BOOM	1	SHA	3
GET	2	YIP	10
JOB	2	WAH	1

Generate a corresponding Huffman tree, and use it to encode the following message:

```

Get a job
Sha na na na na na na na na
Get a job
Sha na na na na na na na na
Wah yip yip yip yip yip yip yip yip
Sha boom

```

How many bits are required for the encoding? What is the smallest number of bits that would be needed to encode this song if we used a fixed-length code for the eight-symbol alphabet?"

```

*/

/* Huffman encoding trees are represented as binary trees whose leaves
   (represented by the tip symbol) carry individual symbols and their
   weights (frequencies), and whose interior nodes (represented with the bin
   symbol) store the sets of symbols (represented as lists) found in the
   corresponding subtrees, together with the corresponding weights (which are
   the sums of the subtree weights). The nil symbol denotes an empty tree. */

public type HuffmanTree = const nil, tip X W, bin Xs W H1 H2;

private syms H, weight H;

syms (tip X W)          = [X];
syms (bin Xs W H1 H2)  = Xs;

weight (tip X W)       = W;
weight (bin Xs W H1 H2) = W;

/* The decoding algorithm. It takes as its arguments a Huffman tree and a
   list of zeros and ones, and reconstructs the original message. */

public decode H Bs;
private decode1 H U Bs;

decode H Bs              = decode1 H H Bs;

decode1 H (tip X W) Bs  = [X|decode H Bs];
decode1 H (bin Xs W H1 H2) [B|Bs]
    = decode1 H H1 Bs if B=0;
    = decode1 H H2 Bs otherwise;
decode1 H U []         = [];

/* The encoding algorithm. It takes as its arguments a Huffman tree and
   a list of symbols, and returns the coded message. */

public encode H Xs;
private encode1 H X;

encode H Xs              = cat (map (encode1 H) Xs);

encode1 (tip X W) X     = [];
encode1 (bin Xs W H1 H2) X
    = [0|encode1 H1 X] if any (=X) (syms H1);
    = [1|encode1 H2 X] if any (=X) (syms H2);

```

```

/* Construct a Huffman tree, starting from a list of (symbol,weight) pairs. */

public huffman_tree XWs;
private mk_huffman_tree Hs, mk_leaf_set XWs, mk_leaf XW;
private add_tree H1 H2, merge_tree H1 H2;

huffman_tree XWs          = mk_huffman_tree (mk_leaf_set XWs);

mk_huffman_tree []        = nil;
mk_huffman_tree [H]       = H;
mk_huffman_tree [H1,H2|Hs]
    = mk_huffman_tree (add_tree (merge_tree H1 H2) Hs);

mk_leaf_set XWs           = foldr add_tree [] (map mk_leaf XWs);
mk_leaf (X,W)             = tip X W;

add_tree H1 []            = [H1];
add_tree H1 [H2|Hs]       = [H2|add_tree H1 Hs] if weight H1 >= weight H2;
    = [H1,H2|Hs] otherwise;

merge_tree H1 H2          = bin (syms H1++syms H2) (weight H1+weight H2) H1 H2;

/* Determine the (symbol,weight) pairs for a particular message: */

public weights Xs;
private add_weight X XWs;

weights Xs                = foldr add_weight [] (qsort (<) Xs);

add_weight X []           = [(X,1)];
add_weight X [(X1,W1)|XWs]
    = [(X1,W1+1)|XWs] if X=X1;
    = [(X,1),(X1,W1)|XWs] otherwise;

```

## 5.4 Streams

Lazy evaluation paves the way for some important and powerful programming techniques. The following program `streams.q` introduces streams as a device to implement different kinds of backtracking and dynamic programming algorithms in a uniform setting.

```

/* Some stream programming examples. 2006-06-07 AG */

/* Helper function to print a stream. E.g.: 'print primes'. */

```

```

print          = do (\X.printf "%s\n" $ str X || flush);

/* Implicit definition of {1..}, following SICP (http://mitpress.mit.edu). */

ones          = repeat 1;
ints          = {1|zipwith (+) ints ones};

/* The same as a variable, evaluated lazily (with memoizing). This is much
   faster since the stream is built in-place, by "chasing its own tail". This
   works because Q's global variable environment uses dynamic binding, and the
   (unevaluated) instance of 'ints2' on the right-hand side of the definition
   will not be evaluated before the variable has already been defined. Thus
   the stream effectively includes a circular reference to itself. */

var ints2     = lazy {1|zipwith (+) ints2 ones};

/* Hamming sequence, using the same idea as above. (Original version
   contributed 05-08-1994 by Klaus Barthelmann.) Note that if you leave out
   the 'var' in front then the tails grow exponentially due to the repeated
   recursive invocations of 'hamming', with the obvious undesirable
   consequences on running time and memory requirements. */

var hamming   = lazy {1|merge (map (2*) hamming)
                             (merge (map (3*) hamming) (map (5*) hamming))});

merge {X|Xs} {Y|Ys}
      = {X|merge Xs {Y|Ys}} if X<Y;
      = {Y|merge {X|Xs} Ys} if X>Y;
      = {X|merge Xs Ys} otherwise;

/* Fibonacci numbers. Again the same "chase your own tail" idea at work
   here. */

var fibs     = lazy {0,1|zipwith (+) fibs (tl fibs)};

/* Another implementation of the stream of Fibonacci numbers with global
   hashing. */

special hashed X; def H = ref emptyhdict;
hashed X       = get H!'X if member (get H) 'X;
               = put H (update (get H) 'X Y) || Y where Y = X;

fibs2        = lazy {0,1|zipwith (+) (hashed fibs2) (tl (hashed fibs2))};

/* Erathosthenes' prime sieve using stream comprehensions. */

```

```

primes          = sieve {2..};
sieve {N|Ns}    = {N|sieve {M:M in Ns,M mod N<>0}};

/* The same without using stream comprehensions (not quite as elegant, but
   faster). */

primes2         = sieve2 {2..};
sieve2 {N|Ns}   = {N|sieve2 (filter (neg (divides N)) Ns)};
divides M N     = (N mod M=0);

/* Another sieve algorithm. This one computes the stream of Kth powers without
   using multiplication. (Contributed 04-03-1993 by Klaus Barthelmann.) */

powers K        = pow_sieve K {0..} if K>=1;
pow_sieve 1 Xs  = Xs;
pow_sieve K Xs  = pow_sieve (K-1) (scanl (+) 0 (dropmod K 0 Xs)) if K>1;
dropmod K I {X|Xs}
                = dropmod K (I+1) Xs if I mod K = 0;
                = {X|dropmod K (I+1) Xs} otherwise;

/* A nice 8 queens algorithm, literal translation of a Haskell program that
   can be found on the web. */

queens8         = queens 8;
queens 0        = {[]};
queens N        = {[Q|B] : B in queens (N-1), Q in [0..7], safe Q B};
safe Q B        = not any (checks Q B) [0..(#B-1)];
checks Q B I    = (Q=B!I) or else (abs(Q-B!I)=I+1);

/* Another stream comprehension example. Compute the stream of all
   permutations of the given list Xs. */

permute []      = {[]};
permute Xs     = {[Y|Ys] : Y in Xs, Ys in permute (filter (neq Y) Xs)};

/* The same implemented directly, without using stream comprehensions.
   Essentially, this just saves one nested map and hence is a bit faster,
   but also less readable and not as straightforward to write. */

permute2 []     = {[]};
permute2 Xs    = streamcat
                (map (\X.map (cons X) (permute2 (filter (neq X) Xs)))) Xs);

```

## 5.5 Eight Queens

Here are two more variations of the 8 queens algorithm. The first one is a more elaborate version of the algorithm from the preceding section, which illustrates how to implement backtracking using streams.

```
/* queens.q: determine valid placements of N queens on an NxN board
   05-08-1993 AG */

/* Say 'hd (queens 8)' to obtain the first solution. (On slower computers this
   may take a while.) To print all solutions, evaluate 'print (queens 8)'
   (abort with Ctl-C when you get bored). Placements are given as lists of
   (column,row) pairs. The algorithm enumerates the solutions in
   lexicographic order (first all solutions which place the first queen into
   row 1, then the solutions which place it into row 2, etc.). To determine
   placements on any NxN board, use N as the parameter to queens. */

/* queens N returns the stream of all valid placements of N queens on an
   NxN board: */

queens N      = cols N N;

/* cols N I generates the stream of all valid placements of I queens in
   columns 1..I of an NxN board: */

cols N I      = {[[]] if I<1;
                 = {P++[(I,J)] : P in cols N (I-1), J in [1..N], safe (I,J) P}
                 otherwise;

/* safe determines whether a queen at position (I1,J1) is safe w.r.t. a given
   placement: */

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

/* check determines whether queens at positions (I1,J1) and (I2,J2) hold
   each other in check: */

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

/* helper function to print a stream */

print        = do (\X.printf "%s\n" $ str X || flush);
```

The second variation shows how to implement backtracking in a direct fashion, using

side-effects. This solution employs the `fail` function, which makes an equation fail while it is already being executed, as well as the `catch` and `throw` exception handling operations, which are used in the `queens1` function to implement non-local value returns.

```

/* queens2.q: a backtracking variation of the N queens algorithm which uses
   exception handling 07-23-01 AG */

/* The basic backtracking technique is the same as in queens.q, but the
   implementation here first pursues a single solution path and then uses the
   fail construct to force backtracking. This is a little bit faster and also
   needs less memory than the stream-based implementation in queens.q. */

/* queens N prints all valid placements of N queens on an NxN board. Note the
   use of 'fail' in the recursive branch of the algorithm (where a new
   placement is added to the current list), which forces backtracking. The
   rest of the algorithm is tail-recursive. */

queens N      = search N 1 1 [];

search N I J P = printf "%s\n" $ str P || flush 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;

/* queens1 N just computes the first solution. This is the same algorithm as
   above, but it employs a catch/throw construct to exit from the algorithm
   and return the result as soon as we have found a solution. Note that the
   "exception handler" in this case is simply the identity function. */

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;

/* Verify that a queen placement is safe w.r.t. a given list of other
   placements. This is the same as in queens.q. */

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);

```

## 5.6 Fixed Points

Functional programmers are obsessed with “fixed point combinators”, that is, functions  $G$  with  $G(F) = F(G(F))$  for all functions  $F$ . As we’ve seen in Section 2.9 already, there is a fairly practical reason for this, since such functions can be used to create anonymous recursive functions. The following program `fixpt.q` illustrates how both the “call-by-name” and “call-by-value” fixed point combinators  $Y$  and  $Z$  can be implemented in Q (note that the combinator names are decorated with a leading underscore in the program, as function symbols must always start with a lowercase letter or ‘\_’).

```
/* Fixed point combinator examples (originally written by Klaus Barthelmann
   08-1993, completely revised 06-2006 by Albert Graef). */

/* Fixed point combinators are characterized by the property that they compute
   fixed points of a function, i.e., G is a fixed point combinator iff G F = F
   (G F) for each (higher-order) function F. They provide a way to implement
   "anonymous recursion", i.e., define recursive functions using nothing but
   simple lambda abstractions. The basic idea is that the function is defined
   in terms of a second lambda abstraction which takes the function itself as
   an extra "placeholder" argument and performs a "single step" of the
   recursion. See, e.g., http://en.wikipedia.org/wiki/Y\_combinator for
   details. */

/* The applicative order fixed point combinator ("Z combinator"), straight
   from the textbook. */

_Z = \F.(\X.F (\Y. X X Y)) (\X.F (\Y. X X Y));

/* The normal order fixed point combinator ("Y combinator"). Here the textbook
   version, _Y = \F.(\X.F (X X)) (\X.F (X X)), must be slightly modified to
   make it work in Q, since normally Q's eager evaluation strategy would loop
   on _Y F. The trick is to break the infinite recursion by placing quotes at
   the right places. This will defer the recursive evaluations until the
   function value is actually needed. We just have to remember that in the
   definition of the "single step" function the quotes then have to be
   stripped again from the function placeholder argument (see below for
   examples). */

_Y = \F.(\X.F '(X X)) (\X.F '(X X));

/* Example 1: the factorial. Note the ' before the function placeholder
   argument in the Y combinator version, which is needed to get rid of the
   extra quotes in the definition of _Y. */

zfact = _Z (\F X.if X<=0 then 1 else X*F (X-1));
```

```

yfact = _Y (\'F X.if X<=0 then 1 else X*F (X-1));

/* Example 2: the stream of all Fibonacci numbers. Note the extra "dummy"
   argument in the Z combinator version. Also note the embedded lambda in the
   body of the iterated function, which makes sure that the function argument
   only needs to be evaluated once in the lambda body. (If we don't do this
   then the running time will grow exponentially.) */

zfibs = _Z (\F X.lazy {0,1|(\F.zipwith (+) F (tl F)) (F X)}) ();

yfibs = _Y (\'F.lazy {0,1|(\F.zipwith (+) F (tl F)) F});

/* Helper function to print the elements of a stream (ex.: 'print zfibs'). */
print = do (\X.printf "%s\n" $ str X || flush);

```

## 5.7 Symbolic Rewriting

This is a simple example illustrating symbolic rewriting rules.

```

/* The Q interpreter can actually perform computations with arbitrary
   "symbolic" expressions, and equations can be arbitrary rewriting
   rules. E.g., the following equations implement distributivity and
   associativity. Try with something like 'square (A+B)'. */

(X+Y)*Z      = X*Z+Y*Z;
X*(Y+Z)      = X*Y+X*Z;

X+(Y+Z)      = (X+Y)+Z;
X*(Y*Z)      = (X*Y)*Z;

double X     = 2*X;
square X     = X*X;

/* Another tiny example: symbolic differentiation. E.g., try something like
   'diff X (5*square X+double X+10)'. */

diff X X     = 1;
diff X (U+V) = diff X U + diff X V;
diff X (U*V) = U*diff X V + V*diff X U;
diff X Y     = 0 otherwise;

/* A few additional simplification rules. */

```

```

X*0          = 0;
0*X          = 0;

X*1          = X;
1*X          = X;

X+0          = X;
0+X          = X;

```

Here is another example, which shows how to compute disjunctive normal forms of logical expressions.

```

/* dnf.q: This is a tiny example for symbolic expression manipulation. It
   transforms logical expressions into disjunctive normal form (DNF).
   05-08-1993 AG */

// eliminate double negations:
not not A      = A;

// push negations inward (de Morgan's laws):
not (A or B)   = not A and not B;
not (A and B)  = not A or not B;

// distributivity laws:
A and (B or C) = A and B or A and C;
(A or B) and C = A and C or B and C;

// associativity laws:
A and (B and C) = (A and B) and C;
A or (B or C)   = (A or B) or C;

```

## 5.8 System Programming

Q offers a C/C++ interface, which incurs very little overhead for marshalling between Q and C/C++ data, so that you can access your favourite system libraries quite easily and in an efficient manner. In particular, the core distribution already includes a fairly complete POSIX system interface, which lets you do many of the things that you'd normally use Perl and Python for. Here is a simple example showing the use of Berkeley sockets. It also illustrates the use of "byte strings" to store and transfer arbitrary binary C data. The distribution contains many more examples also for the web, database, graphics and multimedia interfaces, which turn Q into a useful scripting language.

```

/* dgram.q: connectionless server/client example using datagrams */

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;

```

# Chapter 6

## Conclusion

We believe that Q provides a viable alternative to “mainstream” modern-style functional languages like Haskell and ML, as it supports many of the same core features such as curried functions, function definitions by pattern matching, algebraic data types and eager as well as lazy evaluation. The main downside of Q is that there’s currently no native code compiler, so you have to be content with the performance of the interpreted implementation for now. One compelling advantage of Q is that it offers a more general approach to term rewriting, combined with runtime facilities for symbolic expression manipulation and Lisp-like metaprogramming. Q also has better third-party library support, at least in some important application areas, and can be further extended using SWIG, which has become a kind of standard in the world of scripting languages. Moreover, Q foregoes intricate type systems and monadic I/O and should therefore be more accessible to the novice, enabling the aspiring functional programmer to learn the art of “programming by equations” before becoming acquainted with other, more rigorous languages. Some will bemoan Q’s lack of static strong typing, but many of its current users actually see this as an advantage, and for a “functional scripting language” it certainly feels appropriate. “If you want ML, you know where to find it.”

## Future Work

Implementing a new programming language with appropriate library support is a daunting task, and it goes without saying that there are still many things to do. One of the top challenges is to develop a compiler which translates Q bytecode to reasonably efficient machine code. The Q interpreter can more or less compete in execution speed with other functional language interpreters, but native code generation would obviously be an important enhancement, extending Q’s scope towards more demanding and CPU-intensive applications, including a self-hosting compiler and interpreter.

The current interpreter also still has some room for improvement. In particular, since Q is already being used for soft realtime processing in multimedia and computer music applications, the advent of multicore architectures in commodity PC systems suggests

that we should provide an implementation which scales well with the ever-growing number of processors in SMP systems. This isn't the case right now as the current implementation involves some global shared data structures such as the expression heap and the runtime symbol table which need to be protected from concurrent read/write accesses. One approach to tackle this problem is the use of high-performance lock-free data structures such as those employed by the hard realtime and signal processing communities; see, e.g., [2]. However, at the time of this writing, some of these techniques still need to be adapted to modern SMP architectures, and most of them also pose portability problems, although efforts are underway to produce an open-source library which provides efficient cross-platform implementations of such operations.<sup>1</sup>

Another interesting research topic are less obtrusive type systems with automatic type inference for use in dynamically typed functional languages like Q. Nathan Whitehead has done some work on this for his Q dialect QT.<sup>2</sup> While Q's notion of algebraic types with inheritance appears to work rather well in the context of a term rewriting programming language, a separate type checker would be a useful tool which could also help to generate better code in a future native code compiler.

Another important question is how to make better use of Q's symbolic rewriting capabilities. The current standard library doesn't really exploit these as it was based on designs for languages of the Miranda/Haskell variety which lack those features. Implementing a full CAS (computer algebra system) in Q would surely be a major project in itself, but Rob Hubbard has already demonstrated with his comprehensive rational number and polynomial libraries that Q in fact provides a viable platform for such endeavours. Other interesting avenues for future developments are more and better functional data structure implementations, such as those described in Okasaki's book [15], and a high-level API for stream-based processing of event-driven control signals [18]. The latter is already being worked on, and should prove to be useful especially in computer music and other realtime multimedia applications.

Last but not least, a more mundane but quite essential and ongoing effort is the further advancement of the programming interfaces to third-party libraries. Andrew Berg has been working on an SDL<sup>3</sup> module, and we also plan to provide better support for video playback and editing via GStreamer<sup>4</sup> and vector graphics using Cairo<sup>5</sup>. Direct SWIG wrappers for more elaborate GUI libraries like GTK and Qt are also in order (GTK applications can already be implemented via Q's Gnocl interface, but currently Qt applications in Q need to wrap each individual GUI using the C/C++ interface, which is awkward). Also, as Yann Orlarey has pointed out, a more generic kind of functional GUI library might be called for, as a playground for research into novel GUI concepts.

---

<sup>1</sup><http://www.audiomulch.com/~rossb/code/lockfree/liblfd/>

<sup>2</sup><http://www.cse.ucsc.edu/~nwhitehe/qt.html>

<sup>3</sup><http://www.libsdl.org>

<sup>4</sup><http://gstreamer.freedesktop.org>

<sup>5</sup><http://cairographics.org>

# Bibliography

- [1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.
- [2] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 28–37. IEEE Computer Society Press, 1995.
- [3] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, 1988.
- [4] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. Technical Report UU-CS-2005-005, Institute of Information and Computing Sciences, Utrecht University, 2005.
- [5] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and implementation of an algebraic programming language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, LNCS 782, pages 228–244. Springer, 1994.
- [6] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.
- [7] Albert Gräf. Left-to-right tree pattern matching. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, LNCS 488, pages 323–334. Springer, 1991.
- [8] Albert Gräf. *The Q Programming Language*. <http://q-lang.sf.net>, 2007.
- [9] Rob Hubbard. “ $Q+Q$ ”: *Q Rational Number Library*. <http://q-lang.sf.net>, 2006.
- [10] Rob Hubbard. “ $Q[i][X]$ ”: *Polynomial Module (with Gaussian Number support)*. <http://q-lang.sf.net>, 2007.
- [11] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

- [12] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. <http://haskell.org>, September 2002.
- [13] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
- [14] Michael O’Donnell. *Equational Logic as a Programming Language*. Series in the Foundations of Computing. MIT Press, Cambridge, Mass., 1985.
- [15] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.
- [16] Chris Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, Baltimore, Maryland, USA, September 1998.
- [17] Rinus Plasmeijer and Marko van Eekelen. *Clean Version 2.0 Language Report*. University of Nijmegen, December 2001.
- [18] H. J. Reekie. *Realtime Signal Processing: Dataflow, Visual and Functional Programming*. PhD thesis, University of Technology at Sydney, Australia, 1995.
- [19] David A. Turner. An overview of Miranda. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 1–16. 1990.
- [20] Wouter van Oortmerssen. *Concurrent Tree Space Transformation in the Aardappel Programming Language*. PhD thesis, University of Southampton, UK, 2000.
- [21] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. Principles of Programming Languages*. ACM, 1987.
- [22] Wikipedia article “Functional programming”. [http://en.wikipedia.org/wiki/Functional\\_programming](http://en.wikipedia.org/wiki/Functional_programming), June 2007.