Quark Language Reference Manual

Name	**UNI**
Daria Jung	djj2115
Jamis Johnson	jmj2180
Jim Fan	lf2422
Parthiban Loganathan	pl2487

Table of contents

Introduction

This is the language reference manual for Quark, a high level language for quantum computing. Quark makes it simple to perform complicated mathematical operations required for quantum computing in a simple and intuitive manner. In addition to standard types featured in most common programming languages, Quark supports complex numbers, fractions, matrices and quantum registers.

Reference Manual

Grammar Notation

FILL OUT LATER

Lexical Conventions

A program in QUARK includes at least one function definition, though something trivial like a variable declaration or a string should compile. Programs are written using a basic source character set accepted by the C++ compiler in use. Refer to what source-code file encoding your compiler accepts. The QUARK compiler will only output ASCII.

Comments

MATLAB style commenting is supported. A MATLAB style comment begins with `%` and ends with `%`. Multi-line MATLAB comments start with `%{` and end with `}%`. Any sequence of characters can appear inside of a comment except the string `}%`. These comments do not nest.

Whitespace

Whitespace is defined as the ASCII space, horizontal tab and form feed characters, as well as line terminators and comments.

Tokens

Tokens in QUARK consist of identifiers, keywords, constants, and separators. Whitespace is ignored and not taken into consideration.

Identifiers

An identifier is composed of a sequence of letters and digits, the first of which must be a letter. There is no limit on the length of an identifier. The underscore character `_` is included in the regular expression pattern for letters.

Two identifiers are the same if they have the same ASCII character for every letter and digit.

```
digit -> ['0'-'9']
letter -> ['a'-'z' 'A'-'Z' '_']
Identifier -> letter (letter | digit)*
```

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

> def
> bool
> int
> float
> fraction
> complex
> qreg
> void
> mod
> in
> return
> continue
> break
> while

> for
> else
> if
> and
> not
> or

Reserved Prefix

There is only one reserved prefix in QUARK:

`_QUARK_`

Punctuation

Parenthesis – Expressions can include expressions inside parenthesis. Parenthesis can also indicate a function call.

Braces – Braces indicate a block of statements.

Semicolon – Semicolons are used at the end of every statement as a terminator. Semicolons are also used to separate rows in the matrix data type.

Colon – Colons are used to denote slicing in arrays and within a function declaration. In a function declaration, formal arguments appear between the colon and a left curly brace.

Dollar Sign – The dollar sign separates the numerator value from the denominator value in a fraction data type.

Comma – Commas have several use cases. Commas are used to separate formal arguments in a function declaration, elements in arrays and matrices, and the size and initial state of a `qreg` .

Escape Sequences

Certain characters within strings need to be preceded by a backslash. These characters and the sequences to produce them in a string are:

Character	Sequence
\”	“
\n	linefeed
\r	carriage return
\t	horizontal tabulation
\b	backspace

Data Types

The data types available in QUARK are:

> int
> float
> fraction
> bool
> complex
> string
> qreg
> matrix
> void

Additionally, the aggregate data type of array is available to the user.

int

An `int` is a 64-bit signed integer.

float

A `float` is a 64-bit signed floating-point number.

fraction

A `fraction` is denoted by two `int` types separated by `$`. The `int` value to the left of `$` represents the numerator, and the `int` value to the right of `$` represents the denominator. QUARK provides an inverse operator `~`.

```
frac foo = 2$3; % represents 4/6
~foo; % 3$2
```

Note on int, float, and fraction

Fraction types may be compare to int and float types using the following comparators: `<`, `>`, `<=`, `>=`, `!=`, and `==`.

```
int i = 3;
float ft = 2.0;
frac f = 2$3;

i > f;
ft <= f;
```

bool

A `bool` value is denoted using the literals `true` or `false`.

complex

A `complex` type is generated from two `int` or `float` values; if given a mix of `int` and `float` types, QUARK will implicitly type cast. A `complex` type can also be generated with one numerical value, which will be assigned to the real part of a complex number; imaginary will default to 0. The real and imaginary parts of a complex number can be accessed by `real` and `im` accessors.

```
complex cnum = i(3.0, 1);
real(cnum); % 3.0
im(cnum); % 1
complex cnum2 = i(9) % this gives us i(9, 0)
```

string

A `string` is a sequence of characters. String literals are placed between double quotations. QUARK supports string lexicographic comparison using `<` , `>` , `<=` , `>=` , `!=` , and `==` .

qreg

A `qreg` type represents a quantum register. A `qreg` accepts two `int` types. The left value denotes the initial size of a quantum register, and the right value denotes the initial bit.

```
qreg q = <| 1, 1 |>;
```

Qreg must be passed as an LValue to any function:

```
% disallowed
hadamard(<|10, 3|>);

% allowed
qreg q = <|10, 3|>;
hadamard(q);
```

Additionally, qreg values may be measured using the destructive `?` operator and the non-destructive (but unrealistic) `?'` operator; the `?` and `?'` operators may only operate on an LValue of type qreg.

```
q ? [2:10];  % measures qubit 2 to 10
```

matrix

QUARK allows you to create matrices; a `matrix` uses a special bracket notation to distinguish from arrays, and rows are separated by semicolons. Matrices may be composed of only `int` , `float` , or `complex` . Matrix elements may be accessed with a square bracket notation by separating the column and row index numbers by commas.

QUARK provides the prime operator `'` for matrix transposition.

```
float[[]] mat = [| 1.2, 3.4; 5.6, 7.8 |];
mat[2, 1];

mat'; % transpose matrix
```

array

QUARK allows arrays of any of the above data types. Arrays are of variable length and are arbitrarily dimensional.

Arrays can be initialized using a comma-separated list delimited by square brackets [ ]. Additionally, arrays can be declared with a size to create an array of uninitialized elements.

Arrays may be concatenated with the `&` operator as long as there is a dimension and type match.

```
int[5]; % gives us [0,0,0,0,0]
int[] a = [1, 2, 3]; % array initialization
int[][] b = [[1,2,3], [4,5,6]]; % 2-d array

[11, 22, 33] & int[3]
% gives us [11, 22, 33, 0, 0, 0]
```

Array indices can be accessed using the square bracket notation with an integer such as:

```
int[] arr = [0, 1, 2];
arr[0];
```

or

```
int[] arr = [0, 1, 2];
int i = 0;
arr[i];
```

Indices of multidimensional arrays may be accessed by separating the dimensional index numbers by commas:

```
int[][] arr = [[0,1,2],[3,4,5]]
arr[1][1]; % accesses 4
```

The built-in `len` function returns an `int` representing the length of the array.

Membership may be tested using the keyword `in`.

```
int x = 5;
```

```
if x in [1:10]:
    % statement here is executed
```

void

Void is a type for a function that returns normally, but does not provide a result value to the caller.

Function types

Functions take in zero or more variables of primitive or array types and optionally return a variable of primitive or array type. A function declaration always begins with `def`, the return type of the function, a colon `:`, and a list of formal parameters which may be empty.

```
def void main: int x
{
    % statement
}
```

Declarations

Declaring a Variable

Variables can be defined within individual functions or in the global scope. Variables may be declared and then defined, or declared and defined simultaneously. An expression to which a value may be assigned is called an LValue.

```
int x; % definition
x = 5; % declaration
int y = 6; % definition and declaration
```

x and y are LValues. LValues are named as such because they can appear on the left side of an assignment (though they may also appear on the right side).

Declaring an Array

As previously shown, arrays can be multidimensional, and may be of variable length. Arrays may be declared on their own with a size to get an uninitialized array of the given size. They can also be initialized with values upon declaration.

```
int[5]; % gives us [0,0,0,0,0]
int[] a = [1, 2, 3]; % array initialization
int[][] b = [[1,2,3], [4,5,6]]; % 2-d array
```

Declaring a Matrix

A matrix declaration uses the special notation of piped square brackets. Matrix rows are distinguished using the `;` separator between elements of rows. Initializing an empty complex matrix initializes an all-zero 3-by-4 complex matrix.

```
float[|] floatmat = [| 1.2, 3.4; 5.6, 7.8 |];
complex[|] mat; % this gives us complex[| 3, 4 |]
```

Operators

Arithmetic

Operator	
+	addition
-	subtraction
++	unary increment by one
--	unary decrement by one
/	division
*	multiplication

`mod`	modulo
`**`	power

Concatenation

Operator	
`&`	String and array concatenation

Assignment

Operator	
`=`	assigns value or right hand side to left hand side
`+=`	addition assignment
`-=`	subtraction assignment
`*=`	multiplication assignment
`/=`	division assignment
`&=`	bitand assignment

Assignment has right to left precedence.

Logical

Operator	
`!=`	not equal to
`==`	equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than

`<=`	less than or equal to
`and`	unary and
`or`	unary or
`not`	unary not

Bitwise Logical / Unary

Operator	
`~`	Bitwise not and fraction inversion
`&`	Bitwise and
`^`	Bitwise xor
`|`	Bitwise or
`<<`	Bitwise left shift
`>>`	Bitwise right shift

Quantum

Operator	
`?`	quantum measurement query (destructive)
`?'`	quantum measurement query (non-destructive)

The `?` and `?'` operators may only be invoked on an LValue.

```
q ? [2:10];  % measures qubit 2 to 10
```

Ternary Operator

QUARK supports Python style ternary conditional operators:

```
3 if true else 5;

4 if 3 > 2 if not (1==1) else 3 < 2 else -2 if true else 3
```

Ternary operators are right associative.

Operator Precedence and Associativity

Operator	**Associativity**
`* / mod`	left
`+ -`	left
`>> <<`	left
`> >= < <=`	left
`== !=`	left
`&`	left
`^`	left
`|`	left
`and`	left
`or`	left
`?`	right
`in`	left
`= += -= *= /= &=`	right

Operators within the same row share the same precedence. Higher rows indicate higher precedence.

Statements

Statements are the smallest components of a program used to express that an action is to be carried out.

Statements are used for variable declarations and assignment, control flow, loops, function calls, and expressions. All statements end with a semicolon `;`. Statements are used within blocks. The following are examples of statements and are by no means exhaustive:

```
string hello = "hello world";
int x = 10;
if x > 5
foo(4);
while x != true
for x in [1:10]
4 + 6
qreg q0 = <| nbit * 2, 0 |>;
```

Blocks

A block is defined to be inside curly braces `{ }`, which may include empty statements and variable declarations.

A block looks like:

```
{
    % statements here
}
```

Return Statement

The return keyword accepts an expression, and exits out of the nearest calling block or smallest containing function.

If else Statement

If statements take expressions that reduce to a boolean, and followed by a colon `:` and a statement block. If the following statement is only one line, curly braces are unnecessary.

```
if p == 1:
    return a;

if (3 > 1):
{
    % multiple statements
}
```

While Loop

A while loop is of the form:

```
while(condition):
{
    % statement
}
```

As with `if else` statements, if the following statement is only one line, curly braces `{}` are unnecessary.

```
while exp_mod(b, i, M) != 1:
    i ++;
```

The condition of the while loop may not be empty.

For Statement

QUARK supports two types of iterators, array and range, for its for statements.

Array Iterator

An array iterator allows you to sweep a variable across an array, evaluating the inner statement with identifiers assigned to a new value before each iteration. The identifier after `for` is assigned to the value of each element of the array sequentially.

The identifier may be declared ahead of time or within the for statement itself.

```
int[] arr = [1,2,3];
for int i in arr:
    print i;

% 1
% 2
% 3
```

Range Iterator

A range iterator allows you to sweep a variable across an array, evaluating the inner statement with identifiers assigned to a new value before each iteration. The identifier after `for` is assigned to each integer in the range.

The identifier may be declared ahead of time or within the for statement itself.

```
int i;
for i in [1:10]
for int i in [1:10:2]
```

A range consists of three integers separated by colons `[start : stop : step]`. Start denotes the start of the range, stop denotes the exclusive end of a range, and step denotes the step size of the range. If the step and the last colon is excluded, the step is defaulted to 1. If the start value is excluded, it is defaulted to 0.

The following are various ways of declaring ranges:

```
0:5:2 % this gives us 0, 2, 4
:5 % 0, 1, 2, 3, 4
1:3 % 1, 2
```

Break and Continue

The `break` statement causes a while loop or for loop to terminate.

The `continue` statement provides a way to jump back to the top of a loop earlier than normal; it may be used to bypass the remainder of a loop for an iteration.

Functions

QUARK allows users to define functions.

Function Declaration

Functions are composed of the form:

```
def return_type func_name: type arg1, type arg2
{
    % statements in function body

    return return_type
}
```

Functions are defined only by identifying the block of code and the keyword `def`, giving the function a name, supplying it with zero or more formal arguments, and defining a function body. Function return types are of any data type previously described, or `void` for no value.

Some examples of function declarations are:

```
def void hello:
{
    print("hello world");
}

def int addition: int x, int y
{
    return x + y;
}
```

Casting

QUARK does not allow explicit type casting.

Overloading

QUARK keeps separate symbol tables for functions and types, and as such the same identifier may be used as a variable and a function at the same time.

Built-in functions are overridable because they are not stored in the function table, with the exception of the following built-in functions:

> print (with \n)
> print_noline
> apply_oracle

Otherwise, function overloading by itself is not supported.

Scoping

In QUARK, there are both global and local scopes. QUARK uses block scoping. A block is a section of code contained by a function, a conditional (if/while), or looping construct (for). Variables defined in the global scope can be used in any function, as well as in any block within that function. Each nested block creates a new scope, and variables declared in the new scope supersede variables declared in higher scopes.

Below is an example of the subtleties of QUARK's scoping rules:

```
int i = 1000;
for i in [0:7:2]:
{
    % statement
}

% i changed.

int i = 1000;
for int i in [0:7]:
{
    % statement
}

i; % still 1000
```

Grammar

Below is the grammar for QUARK. Words in capital letters are tokens passed in from the lexer.

```
ident:
    ID { Ident($1) }

vartype:
    INT      { Int }
  | FLOAT    { Float }
  | BOOLEAN  { Bool }
  | STRING   { String }
  | QREG     { Qreg }
  | FRACTION { Fraction }
  | COMPLEX  { Complex }
  | VOID     { Void }

datatype:
  | vartype { DataType($1) }
  | datatype LSQUARE RSQUARE { ArrayType($1) }
  | datatype LMATRIX RSQUARE { MatrixType($1) } /* int[|] */

/* Variables that can be assigned a value */
lvalue:
  | ident                          { Variable($1) }
  | ident LSQUARE expr_list RSQUARE
      { ArrayElem($1, $3)}

expr:
  /* Logical */
  | expr LT expr           { Binop($1, Less, $3) }
  | expr LTE expr          { Binop($1, LessEq, $3) }
  | expr GT expr           { Binop($1, Greater, $3) }
  | expr GTE expr          { Binop($1, GreaterEq, $3) }
  | expr EQUALS expr       { Binop($1, Eq, $3) }
  | expr NOT_EQUALS expr   { Binop($1, NotEq, $3) }
  | expr AND expr          { Binop($1, And, $3) }
  | expr OR expr          { Binop($1, Or, $3) }

  /* Unary */
  | BITNOT expr              { Unop(BitNot, $2) }
  | MINUS expr %prec UMINUS { Unop(Neg, $2) }
  | NOT expr                 { Unop(Not, $2) }

  /* Arithmetic */
  | expr PLUS expr    { Binop($1, Add, $3) }
  | expr MINUS expr   { Binop($1, Sub, $3) }
  | expr TIMES expr   { Binop($1, Mul, $3) }
  | expr DIVIDE expr  { Binop($1, Div, $3) }
  | expr MODULO expr  { Binop($1, Mod, $3) }
  | expr POWER expr   { Binop($1, Pow, $3) }

  /* Bitwise */
  | expr BITAND expr        { Binop($1, BitAnd, $3) }
  | expr BITXOR expr        { Binop($1, BitXor, $3) }
  | expr BITOR expr         { Binop($1, BitOr, $3) }
  | expr LSHIFT expr        { Binop($1, Lshift, $3) }
  | expr RSHIFT expr        { Binop($1, Rshift, $3) }

  /* Query */
  | expr QUERY expr         { Queryop($1, Query, $3,
IntLit("QuerySingleBit")) }
  | expr QUERY_UNREAL expr  { Queryop($1, QueryUnreal, $3,
IntLit("QuerySingleBit")) }
  | expr QUERY LSQUARE COLON expr RSQUARE { Queryop($1, Query, IntLit("0"),
$5) }
  | expr QUERY_UNREAL LSQUARE COLON expr RSQUARE  { Queryop($1, QueryUnreal,
IntLit("0"), $5) }
  | expr QUERY LSQUARE expr COLON expr RSQUARE         { Queryop($1, Query,
$4, $6) }
  | expr QUERY_UNREAL LSQUARE expr COLON expr RSQUARE  { Queryop($1,
QueryUnreal, $4, $6) }

  /* Parenthesis */
  | LPAREN expr RPAREN { $2 }

  /* Assignment */
  | lvalue ASSIGN expr { Assign($1, $3) }
  | lvalue             { Lval($1) }

  /* Special assignment */
  | lvalue PLUS_EQUALS expr { AssignOp($1, AddEq, $3) }
  | lvalue MINUS_EQUALS expr { AssignOp($1, SubEq, $3) }
  | lvalue TIMES_EQUALS expr { AssignOp($1, MulEq, $3) }
  | lvalue DIVIDE_EQUALS expr { AssignOp($1, DivEq, $3) }
  | lvalue BITAND_EQUALS expr { AssignOp($1, BitAndEq, $3) }

  /* Post operation */
  | lvalue INCREMENT { PostOp($1, Inc) }
  | lvalue DECREMENT { PostOp($1, Dec) }

  /* Membership testing with keyword 'in' */
  | expr IN expr    { Membership($1, $3) }

  /* literals */
  | INT_LITERAL                                 { IntLit($1) }
  | FLOAT_LITERAL                               { FloatLit($1) }
  | BOOLEAN_LITERAL                             { BoolLit($1) }
  | expr DOLLAR expr                            { FractionLit($1, $3) }
  | STRING_LITERAL                              { StringLit($1) }
  | LSQUARE expr_list RSQUARE                   { ArrayLit($2) }
  | datatype LSQUARE expr RSQUARE               { ArrayCtor($1, $3) }
  | LMATRIX matrix_row_list RMATRIX             { MatrixLit($2) }
  | datatype LMATRIX expr COMMA expr RMATRIX { MatrixCtor($1, $3, $5) }
  | COMPLEX_SYM expr COMMA expr RPAREN          { ComplexLit($2, $4) }
  | COMPLEX_SYM expr RPAREN                     { ComplexLit($2,
FloatLit("0.0")) }
  | LQREG expr COMMA expr RQREG                 { QRegLit($2, $4) }

  /* function call */
  | ident LPAREN RPAREN             { FunctionCall($1, []) }
  | ident LPAREN expr_list RPAREN   { FunctionCall ($1, $3) }

expr_list:
  | expr COMMA expr_list { $1 :: $3 }
  | expr                 { [$1] }

/* [| r00, r01; r10, r11; r20, r21 |] */
matrix_row_list:
  | expr_list SEMICOLON matrix_row_list { $1 :: $3 }
  | expr_list            { [$1] }

decl:
  | datatype ident ASSIGN expr SEMICOLON                { AssigningDecl($1,
$2, $4) }
  | datatype ident SEMICOLON                            { PrimitiveDecl($1,
$2) }

statement:
  | IF expr COLON statement ELSE statement
      { IfStatement($2, $4, $6) }
  | IF expr COLON statement %prec IFX
      { IfStatement($2, $4, EmptyStatement) }

  | WHILE expr COLON statement { WhileStatement($2, $4) }
  | FOR iterator COLON statement { ForStatement($2, $4) }

  | LCURLY statement_seq RCURLY { CompoundStatement($2) }

  | expr SEMICOLON { Expression($1) }
  | SEMICOLON { EmptyStatement }
  | decl { Declaration($1) }

  | RETURN expr SEMICOLON { ReturnStatement($2) }
  | RETURN SEMICOLON { VoidReturnStatement }

  /* Control flow */
  | BREAK { BreakStatement }
  | CONTINUE { ContinueStatement }

iterator:
  | ident IN LSQUARE range RSQUARE { RangeIterator(NoneType, $1, $4) }
  | datatype ident IN LSQUARE range RSQUARE { RangeIterator($1, $2, $5) }
  | datatype ident IN expr { ArrayIterator($1, $2, $4) }

range:
  | expr COLON expr COLON expr { Range($1, $3, $5) }
  | expr COLON expr { Range($1, $3, IntLit("1")) }
  | COLON expr COLON expr { Range(IntLit("0"), $2, $4) }
  | COLON expr { Range(IntLit("0"), $2, IntLit("1")) }

top_level_statement:
  | DEF datatype ident COLON param_list LCURLY statement_seq RCURLY
      { FunctionDecl($2, $3, $5, $7) }
  | datatype ident COLON param_list SEMICOLON
      { ForwardDecl($1, $2, $4) }
  | decl { Declaration($1) }

param:
  | datatype ident { PrimitiveDecl($1, $2) }

non_empty_param_list:
  | param COMMA non_empty_param_list { $1 :: $3 }
  | param { [$1] }
```

```
param_list:
  | non_empty_param_list { $1 }
  | { [] }

top_level:
  | top_level_statement top_level {$1 :: $2}
  | top_level_statement { [$1] }

statement_seq:
  | statement statement_seq {$1 :: $2 }
  | { [] }
```