Igor N. Dultsev

yhaskell@live.com

Que

language specification

version 0.2

# Contents

[Contents 1](#_Toc362270556)

[Preface 2](#_Toc362270557)

[Hello World 2](#_Toc362270558)

[Program structure 3](#_Toc362270559)

[Package hierarchy 3](#_Toc362270560)

[Type system 3](#_Toc362270561)

[Object type 3](#_Toc362270562)

[Interface types 4](#_Toc362270563)

[Internal types 4](#_Toc362270564)

[Class types 4](#_Toc362270565)

[Algebraic types 4](#_Toc362270566)

[Function types 5](#_Toc362270567)

[Expressions 5](#_Toc362270568)

[Statements 6](#_Toc362270569)

[Expression statements 6](#_Toc362270570)

[Declaration statement 6](#_Toc362270571)

[Iteration statements 6](#_Toc362270572)

[Code transfer statements 6](#_Toc362270573)

[Execution and exception handling statement 7](#_Toc362270574)

[Import and Module statement 7](#_Toc362270575)

[Types 7](#_Toc362270576)

[Classes 7](#_Toc362270577)

# Preface

Que is a simple modern multi-paradigm type-safe language. Syntax of Que is composed in minimalist style, easy to read and understand.

Que is *object-oriented* language. Everything in Que is an object: modules, variables, instances of classes, functions, methods.

Que is *procedure-oriented* language. Program in Que can be composed without any objects, using just variables and functions.

Que is a *functional* language. It allows a concept of *pure* functions, which are computable, type safe, may be assigned to a variable and processed as any functional language allows us to.

Que uses garbage collection to ease a live of a programmer. Type safety and exception handling denies Que programs to do many incorrect things, including reading from uninitialized variables, indexing arrays wrongly and casting variables to incompatible types.

Que uses unified type system. Every type in Que has its own place in a lattice of types with Object type as lower bound of lattice and None as its upper bound.

The rest of this chapter contains a gentle introduction in a syntax and semantics of Que language. The rest of this document contains specifics of Que language in more detailed way.

## Hello World

Hello world program is a traditional way to show how the program in some language looks like. Therefore, here it is:

module hello

import lib.io

let main = args -> do

console.print “Hello, World!\n”

return 0

Que source files typically have an extension “.q”.

If we assume that file would have a name hello.q, we can compile and run this program by using compile and run commands in a command prompt:

$ qc hello.q

$ que hello.que

Hello, World!

$

The “Hello World” source starts by **module** directive indicating that this is a source code for hello module. Then, **import** lib.io directive tells compiler that content of lib.io module should be imported into our module scope. That allows then using a **console** object when we need it.

Next, we use let operator to create object called main which is a function. It gets argument args (it’s type is derived from the program) and processes it with do statement.

**console**.print procedure prints it’s arguments into the console, and **return** statement sets a return value of main function and finishes its execution.

## Program structure

The key organizational concepts in Que are programs, modules, types, members and packages. Program written in Que consists of one or many packages, which contain a number of modules.

Each module represent object module, and consist of its members as well as of underlying types.

Types can be divided in four groups: Classes, interfaces, algebraic types and functions.

Type members are methods and properties of objects, functions and variables of modules.

Que program consists of modules, each residing in one or many files. One or many modules may be compiled in a package (which usually have .que extension) and then may be packed into special form of executable that may be run under different operating systems.

Modules and classes compose hierarchical name structure, and dot-delimited system is used: for example console module (which resides in lib.io module) would have name lib.io.console.

Each package contain meta information allowing us to get rid of header files and #include statements.

## Package hierarchy

All type hierarchy is divided in two huge groups.

Packages with name starting with sys contain language-specific entites, like standard types (sys.object, sys.integer etc.). Also there lie modules like sys.ref (language reflection), and so on.

Packages with name starting with lib contain standard library. Work with IO system, networking, threading, and so on will reside in package of that group.

When user creates a program, he usually uses package with name of his program (as we used hello module for our Hello, World program), and stores all his program code inside that package.

When user creates a user library, he should put it in lib.usr package.

## Type system

In short, type system of Que may be described as follows:



### Object type

Object is a type that every object has. Any type derives from object; that way there always is a lower bound for all types in a type lattice.

### Interface types

Interface types are types that contain information about type signature: it defines signatures of all methods and operators that this type would have. Any type has its own interface type; that way any type (except object) realize some interface.

### Internal types

Internal types are the simplest types that may be found in any languages: byte, int, long, char, float, double.

### Class types

Class types are the most common types in every object oriented language. One class may inherit from another.

Class types are not structural compatible: for example, let say we have the following two classes:

class A is

private some = 42

public foo = () -> some

public A = x -> do some = x

class B is

private some = 42

public foo = () -> (math.abs some) + 58

public B = () -> do some = x

type I is

foo: () -> int

Both A and B realize the same interface type I, but obviously they are not and should be not compatible. Therefore, casting the object of type A to B and back is not possible:

let x = A ()

let y = B ()

console.print (foo x) # prints 42

console.print (foo y) # prints 100

y = x # error: invalid type cast

However, there is nothing wrong if we cast them both to I:

let z = x as I or null

console.print (foo z) # prints 42

z = y as I or null

console.print (foo z) # prints 100

x = z # still error: invalid cast

### Algebraic types

Algebraic types came to Que from functional world. Que supports both product and sum algebraic types.

Product types are supported as tuples.

One could think about sum algebraic types as enumerations reinvented. In a variable of sum type T, values of many types may be stored.

For example, all of types introduced below are sum types:

type Bool = True | False

type Maybe a = Nothing | Value a

type Tree a = Nothing | Leaf a | Node (Tree a) (Tree a)

In the examples above, a variable of type Bool may store values of True or False;

* A variable of type Maybe a may store values of Nothing or Value x, where x has type a.

let’s suppose that type argument a equals int for simplicity. Then the following values are correct values for variable of type Maybe a:

* + Nothing
  + Value 42
  + Value -925

Note, that value like “42” is not correct value for variable of type Maybe a

* A variable of type Tree a may store values of Nothing, or Leaf x (where x has type a), or Node x y (where x and y have type Tree a). So, correct values of variable t of type Tree int would be:
  + Nothing
  + Leaf 9
  + Node (Node Nothing (Leaf 1)) (Leaf 42)

### Function types

­There is two types of function in Que.

First one is a *pure* function. Pure function gets input data from its arguments and returns value. The value of pure function on the same argument will always be the same.

Pure function types are denoted as «a’ -> b’».

Second one is a dirty function or method. They may change values of fields, or their arguments.

For example, dist method in the code below doesn’t change anything in its body, but it is not pure, because it is not right in general that it will return the same value on the same argument (because we may change the object we call dist method from).

However, to method pure: if p1 = p2 and q1 = q2, then p1 to q1 will be equal to p2 to q2.

class Position is

public x, y : double

public dist = p : Position -> math.sqrt ((p.x – x)^2 + (p.y – y)^2)

public to = infix p q -> math.sqrt ((p.x – q.x)^2 + (p.y – q.y)^2)

Dirty function types are denoted as «a’ ~> b’»; methods are denoted as «(c’) a’ -> b’».

## Expressions

Expressions are constructed from operators and operands. Priority of operators apply. Operators may be overloaded, and new operators can be introduced.

List of operators (priority is the same in groups and lowers from beginning to the end):

* Primary operators:
  + Member access (x.foo),
  + Indexing (x[100])
  + Type operator (typeof x);
* Unary operators
  + Positive and Negative values (+x, -x)
  + Logical negation (!x)
  + Bitwise negation (~x)
  + Asynchronous wait (waitfor funcAsync x)
* Function invocation (% f), power operator e^x, infix operators defined by user
* Multiplicative operators group (x\*y, x / y, x % y)
* Additive operators group (a + b, a – b)
* Shifts (x << y, x >> y)
* Relation (x < y, x > y, x <= y, x >= y)
* Type cheking (x as T, x is T)
* Equality (x == y, x != y)
* Logical operations (x & y, x ^ y, x | y)
* Conditional operators ( x && y, x || y)
* Exception value statement (x or y)
* Ternary ( x ? y : z)
* Assignement (x = y, x+=y, x-=y, x\*=y, x/=y)
* Pattern matching expression (match expr with BLOCK)
* Lambda expression ( x -> f(x) )
* Function invocation (f x y z)
* Condition expression (if expr1 then expr2 elseif expr3 then expr4 … else exprn+1)
* Let expression (let x = 5 in x + x)

## Statements

Statements used mostly in OO/Pr part of programs. They are computed one by one in the program. There are expression statement, assignment statements, declaration statements, iteration statements, control transfer statements, execution (and exception control) statement, import and module statements.

### Expression statements

See section above for information about expressions.

### Declaration statement

Let statement declares new object in current scope. Const statement declares new constant in the current scope.

let x = 5

const pi = 3.141592653589

### Iteration statements

There is three iteration statements. Conditional iteration provides two statements and collection iteration provides one more:

while x > 5 do x -= 1

do x-=1 while x > 5

for x in L console.print x

### Code transfer statements

Return statement interrupts function execution and returns it’s value:

let prd = x -> do

console.print x

return 2 \* x

Break statement ends an iteration statement; continue moves to the next step of iteration statement it is in:

while x = % console.readline or None do

if x == ‘exit’ then do return 42

elseif x.match /\d\*/ do break

elseif x.match /ZZ/ do

continue

else

console.print ‘No number specified.’

file.append x

### Execution and exception handling statement

Do statement may or may not have except section to catch an exception. Also it may or may not contain final section.

let main = args -> do

....let f = file.open “file.txt”, “w”

....for x in args do

......f.write x ‘\n’

..except e

....console.print ‘Error: ‘ e.message

..finally

....% f.close

....return 0

### Import and Module statement

Import statement adds all public objects in module imported to the current module.

Module statement selects a name for a module that reside in file. It is mandatory to have one module statement in each file.

module hello

import lib.io

## Types

Types were already discussed in *type system* paragraph. Let’s just add that interface type may only contain method and operator signatures and they are always public.

## Classes

Classes are one of the most fundamental types in Que. In contrast with type, which only contain signatures, class type may contain both properties and methods along with operators.

Class may contain signatures as well as actual function values. If there is at least one method present without it’s body, then that class is called abstract and cannot be instantiated.

Each entity in class may have one of four visibility levels:

* Private entities are visible only in class they were declared;
* Protected entities are visible in class they were declared and in its descendants;
* Internal entities are visible inside module class were declared in;
* Public entities are visible everywhere.

If none specified, field stays internal.

# Compiler functional specification

Que compiler is a program that converts several Que source files into an executable or a library. It consists of the following five parts:

1. **Lexer (or lexical analyzer)** converts a sequence of characters from the file to a sequence of tokens. Tokens represent atomic entities of Que language: constants, identifiers, different types of brackets, block delimiters.
2. **Syntax analyzer** gets a sequence of tokens provided by a lexer procedure and converts them into the *parse tree* – a structural representation of the que program. Most of syntax errors are caught in those two phases. A parse tree along with the additional information is often called an Intermediate Representation of Que program.
3. **Semantic analyzer** adds additional information into IR – the results of type computation and checking, symbol table, etc.
4. **F-Optimizer** tries to change an IR so the result would be faster and smaller. It performs a number of conversions aimed at increasing the resulting code speed or size.
5. **Code generator**, as it follows from name, generates a binary code of program. The result may be executed on the target machine.

In the following part of this book, aspects of Que compilation will be explained.

## Lexical analyzer

Lexical analyzer gets a sequence of characters (from file, for example) and converts them into sequence of tokens.

Que lexer parses a program line by line. This is to get an indentation information from a file.

### Indentation parsing

From each line, the amount of leading whitespace characters is computed. Each time user increases that amount, compiler sees it and throws new amount into a stack and yields the BeginBlock token. When user decreases an amount of leading whitespace characters, compiler throws away a top value of indentation stack (yielding an BlockEnd token) until it becomes less or equal to current value. If it is equal, program continues. If not, an indentation error is thrown.

### Token types