QLang: Qubit Language
(Reference Manual)

Christopher Campbell Clément Canonne Sankalpa Khadka Winnie Narang
Jonathan Wong

October 27, 2014

Contents

1 Introduction

2 Lexical conventions

There are five kinds of tokens in the language, namely (1) identifiers, (2) keywords, (3) constants, (4) expression operators, and (5) other separators. At a given point in the parsing, the next token is chosen as to include the longest possible string of characters forming a token.

2.1 Character set

`QLang` supports a subset of ASCII; that is, allowed characters are `a-zA-Z0-9@#,-_;:()[]{}<>=+/|*`, as well as tabulations `\t`, spaces, and line returns `\n` and \r.

2.2 Comments

Comments start with a # sign, which then extends until the next carriage return. Multiline comments are not supported.

2.3 Identifier (names)

An identifier is an arbitrarily long sequence of alphabetic and numeric characters, where `_` is included as "alphabetic". It must start with a lowercase or uppercase letter, i.e. one of `a-zA-Z`.
The language is case-sensitive: `hullabaloo` and `hullABaLoo` are considered as different.

2.4 Keywords

The following identifiers as reserved for keywords, and no one shall use them because it's forbidden and uncool.

```
pi e
int float comp rvect cvect mat
true false
if elif else
def for from to by while break
or and xor
not re im norm isunit trans det adj conj sin cos tan
```

2.5 Constants

There are fours sorts of constants in the language, namely *integer*, *float*, *complex* and *identifier* constants. The first are comprised of any sequence of integers of the form `0|([1-9][0-9]*)` (recall that integers are non-negative), and have type `int`. The second are of type `float` and have the form R, while the third are of type `com` and have the form R`|`R`+`R`i|`R`i` where R consists of a (i) sign, (ii) an integer part followed by (iii) a point, (iv) a decimal part, then (v) either a `e` or a `E` followed by an exponent part, possibly signed. (i) and (v) are optional, and either (ii) or (iv) can be missing as well. In more detail, R is defined as `[+-]{0,1}(((A.B*|.B+)([eE][+-]?B+)?)|A[eE][+-]?B+)` and A =`0|([1-9]B*)`, B =`0|[1-9]` (that is, R matches a real number such as `2.78e5`, `1.5E-1` or `10.25`).

check this paragraph.

Finally, the identifier constants are a subset of the reserved keywords, and include:

`e` the base of natural logarithm $e = \sum_{k=0}^{\infty} \frac{1}{k!}$. Equivalent to `exp(1)`; has type `com`.

`Pi` the constant π. Has type `com`.

`true` represents the Boolean value true. Stored internally as `int` 1.

`false` represents the Boolean value false. Stored internally as `int` 0.

3 Syntax notation

An operation, or language elementary unit, starts from the end of the previous one, and ends whenever a semicolon (that is not part of a matrix declaration) is encountered.

4 Objects and lvalues

5 Conversions

6 Expressions

6.1 Operator Precedence

Operator Type	Operator	Associativity
Primary Expressions	() [] <| |>	Left
Unary	not re im norm unit trans det adj conj sin cos tan	Right
Binary	* / % + - @ eq lt gt leq geq or and xor ^	Left (except ^ which is Right)
Assignment	=	Left

6.2 Literals

Literals are integers, floats, complex numbers, qubits, and matrices, as well as the built-in constants of the language (e.g. Pi). Integers are of type `int`, floats are of type `float`, complex numbers are of type `com`, qubits are of type `qub`, and matrices are of type `mat`. The built-in constants have pre-determined types described above (e.g. Pi is of type `float`).

The remaining major subsections of this section describe the groups of *expression* operators, while the minor subsections describe the individual operators within a group.

6.3 Primary Expressions

6.3.1 identifier

Identifiers are primary *expressions*. All identifiers have an associated type that is given to them upon declaration (e.g. `float` *ident* declares an identifier named ident that is of type `float`).

6.3.2 literals

Literals are primary *expressions*. They are described above.

6.3.3 (*expression*)

Parenthesized *expressions* are primary *expressions*. The type and value of a parenthesized *expression* is the same as the type and value of the *expression* without parenthesis. Parentheses allow *expressions* to be evaluated in a desired precedence. Parenthesized *expressions* are evaluated relative to each other starting with the *expression* that is nested the most deeply and ending with the *expression* that is nested the least deeply (i.e. the shallowest).

6.3.4 primary-*expression*(*expression*-list)

Primary *expressions* followed by a parenthesized *expression* list are primary *expressions*. Such primary *expressions* can be used in the declaration of functions or function calls. The *expression* list must consist of one or more *expressions* separated by commas. If being used in function declarations, they must be preceded by the correct function declaration syntax and each *expression* in the *expression* list must evaluate to a type followed by an identifier. If being used in function calls each *expression* in the *expression* list must evaluate to an identifier.

6.3.5 [*expression*-elementlist]

Expression element lists in brackets are primary *expressions*. Such primary *expressions* are used to define matrices and therefore are of type `mat`. The *expression* element list must consist of one or more *expressions* separated by commas or semi-colons. Commas separate *expressions* into matrix columns and semi-colons separate *expressions* into matrix rows. The *expressions* must evaluate to the same type and can be of type `int`, `float`, `com`, or `mat`. Additionally, the number of *expressions* in each row of the matrix must be the same. An example matrix is shown below.

```
int a = 3;
int b = 12;
mat my_matrix = [ 0+1, 2, a; 5-1, 2*3-1, 12/2];
```

6.3.6 <*expression*|

Expressions with a less than sign on the left and a bar on the right are primary *expressions*. Such *expressions* are used to define qubits and therefore are of type `qub`. The notation is meant to mimic the "bra-" of "bra-ket" notation and can therefore be thought of as a row vector representation of the given qubit. Following "bra-ket" notation, the *expression* must evaluate to an integer literal of only 0's and 1's, which represents the state of the qubit. An example "bra-" qubit is shown below.

```
qub b_qubit = <0100|;
```

6.3.7 |*expression*>

Expressions with a bar on the left and a greater than sign on the right are primary *expressions*. All of the considerations are the same as for <*expression*|, except that this notation mimics the "ket" of "bra-ket" notation and can therefore be though of as a column vector representation of the given qubit. An example "ket-" qubit is shown below.

```
int a = 001;
qub k_qubit = |a>;
```

6.4 Unary Operators

6.4.1 not *expression*

The result is a Boolean indicating the logical `not` of the *expression*. The type of the *expression* must be `int` or `float`. In the *expressions*, 0 is considered false and all other values are considered true.

6.4.2 re *expression*

The result is the real component of the *expression*. The type of the *expression* must be `com`. The result has the same type as the *expression* (it is a complex number with 0 imaginary component).

6.4.3 im *expression*

The result is the imaginary component of the *expression*. The type of the *expression* must be `com`. The result has the same type as the *expression* (it is a complex number with 0 real component).

6.4.4 norm *expression*

The result is the norm of the *expression*. The type of the *expression* must be `mat`, `com`, `qub` or `float`. The result has type `float`, and corresponds to the 2-norm; in the case of `com` or `float`, this coincides with respectively the module and absolute value.

6.4.5 isunit *expression*

The result is a Boolean indicating if it is true or false that the *expression* is a unit matrix. The type of the *expression* must be `mat`.

6.4.6 trans *expression*

The result is the transpose of the *expression*. The type of the *expression* must be `mat`. The result has the same type as the *expression*.

6.4.7 det *expression*

The result is the determinant of the *expression*. The type of the *expression* must be `mat`. The result has type `float` if the *expression* is an integer matrix or `float` matrix and type `com` if the *expression* is a complex number matrix.

6.4.8 adj *expression*

The result is the adjoint of the *expression*. The type of the *expression* must be `mat`. The result has the same type as the *expression*.

6.4.9 conj *expression*

The result is the complex conjugate of the *expression*. The type of the *expression* must be `com` or `mat`. The result has the same type as the *expression*.

6.4.10 sin *expression*

The result is the evaluation of the trigonometric function sine on the *expression*. The type of the *expression* must be `int`, `float`, or `com`. The result has type `float` if the *expression* is of type `int` or `float` and type `com` if the *expression* is of type `com`.

6.4.11 cos *expression*

The result is the evaluation of the trigonometric function cosine on the *expression*. The type of the *expression* must be `int`, `float`, or `com`. The result has type `float` if the *expression* is of type `int` or `float` and type `com` if the *expression* is of type `com`.

6.4.12 tan *expression*

The result is the evaluation of the trigonometric function tangent on the *expression*. The type of the *expression* must be `int`, `float`, or `com`. The result has type `float` if the *expression* is of type `int` or `float` and type `com` if the *expression* is of type `com`. (If an error occured because of a division by zero, a runtime exception is raised.)

6.5 Binary Operators

6.5.1 *expression* ˆ *expression*

The result is the exponentiation of the first *expression* by the second *expression*. The types of the *expression* must be of type `int`, `float`, or `com`. If the *expressions* are of the same type, the result has the same type as the *expressions*. Otherwise, if at least one *expression* is a `com`, the result is of type `com`; if neither *expressions* are comp, but at least one is `float`, the result is of type `float`.

6.5.2 *expression* * *expression*

The result is the product of the *expressions*. The type considerations are the same as they are for *expression* ˆ *expression*

6.5.3 *expression* / *expression*

The result is the quotient of the *expressions*, where the first *expression* is the dividend and the second is the divisor. The type considerations are the same as they are for *expression* ˆ *expression*. Integer division is rounded towards 0 and truncated. (If an error occured because of a division by zero, a runtime exception is raised.)

6.5.4 *expression* % *expression*

The result is the remainder of the division of the *expressions*, where the first *expression* is the dividend and the second is the divisor. The sign of the dividend and the divisor are ignored, so the result returned is always the remainder of the absolute value (or module) of the dividend divided by the absolute value of the divisor. The type considerations are the same as they are for *expression*ˆ *expression*.

6.5.5 *expression* + *expression*

The result is the sum of the *expressions*. The types of the *expressions* must be of type `int`, `float`, `com`, `mat` or `qub`. If at least one *expression* is a `com`, the result is of type `com`; if neither *expressions* are comp, but at least one is `float`, the result is of type `float`. Qubits and matrices are special and can only be summed with within operands of the same type (and, in the case of matrices, dimensions).

6.5.6 *expression* - *expression*

The result is the difference of the first and second *expression*. The type considerations are the same as they are for *expression* + *expression*.

6.5.7 *expression* @ *expression*

The result is the tensor product of the first and second *expressions*. The *expressions* must be of type of `mat`. The result has the same type as the *expression*.

6.5.8 *expression* eq *expression*

The result is a Boolean indicating if it is true or false that the two *expression* are structurally equivalent. The type of the *expressions* must be the same.

6.5.9 *expression* lt *expression*

The result is a Boolean indicating if it is true or false that the first *expression* is less than the second. The type of the *expressions* must be `int` or `float` and must be the same.

no ordering for complex numbers

6.5.10 *expression* gt *expression*

The result is a Boolean indicating if it is true or false that the first *expression* is greater than the second. The type of the *expressions* must be `int` or `float` and must be the same.

no ordering for complex numbers

6.5.11 *expression* leq *expression*

The result is a Boolean indicating if it is true or false that the first *expression* is less than or equal to the second. The type of the *expressions* must be `int` or `float` and must be the same.

no ordering for complex numbers

6.5.12 *expression* geq *expression*

The result is a Boolean indicating if it is true or false that the first *expression* is greater than or equal to the second. The type of the *expressions* must be `int` or `float` and must be the same.

no ordering for complex numbers

6.5.13 *expression* or *expression*

The result is a Boolean indicating the logical *or* of the *expressions*. The type of the *expressions* must be `int` or `float` and must be the same. In the *expressions*, 0 is considered `false` and all other values are considered `true`.

6.5.14 *expression* and *expression*

The result is a Boolean indicating the logical *and* of the *expressions*. The type considerations are the same as they are for *expression* or *expression*.

6.5.15 *expression* xor *expression*

The result is a Boolean indicating the logical *xor* of the *expressions*. The type considerations are the same as they are for *expression* or *expression*.

6.6 Assignment Operators

Assignment operators have left associativity

6.6.1 lvalue = *expression*

The result is the assignment of the *expression* to the lvalue. The lvalue must have been previously declared. The type of the *expression* must be of the same that the lvalue was declared as. Recall, lvalues can be declared as `int`, `float`, comp, mat, and qubit.

7 Declarations

Declarations are used within functions to specify how to interpret each identifier. Declarations have the form

declaration:
    type-specifier declarator-list

7.1 Type Specifiers

There are four main type specifiers

type-specifier:
    int
    float
    com
    mat

7.2 Declarator List

The declarator-list field of a declaration is a comma-separated sequence of declarators.

declarator-list:
  declarator
  declarator , declarator-list

Declarators refer to a certain object. That object is of the type indicated by the type-specifier in the declaration. Declarators have the syntax

declarator:
  identifier
  declarator ( )
  declarator [constant-expression]
  (declarator)

The grouping in this definition is the same as in expressions.

7.3 Meaning of Declarators

Each expression that has the same form as a declarator is a call to create an object of the specified type. Each declarator has one identifier. Each identifier is of the type indicated by the specifier.

If declarator D has the form

D ( )

then the contained identifier has the type "function returning ...", where "..." is the type which the identifier would have had if the declarator had been D.

If a declarator has the form

D[constant-expression]

or

D[ ]

then it is a declarator whose identifier is of type "array". In the first case, the constant- expression is an expression whose value is determinable at compile time. The type of that constant-expression is int. In the second case, the constant expression 1 is used.

An array may be constructed from one of the basic types, or from another array.

Parentheses in declarators do not alter the type of the contained identifier, but rather the binding of the individual compoents of the declarator.

Not all possibilities of the above syntax are actually allowed. There are certain further restrictions. There are no array of functions.

8 Statements

9 Scope rules

10 Constant expressions

In order to facilitate efficiency in writing expression, the language introduces various mathematical constants such as π , e and matrices such *Pauli* matrices and *Hadamard* matrices which are frequently used in quantum computation. The keywords *I, X, Y, Z, and H* are reserved for this expressions.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

The *Hadamard gate* is defined by the matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

11 Examples

We present some examples that illustrates the use of Qlang in solving quantum computing problems.

11.1 Solving Quantum Computation Problem

11.1.1 Problem1

Evaluate the following expressions: a. $(H \otimes X)|00\rangle$ b. $\langle 101|000\rangle$ c. $\langle 01|H \otimes H|01\rangle$

```
def pseudo = evaluate (){

        # a quit type declaration follows dirac notation
        qubit mat0 = |00>;

        # Both X and H are constant with type mat and
        # @ corresponds to tensor product.
        mat HX = H @ X;

        pseudo = HX * mat0;
}
```

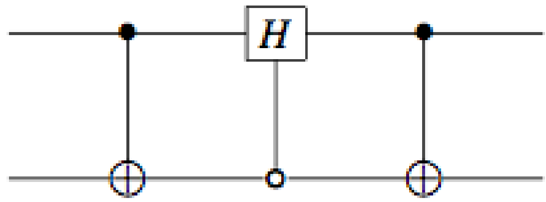

Figure 1: Quantum Circuit implementing series of control gates

11.1.2 Problem 2

Find the matrix corresponding to the quantum circuit:

```
def circuitMat = findMatrix (){

        # all basis qubit in 2 dimension
        qubit mat0=|00>;
        qubit mat1=|01>;
        qubit mat2=|10>;
        qubit mat3=|11>;

        # controlled not matrix
        mat CNOT = [1,0,0,0;0,1,0,0;0,0,0,1;0,0,1,0]

        #controlled hadmard matrix
        mat HNOT = [1/sqrt(2),0,0,1/sqrt(2);0,1,0,0;1/sqrt(2),0,1,-1/sqrt(2);0,0,0
        #composition of control gates
        mat allGates = CNOT * HNOT * CNOT

        # Matrix corresponding to the circuit
        circuitMat =[allGates*mat0:allGates*mat1:allGates*mat2:allGates*mat3]

}
```

11.1.3 Problem 3

Consider the circuit and show the probabilities of outcome 0 where $|\Psi_{in}\rangle = |1\rangle$

```
def probability = outcomeZero(){

        # top and bottom qubits
        qubit top = |0>;
        qubit bottom = |1>;

        # Applying H on top qubit
```

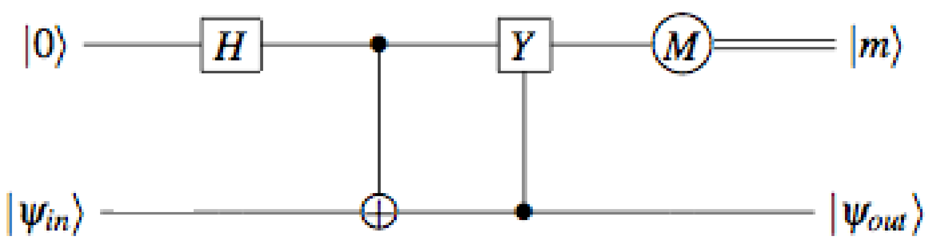

Figure 2: Quantum Circuit

```
        mat output = (H @ I) * (top @ bottom);

        # Controlled Not operator
        mat CNOT = [I, [0,0;0,0]; [0,0;0,0], X];

        # Controlled Y operator
        mat CY = [Y,[0,0;0,0];[0,0;0,0], I];

        # Applying Control Operators
        output = (CY)*(CNOT)*output

        # Applying measurement operator on top qubit |0> <0|
        mat M = (|0>*<0| @ I)

        # state after applying measurement operator on top qubit
        outcome = M * output;

        #probability of outcome
        probability = norm(outcome);

}
```

11.2 Simulation of Quantum Algorithm

11.2.1 Deutsch Jozsa Algorithm

```
def outcome = deutschJozsa(qubit top, mat U){

        # in corresponds to the qubit in top register
        # input is the tensor product of top register and bottom register
        mat input= top @ |1>;

        # application of Hadamard gate on both top and bottom inputs
```

```
        input = (H @ H)*input;

        # application of U gate on the above result
        input = U * input;

        # application of Hadamard gate on the top register
        input = (H @ I)*input;

        # application of measurement operator on the top register
        # top * Adj (top) corresponds to the Measurement operator

        input=(top*Adj(top)@ I)*input;

        #after the measurement is applied, check if the input is 0 or not
        if (input == 0){
                #probability of outcome 0 is 0
                outcome = 0;
        }
        else{
                # probability of outcome 0 is 1
                outcome = 1;
        }
}
```

11.2.2 Grover's Search Algorithm

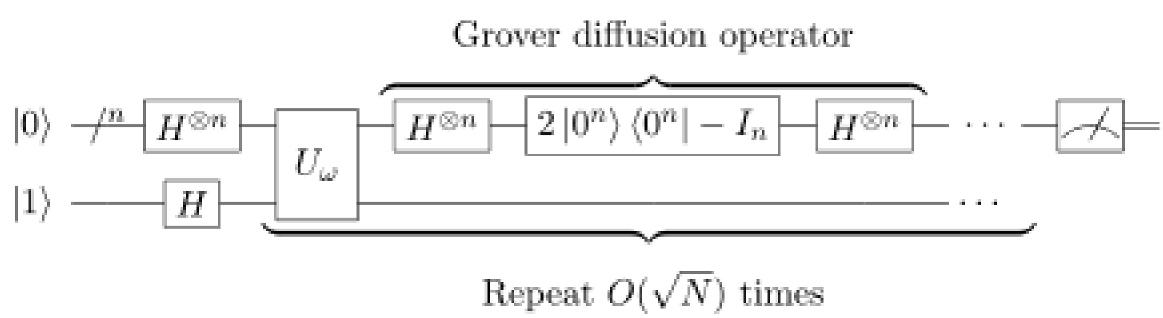

Figure 3: Grover Algorithm Circuit

```
def result = grover (quit top, int x0){
        # returns the probability to find x0 for a function f such that f(x0)=1
        # x0 can be x0=0,1,?,2^n-1
        # this is a special case where n=1

        # qubit in the bottom register
```

```
    qubit bottom = |1>;

    # tensor product of top and bottom qubit
    mat input = top @ bottom;

    #application of Hadamard
    input = (H @ H) * input;

    #define S
    mat S = [1,0;0 -1]

    # k : number of time grover operator is applied
    # for n > 1 k=ceil((pi*2^(n/2))/4);
    int k = 1;

    # define O operator  such that O|x>|q>=|x>|q mod f(x)> or O|x>=(-1)f(x)|x>
    # for n > 1 O = I(2^(n1+1));
    mat O = I;
    O(x0+1, x0+1) = -1;

    # Grover iteration matrix
    mat GO = (G*O)^k;

    # After application of Grover iteration matrix
    mat output = GO * input;

    result = (H @ H)* output;

}
```