Q-Prog

Written by Walter O. Krawec

Introduction

Q-Prog (Quantum Programmer) is an interpreter allowing one to write and then simulate quantum algorithms in a syntax similar to Lisp/Scheme. The document contains information on the syntax and usage of this language. For more information besides what's contained in this document, see the paper "Simulating Quantum Algorithms with Q-Prog".

Building

If you are using Windows, a VC++ project file is included (along with a pre-built binary file). If using another platform, a Makefile is included. Once built, simply run the resulting executable and you should be presented with a prompt: "Q-Prog >". Q-Prog was tested on Windows 7 and Mac OS X.

Syntax

Every program in wok::Interpreter is of the form:

PROGRAM ::= CONSTANT | COMMAND (ARG1) (ARG2) … (ARG_n)

That is to say, a program is either a single word (translated to a constant) or a command followed by a list of arguments. Each argument is actually a program itself (that is, each argument is either a constant or a command followed by its own arguments). The "value of" a PROGRAM is simply the output of that program; it may be an integer, double, string, matrix, complex, space, system, or NULL (meaning 'no

value'). The value of a CONSTANT is that CONSTANT; the value of a COMMAND is the output of that COMMAND given the value of each ARG. (Note that one of the commands supported is the "begin" command which allows you to write a programs that run sequentially).

Q-Prog supports typed variables. Currently supported types are: integer, double, string, complex (two double precision floats), matrix (complex entries), space, and system (the last two will be explained later). Usually a CONSTANT is converted into one of these. If a program consists of only a single word, the value of that program is an integer if it consists only of decimals (or a '-' in front); a double if it is only digits with a single '.' mark; else it is converted to a string. For other types, the proper constructor must be called (e.g. the program "complex 1 2" will return the number 1 + 2i).

Example: The value of the program "123" (without quotes) is 123; the value of "test" is the string "test"; the value of "test 1 2 3" is undefined since "test" is not a command.

When run, Q-Prog should present you with a prompt "Q-Prog >" from which you may write programs (type "exit" (without quotes) and press ENTER to quit). One of the supported commands is "+" (without the quotes). It's defined as follows:

Command: + (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will add Value1 to Value 2
Value: Value1 + Value2

The above block tells us that the value of the program + (VALUE1) (VALUE2) is the integer VALUE1+VALUE2. It also tells us that VALUE1 and VALUE2 must be integer types. At the prompt "Q-Prog >", try the following program (type each line then press ENTER to run the program)

```
+ (1) (2)
```

You should get an integer output equal to "3". Note that, for a CONSTANT, there is no need to include the parenthesis. Hence we could write the above program as:

```
+ 1 2
```

With spaces between the "+" the "1" and the "2". Here are some other programs:

```
+ (+ 1 2) (+ 3 4)
```

```
+ (+ (+1 2) 3) 4
```

Both should evaluate to the integer "10".

One may also create new variables using the "new" command; these variables are manipulated using the "get" and "set" commands:

Command: new (type <STRING>) (name <STRING>)
Description: creates a new variable of the specified type and name.
Value: NULL

Command: get (name <STRING>)
Description: Will return the value of the variable “name”
Value: the value of the variable “name”

Command: set (name <STRING>) (Value <PROGRAM>)
Description: Will set the value of the variable “name” to the value of “Value”. Returns an error if types don’t match.
Value: NULL

However, now we have an issue: if the program is only this single command, how do we use this new variable? We next introduce the “begin” command:

Command: begin (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Will evaluate each argument in sequential order
Value: The value of ARG_n (the last run command)

Also the print command we’ll use in the next example:

Command: print (Value <PROGRAM>)
Description: Will print to the console screen, the value of “Value”
Value: NULL

Here is a simple program:

begin (new integer j) (new integer k) (set j 1) (set k 2) print (+ (get j) (get k))

The value of this program is NULL however the integer “3” is printed to the screen. Two new integer variables are created, one is set to “1”, the other to “2”. The print command will evaluate the program + (get j) (get k) and print it’s value (which is 1+2 = 3). Here is another program:

begin (new integer j) (set j 1) (print j)

The value of this program is also NULL but the string “j” is printed to the screen. That is because the print command prints the value of its argument; its argument is “j” which is a constant whose value is the string “j”. If we changed this command to (print (get j)) then the program would print the integer “1”. One last example:

begin (new integer j) (new integer k) (set j 1) (set k 2) print (+ (get j) (get k)) (100)

The integer “3” is again printed to the screen however now the value of the program is the integer 100 (since the value of a “begin” command is the value of the last command).

One last point we’ll make is that Q-Prog may run in two modes: regular and global. Regular (which is default) will keep variables only within their scope (inside a begin block for example but once that begin block is finished the variable is freed). When running in global mode any variable created is added to a global store; this variable will exist even when the program is finished (so multiple programs may access

the variable). Sometimes it might be useful to create variables in global mode in one program (a setup program), and then have a second program actually use them. Global mode may be accessed using the global command:

Command: global (ARG <PROGRAM>)
Description: Will evaluate the program "ARG" in global mode
Value: value of ARG

For example:

global (new integer j)

will create a new integer "j" on the global store. After running the above program, type "!store" (without the quotes) at the prompt. You should see (Integer, j) printed. Use "get" or "set" to manipulated this variable. To clear the global store, use the "!clear" command. Note that any command with a "!" mark is a "shell" command and cannot be used in a program.

We next look at the "if" command:

Command: if (Test <INTEGER>) (True <PROGRAM>) (False <PROGRAM>)
Description: Will first evaluate "Test"; if this is not "0", will evaluate and return the value of "True"; otherwise will evaluate and return the value of "False"
Value: The value of "True" or "False" depending on the value of "Test"

Command: < (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 < value of Value2; 0 otherwise

Here is an example:

```
begin
        (new integer j) (new integer k)
        (set j 0) (set k 1)
        (if (< (get j) (get k))
                (print ("j is less than k"))
                (print ("j is not less than k"))
        )
```

The value of the program is NULL (since print is NULL); however the string "j is less than k" should be printed to the screen. If you change the value of (set j 0) to (set j 1), the string "j is not less than k" should be printed. Note that you can include a begin command in the True/False arguments of the if command. Also, if you don't need one of the True/False arguments (for instance if you don't need to do anything on account of a True of False condition), you may use the NOP command (which is also useful for writing comments):

Command: NOP (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)

Description: Will ignore each ARG (they are not evaluated) and do nothing.
Value: NULL

Command: nop (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Same as NOP just lower-case depending on user preferences
Value: NULL

```
begin
        (NOP
                This is a comment; anything between the two parentheses is ignored.
        )
        (if 1
                (begin
                        (print ("Always true, value of program is 1+2=3"))
                        (+ 1 2)
                )
                (NOP)
        )
```

Finally, you may type programs in a text editor and run them from the prompt using the "file" command:

Command: file (Filename <STRING>)
Description: Will load in the program located at Filename and evaluate it
Value: value of the program located at Filename

Note: You should place a space between every argument. Within a parenthesis however there should be no spaces between the '(' and your text nor between the ')' and your text. For example use (1) but not (1) or (1).

Quantum Computation

We now provide some examples on how to simulate a quantum algorithm. For more details, see the provided examples (under the "scripts" directory).

To begin, one must create at least one "space". This is done via:

new space (NAME) (DIMENSION)

For example, to create a two dimensional space (e.g. qubits), use (new space Qubit 2) (you don't have to call it "Qubit"). Creating a new space will add "dimension" new kets and bras to the store named: NAME.ket.i and NAME.bra.i for i =0, 1, …, DIMENSION-1. After creating at least one space, you must create a system using the command:

New system (NAME) (NUM_SPACES) (SPACE1) (SPACE2) … (SPACE_n)

Where NUM_SPACES = n and SPACE1 through SPACE_n are the string names of previously created spaces in the store. The result is a Hilbert space consisting of the tensor product of each space. For instance, in the quantum random walk example, we create a coin space of dimension 2 and a position space of dimension 250 as follows:

```
begin
        (new space Coin 2)
        (new space Position 250)
        (new system H 2 Coin Position)
```

From here, one may create a quantum state in this system as a matrix filling in the elements manually (see the matrix, get_element, and set_element commands). Alternatively, one may tensor the appropriate kets as needed. For example, to create the state |0>x|125> from the above system, use:

```
begin
        (new matrix psi)
        (set psi (tensor
                        (get Coin.ket.0)
                        (get Position.ket.125)
                )
        )
```

One may then perform a measurement on the state using the “measure” command (see the Commands section). A quantum operator may be applied simply by using the “mult” or “*” command or by using the “apply_op” command depending on your needs.

Commands

We now list the supported commands in alphabetical order:

Command: + (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will add Value1 to Value 2
Value: Value1 + Value2

Command: - (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will subtract Value2 from Value1
Value: Value1 - Value2

Command: * (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will multiply Value1 to Value 2
Value: Value1 * Value2

Command: / (Value1 <INTEGER>) (Value 2<INTEGER>)

Description: Will compute Value1 / Value2
Value: Value1 / Value2

Command: == (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 is equal to the value of Value2; 0 otherwise

Command: < (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 < value of Value2; 0 otherwise

Command: > (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 > value of Value2; 0 otherwise

Command: and (Arg1 <INTEGER>) (Arg2 <INTEGER>) … (Arg_n <INTEGER>)
Description: Will compute Arg1 & Arg2 & … & Arg_n (integer "and" operation)
Value: the integer Arg1 & Arg2 & … & Arg_n

Command: apply_op (system <SYSTEM>) (index <INTEGER>) (op <MATRIX>) (input_state <MATRIX>)
Description: Will apply the operator "op" to the "index" subspace of "system" (indices start at 0) to the quantum state "input_state". Returns a vector (MATRIX) representing the new state of the system.
Value: See description; <MATRIX>

Command: begin (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Will evaluate each argument in sequential order
Value: The value of ARG_n (the last run command)

Command: case (Test <PROGRAM>) (Case1 <PROGRAM>) (Prog1 <PROGRAM>) … (Case_n <PROGRAM>) (Prog_n <PROGRAM>)
Description: Will evaluate Test and compare it to the value of each "Case" in order; as soon as a match is found, the corresponding Prog is evaluated. If the value of Case_n is the string "default", then Prog_n is always evaluated if none of the other Cases match.
Value: The value of Prog_i if Case_i matches Test; NULL otherwise

Command: complex (Real <STRING>) (Imag <STRING>)
Description: Constructs a new complex type from the given string arguments. Note "Real" and "Imag" are not evaluated, they are taken directly as it is.
Value: A complex number equal to Real + Imag * i

Command: cond (Test1 <INTEGER>) (Prog1 <PROGRAM>) … (Test_n <INTEGER>) (Prog_n <PROGRAM>)
Description: Will evaluate each "Test" in order; if the value of one of the "Test" is not "0" then the corresponding program is evaluated (all other Tests and Progs are ignored).
Value: The value of Proc_i where Test_i is not 0; NULL otherwise

Command: cos (Value <DOUBLE>)
Description: Returns the cosine of the value
Value: cos(Value) <DOUBLE>

Command: double (Value <STRING>)
Description: Constructs a new double from the given string argument. Note "Value" is not evaluated, it is taken directly as it is.
Value: A double equal to the given string (0 if the string is ill-formed)

Command: exp (Value <COMPLEX>)
Description: Computes exp(Value).
Value: exp(Value) <COMPLEX>

Command: file (Filename <STRING>)
Description: Will load in the program located at Filename and evaluate it
Value: value of the program located at Filename

Command: get (name <STRING>)
Description: Will return the value of the variable "name"
Value: the value of the variable "name"

Command: get_element (mat <MATRIX>) (row <INTEGER>) (col <INTEGER>)
Description: Will return the specified element of "mat" (row and col starts at 0)
Value: COMPLEX value; the entry of "mat" specified by "row" and "col".

Command: global (ARG <PROGRAM>)
Description: Will evaluate the program "ARG" in global mode
Value: value of ARG

Command: if (Test <INTEGER>) (True <PROGRAM>) (False <PROGRAM>)
Description: Will first evaluate "Test"; if this is not "0", will evaluate and return the value of "True"; otherwise will evaluate and return the value of "False"
Value: The value of "True" or "False" depending on the value of "Test"

Command: imag (Value <COMPLEX>)
Description: Returns the imaginary part of Value.
Value: imag(Value) <DOUBLE>

Command: index (Arg1 <STRING or INTEGER>) (Arg2 <STRING or INTEGER>) … (Arg_n <STRING or INTEGER>)
Description: Will evaluate each Arg in order and return the string equal to "Arg1.Arg2. … .Arg_n" (without quotes). Used for array indexing.
Value: The string "Arg1.Arg2. … .Arg_n" (without quotes).

Command: integer (Value <STRING>)
Description: Constructs a new integer from the given string argument. Note "Value" is not evaluated, it is taken directly as it is.
Value: An integer equal to the given string (0 if the string is ill-formed)

Command: is_unitary (Value <MATRIX>)
Description: Determines if the given matrix is unitary
Value: Integer "1" if unitary, "0" otherwise.

Command: matrix (numRows <INTEGER>) (numCols <INTEGER>) (type <STRING>) [OPTIONAL]
Description: Constructs a new matrix of size numRows x numCols. "type" may be "ident" (identity matrix), "zero" (zero matrix), "QFT" (Quantum Fourier Transform), "u_random" (random unitary). If "u_random", you must specify three double arrays in the [optional] arguments; the first two arrays must be of size (1/2 (N)(N+1)); the last must be of size N where N = numRows = numCols. If "type" is omitted, you may specify each matrix entry instead in row-major form.
Value: A new matrix.

Command: measure (state <MATRIX>) (system <SYSTEM>) (index <INTEGER>) (result <STRING>)
Description: Will perform a measurement (in the computational basis) of the subspace indexed by "index" of the given state (which must exist in the provided system). "result" is a string name of a variable in the store (integer type) which will hold the value of the measurement (result will hold 0, 1, …, or dim(system[index])-1). Returns a new quantum state representing the collapse of "state" after measurement.
Value: A new MATRIX; see description.

Command: measure_distribution (state <MATRIX>) (system <SYSTEM>) (index <INTEGER>) (result <STRING>)
Description: "result" must be an array of size dim(system[index]). After calling this, result.i will hold the probability of the given state collapsing to system[index].i.
Value: NULL

Command: new (type <STRING>) (name <STRING>)
Description: creates a new variable of the specified type and name. Type may be "integer", "complex", "double", "matrix", "space", "system", or "array" (see below for last three).
Value: NULL

Command: new (type="array" <STRING>) (name <STRING>) (element_type <STRING>) (size <INTEGER>)
Description: creates a new array of the specified name and size. Each entry is of type "element_type". Use the "index" command to index in the array.
Value: NULL

Command: new (type="space" <STRING>) (name <STRING>) (dimension <INTEGER>)
Description: Creates a new Hilbert space of given dimension.
Value: NULL

Command: new (type="system" <STRING>) (name <STRING>) (numSpaces <INTEGER>) (space1 <STRING>) (space2 <STRING>) … (space_n <STRING>)
Description: Creates a Hilbert space which is the result of tensoring space1, space2, …, space_n. numSpaces must be equal to "n".
Value: NULL

Command: NOP (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)

Description: Will ignore each ARG (they are not evaluated) and do nothing.
Value: NULL

Command: nop (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Same as NOP just lower-case depending on user preferences
Value: NULL

Command: not (Arg <INTEGER>)
Description: Will "not" the value of Arg
Value: 1 if value of Arg is 0; 0 otherwise

Command: or (Arg1 <INTEGER>) (Arg2 <INTEGER>) … (Arg_n <INTEGER>)
Description: Will compute Arg1 | Arg2 | … | Arg_n (integer "or" operation)
Value: the integer Arg1| Arg2 | … | Arg_n

Command: pow (base <COMPLEX + INTEGER + DOUBLE>) (e <COMPLEX + INTEGER>)
Description: Raises "base" to the power "e". Note that if base is type COMPLEX then so must "e" be. Otherwise "e" must be INTEGER.
Value: base ^ e; if base is COMPLEX, value is COMPLEX; if base is INTEGER, value Is INTEGER; otherwise DOUBLE.

Command: pow2 (e <INTEGER>)
Description: Returns 2^e (using bit shift).
Value: 2^e <INTEGER>

Command: powm (base <INTEGER>) (e <INTEGER>) (m <INTEGER>)
Description: Returns base ^ e (mod m); computes modulo every iteration to avoid overflow.
Value: base ^ e (mod m) <INTEGER>

Command: print (Value <PROGRAM>)
Description: Will print to the console screen, the value of "Value"
Value: NULL

Command: rand (Max <INTEGER>)
Description: Returns a random integer from 0 to Max-1
Value: Returns a random integer from 0 to Max-1

Command: randf (Min <DOUBLE>) (Max <DOUBLE>)
Description: Returns a random double in the range [Min, Max)
Value: Returns a random double in the range [Min, Max)

Command: real (Value <COMPLEX>)
Description: Returns the real part of Value.
Value: real(Value) <DOUBLE>

Command: rootOfUnity (N <INTEGER>)
Description: Returns an N'th root of unity.
Value: The complex number: exp(2*pi*i/N)

Command: set (name <STRING>) (Value <PROGRAM>)
Description: Will set the value of the variable "name" to the value of "Value". Returns an error if types don't match.
Value: NULL

Command: set_element (mat <MATRIX>) (row <INTEGER>) (col <INTEGER>) (value <COMPLEX + DOUBLE>)
Description: Will return a new matrix constructed from "mat" changing the element at the specified row/col to the specified value. If "value" is DOUBLE, the element is set to value + 0i.
Value: A new matrix (see description).

Command: sin (Value <DOUBLE>)
Description: returns the sin(Value)
Value: sin(Value) <DOUBLE>

Command: sqrt (Value <INTEGER + DOUBLE>)
Description: Returns the square root of Value
Value: sqrt(Value) <DOUBLE>

Command: string (Value <STRING>)
Description: Constructs a new string from the given string argument. Note "Value" is not evaluated, it is taken directly as it is.
Value: A string equal to the given string.

Command: tensor (mat1 <MATRIX>) (mat2 <MATRIX>)
Description: Will return the tensor product of mat1 and mat2.
Value: A matrix representing the tensor of mat1 and mat2.

Command: transpose (Mat <MATRIX>)
Description: Computes the conjugate transpose of the given matrix
Value: The conjugate transpose of Mat.

Command: user.input (Type <STRING>)
Description: Will request input from the user. Type may be "integer" (an integer is returned), "string" (a string is returned), "char" (Windows only, an integer is returned representing the ASCII key pressed), or "kbhit" (Windows only, non-blocking; returns 0 if no key is pressed, otherwise the ASCII code of the pressed key)
Value: Depends on Type and the user input.

Command: while (Test <STRING>) (Prog1 <PROGRAM>) (Prog2 <PROGRAM>) … (Prog_n <PROGRAM>)
Description: Will evaluate Test; if this is 0, the loop terminates. Otherwise each Prog is evaluated in order. If one of the Prog evaluates to the string "break", the loop terminates, if one of the Prog evaluates to the string "continue" the while loop starts over (evaluating "Test" and running Prog1, etc.). After Prog_n has evaluated, the loop repeats by re-evaluating "Test".
Value: NULL