Q-HSK: A Quantum Simulation Language

Katherine Heller, Krysta Svore, and Maryam Kamvar

December 3, 2002

Contents

1 Q-HSK: An Overview **7**
- 1.1 **Introduction** . . . 7
- 1.2 **Team HSK Introduces Q-HSK** . . . 7
  - 1.2.1 **Familiar** . . . 8
  - 1.2.2 **Simple** . . . 8
  - 1.2.3 **Expressive** . . . 8
  - 1.2.4 **Epitomic** . . . 8
  - 1.2.5 **Cutting-Edge** . . . 8
  - 1.2.6 **Educational** . . . 8
- 1.3 **Sample Programs** . . . 9
- 1.4 **Conclusion** . . . 10

2 A Brief Tutorial on Q-HSK **11**
- 2.1 **Introduction** . . . 11
- 2.2 **Quantum Computers** . . . 11
  - 2.2.1 **Qubits** . . . 11
  - 2.2.2 **Quantum Gates** . . . 12
- 2.3 **A Simple Program** . . . 12
  - 2.3.1 **Why do we need include statements?** . . . 13
  - 2.3.2 **What are functions?** . . . 13
  - 2.3.3 **How do we declare a variable?** . . . 13
  - 2.3.4 **What is `qreg`?** . . . 14
  - 2.3.5 **What are the other library functions?** . . . 14
- 2.4 **The Graphical Interface** . . . 14

3 Q-HSK Reference Manual **17**
- 3.1 **Overview of Q-HSK** . . . 17
  - 3.1.1 **Features** . . . 17

3.1.2 **A Simple Example** . . . 17
3.2 **Q-HSK Syntax** . . . 18
3.2.1 **Statements** . . . 18
3.2.2 **Definitions** . . . 18
3.2.3 **Expressions** . . . 19
3.2.4 **Comments** . . . 19
3.2.5 **Include File** . . . 19
3.3 **Classical Expressions** . . . 19
3.3.1 **Classical Data Types** . . . 19
3.3.2 **Integer** . . . 19
3.3.3 **Real Number** . . . 20
3.3.4 **Complex Number** . . . 20
3.3.5 **Character** . . . 20
3.4 **Operators** . . . 20
3.4.1 **Arithmetic Operators** . . . 20
3.4.2 **Conditional Operators** . . . 20
3.5 **Functions** . . . 21
3.6 **Terminals** . . . 22
3.6.1 **Keywords** . . . 22
3.6.2 **Identifiers** . . . 22
3.6.3 **Variables** . . . 22
3.6.4 **Scope** . . . 22
3.7 **Pointers** . . . 22
3.8 **Classical Statements** . . . 23
3.8.1 **Function Calls** . . . 23
3.8.2 **Assignment** . . . 23
3.9 **Quantum Statements** . . . 23
3.9.1 **Quantum Registers** . . . 23
3.9.2 **Quantum Expressions** . . . 23
3.9.3 **Unitary Operations** . . . 23
3.9.4 **Non-Unitary Operations** . . . 24
3.10 **Control Flow** . . . 24
3.10.1 **Blocks** . . . 24
3.10.2 **Conditional Branch** . . . 24
3.10.3 **Loops** . . . 24
3.11 **Subroutines** . . . 25
3.11.1 **Restrictions** . . . 25
3.11.2 **Functions** . . . 25

3.11.3 **Included Subroutines** 25

4 Q-HSK Project Plan **27**
4.1 **Process** 27
4.2 **Implementation Style Sheet** 27
4.2.1 **Indentation** 27
4.2.2 **Tabs** 27
4.2.3 **Braces** 27
4.2.4 **Statements and Blocks** 28
4.2.5 **Methods** 28
4.2.6 **Function Names** 28
4.2.7 **Variables** 28
4.2.8 **Constants** 28
4.2.9 **Comments** 29
4.3 **Timeline** 29
4.4 **Roles and Responsibilities** 30
4.5 **Software Development Environment** 31
4.6 **Project Log** 31

5 Q-HSK Architecture **35**
5.1 **Structural Diagram** 35
5.2 **Interfaces** 36
5.3 **Team Responsibilities** 36

6 Q-HSK Test Plan **37**
6.1 **Examples** 37
6.1.1 **Hadamard** 37
6.1.2 **Fourier Transform** 38
6.2 **Test Suites** 39
6.2.1 **Test 1** 39
6.2.2 **Test 2** 39
6.2.3 **Test 3** 40
6.2.4 **Test 4** 40
6.3 **Automation** 41
6.4 **Team Responsibilities** 41

7 Lessons Learned **43**
7.1 **Katherine Heller** . . . 43
7.2 **Krysta Svore** . . . 44
7.3 **Maryam Kamvar** . . . 44
7.4 **Future Advice** . . . 45
7.4.1 **Communication** . . . 45
7.4.2 **Organization** . . . 46
7.4.3 **Expansion** . . . 46

A Q-HSK Grammar **47**
A.1 **Notation** . . . 47
A.2 **Grammar** . . . 47

B Shor's Prime Factorization Algorithm **51**

C Q-HSK Library Functions **61**
C.1 **Classical Functions** . . . 61
C.1.1 `TestPrime` . . . 61
C.1.2 `TestPrimePower` . . . 61
C.1.3 `RegSize` . . . 61
C.1.4 `GetQ` . . . 62
C.1.5 `denominator` . . . 62
C.2 **Quantum Functions** . . . 62
C.2.1 `modexp` . . . 62
C.2.2 `computeHadamard` . . . 62
C.2.3 `DFT` . . . 63
C.2.4 `Norm` . . . 63
C.2.5 `Measure` . . . 63

D Implementation Source Code **65**
D.1 **The Lex File** . . . 65
D.2 **The Yacc File** . . . 66
D.3 **The Translation File** . . . 67
D.4 **The Java File** . . . 68

Chapter 1

Q-HSK: An Overview

1.1 Introduction

Currently, quantum computing is a very theoretical field as no highly functional quantum computer exists. Researchers expect quantum computing to be realized in the next 30 years, but until the field matures, the current state of quantum technology allows only for simple calculations.

Quantum simulators are used to model the activity of a quantum computer on a classical computer, however their functionality is limited. Moreover, within the quantum community, the algorithmic language used to define quantum functions and operations is not formalized, which adds an unnecessary level of complexity to quantum computer simulation.

In order to simulate an algorithm, researchers must code complex mathematical operations using an existing programming language. Current programming languages, such as C/C++, MatLab and Java, lack the high-level mathematical tools for operations related to quantum mechanics.

1.2 Team HSK Introduces Q-HSK

Q-HSK is meant to facilitate the implementation of quantum algorithms on a classical computer. Below you will find the key attributes of our language.

1.2.1 Familiar

Q-HSK expands upon the C language, and uses a similar syntax. This creates an easy-to-use development environment for programmers familiar with C and is expected to spur the adoption of Q-HSK in the Computer Science community.

1.2.2 Simple

Q-HSK hides the complex mathematics of quantum mechanics within its built-in functions. This allows the researcher to perform complex operations with ease and efficiency. Programs are thus compact, powerful, and easy to follow.

1.2.3 Expressive

Q-HSK builds upon C with additional types and operators for use in a quantum environment. This provides a higher level of abstraction for elegant formulation of quantum algorithms.

1.2.4 Epitomic

Q-HSK has the potential to be the first global language for quantum computer simulation. Currently there is no unification of simulation languages and techniques. Q-HSK hopes to provide this needed standardization.

1.2.5 Cutting-Edge

Q-HSK will be instrumental in aiding the development of a quantum computer. It will allow researchers to efficiently develop and test new algorithms. Furthermore, this language will not be rendered obsolete when an extensive quantum computer is realized; it is sustainable with a quantum architecture.

1.2.6 Educational

Q-HSK will assist both the beginning and advanced user interested in quantum computation and quantum algorithms. The graphical visualization of quantum algorithms provides a novel method for both teaching and research.

1.3 Sample Programs

Three sample programs are provided below.

The following program calculates the hadamard transform of a quantum bit (qubit) and then measures it on a quantum simulation. It demonstrates the use of the new data type `qreg`. It also contains the functions `computeHadamard` and `Measure`, which is included in the new language.

```
int main()
{
    qreg *a; /*qbit a is an array of 5 registers |00000>*/
    a = create(5, "a");
    int d;
    a = computeHadamard(a, RegSize(a)); /*compute the hadamard
transform of a*/
    d = Measure(a, RegSize(d), ''d''); /*measure the result*/
    printf("%d",d); /*print the result*/
    return 0;
}
```

The following program calculates the quantum fourier transform of a qubit on a quantum simulation. This requires a function call to `computeHadamard` and to `DFT`, the quantum fourier transform function.

```
int main()
{
    qreg *a; /*qbit a is an array of 7 registers |0000000>*/
indent a = create(7, "a");
    int q;
    a = computeHadamard(a, RegSize(a)); /*compute the hadamard
transform of a*/
    a =DFT(a, GetQ(a), RegSize(a)); /*perform the discrete quantum
fourier transform*/
    q = Measure(a, RegSize(q), ''q''); /*measure the result*/
    printf(''%d'', q); /*print the result*/
    return 0;
```

```
}
```

The following program calls Shor's algorithm on the number 15 using quantum simulation. It calls the function `shor` which is located in Appendix C. Slight modifications to the source code given in Appendix C will have to be made.

```
void main()
{
    int prime;
    prime=15;
    shor(prime); /*calculate the factorization using Shor's algorithm*/
}
```

1.4 Conclusion

Q-HSK is a language that facilitates quantum computing research by providing a forum for standard expression of quantum operations. The programs can be simply composed and implemented, as demonstrated by the given examples. It is projected that Q-HSK will be assimilated into mainstream Computer Science because of its syntax similarity to the existing C language, and the advantages it offers.

For more information send email to:
Maryam Kamvar (mkamvar@cs.columbia.edu)
Krysta Svore (kmsvore@cs.columbia.edu)
Katherine Heller (heller@cs.columbia.edu)

Chapter 2

A Brief Tutorial on Q-HSK

2.1 Introduction

We begin the tutorial by looking at a small Q-HSK program. It is important that the programmer understand basic C/C++ programming concepts. A basic knowledge of linear algebra and quantum mechanics is desired, but is not necessary for the Q-HSK programmer. Before jumping into the program, let's review some fundamentals of quantum computation.

2.2 Quantum Computers

Several definitions and concepts are given below to assist the reader in learning about the magic of quantum computation. However, this review is far from complete. For more information, see [1].

2.2.1 Qubits

A quantum computer exploits quantum mechanics to perform operations on bits. A quantum circuit has quantum bits which are analagous to classic bits, however a quantum bit can assume not only a value of 0 or 1, but also a superposition of weighted 0 and 1 combinations. This property is due to the quantum mechanical properties of the quantum bits. Thus, using quantum algorithms allows for exponentially faster computation. A quantum bit is conventionally referred to as a qubit. One or more qubits can be placed in

a quantum register, much like one or more classical bits can be placed in a classical register.

2.2.2 Quantum Gates

We again can draw upon the analogy to classical computers to describe quantum gates. Qubits are passed through quantum gates in a quantum circuit, just as bits are passed through classical gates in a classical circuit. Quantum gates include a subset of gates similar to those found in a classical circuit. These include NOT gates, XOR gates, and AND gates. Quantum gates also include a new set of quantum specific gates. Examples of such gates are the unitary operations which are reversible computations. There are Hadamard transforms, Fourier transforms, controlled phase shifts, and others. The true power of quantum computation lies in these quantum gates that are able to act as black boxes and perform complicated mathematical operations in parallel. Refer to [1] for more information about quantum gates. In addition, the graphical interface will provide more information about each gate.

2.3 A Simple Program

One way to learn Q-HSK is to look at a sample program and step through it. We shall begin by looking at the following program that demonstrates the use of thye `qreg`: a Q-HSK specific data type.

```
1 #include ‘‘complex’’
2 #include <stdio.h>

3 int main()
4 {
5    int n, q, value;
6    n = 15;
7    q = GetQ(n);
8    qreg *reg1;
9    reg1 = create(RegSize(q)-1, ‘‘reg1’’);
10   reg1 = computeHadamard(reg1, q-1);
11   qreg *reg2;
12   reg2 = create(RegSize(n)-1, ‘‘reg2’’);
```

```
13   reg2 = Norm(reg2, RegSize(n));
14   value = Measure(reg2, RegSize(n), ``reg2'');
15   printf(``The value is %d\n'', value);
16   return 0;
17 }
```

2.3.1 Why do we need include statements?

`Lines 1-2` are the include statements containing the proper header files. In Q-HSK, the C libraries containing the C functions used within the program must be declared in the include statements. Also, the `complex` header file provided by the Q-HSK system must be included if the type `complex` or the type `qreg` is used in the program.

2.3.2 What are functions?

The `main` function marks the start of a Q-HSK program. Q-HSK does not allow `main` to return type `void` since there is no type `void` in Q-HSK. The return type of `main` in this program is an integer, but any function can return any of the Q-HSK data types. There are no arguments to `main` in this program, but it is possible to pass in arguments to a function if desired using syntax similar to C. Reading arguments from the command line again follow C syntax. This is achieved by passing in parameters to the `main` function:

```
main(int argv, char *argv[]);
```

If the value of the variable `n` was an argument to the program, `line 6` would be replaced with:

```
n=atoi(argv[1]);
```

It's that simple!

2.3.3 How do we declare a variable?

Declarations in Q-HSK are identical to variable declarations in C, with one exception: variables cannot be defined in the same statement they are de-

clared. The line `int a=5;` is not allowed in Q-HSK! `Line 5` contains the declaration of the variables, and `lines 6-7` are the definitions of each variable. `Line 7` calls the Q-HSK library function `GetQ()`, a function that returns the integer representing the number of qubits required for the binary representation of a number n. For further details, see the Appendix C.

2.3.4 What is `qreg`?

`qreg` is a new type in Q-HSK that allows the user to manipulate quantum registers and qubits directly. A qubit is a complex number. A `qreg` is a vector of qubits. In the above program, `line 8` declares a quantum register `reg1`. In this case, `reg1` is a pointer to a quantum register. This means that it is dynamically allocated memory. Just as in C, memory allocation must be explicit. In Q-HSK, the function `create` is similar to `malloc` in C. On `line 8` we create `RegSize(q)-1` units of space for `reg1`. See Appendix C for details on the Q-HSK library functions `RegSize` and `create`.

2.3.5 What are the other library functions?

There are several other functions used in the above sample program. `computeHadamard` is an implementation of the Hadamard transform. `Norm` normalizes the register values so that the probability of measuring a state is given by summing the squares of its probability amplitudes. `Measure` measures the quantum register, collapses it into a classical state, and outputs a single integer. All details on these functions can be found in Appendix C.

2.4 The Graphical Interface

During the compilation of a Q-HSK program, information is aggregated in order to construct a visualization of the quantum flow of the algorithm. This information is then translated into a Java program which is automatically executed to display the Q-HSK statements concurrently with the construction of the quantum circuits.

To run the above program, copy and paste it into your text editor and save it as `demo.q`. Type `qhsk demo.q ''demo''` to compile the program. If there are no errors, then two executable files will be formed called `cdemo` and

`jdemo`. `cdemo` is the textual output and `jdemo` is the graphical Java file that can be loaded to see the quantum circuit in action.

Chapter 3

Q-HSK Reference Manual

3.1 Overview of Q-HSK

The Q-HSK programming language is designed to facilitate the implementation and simulation of quantum algorithms on a classical computer. Q-HSK outputs a graphical representation of quantum operations and gates, thus providing a unique tool for a deeper understanding of quantum algorithms.

3.1.1 Features

Q-HSK offers many key features for quantum simulation programming. These include:

- Several classical data types (`int`, `real`, `complex`, `char`)
- A quantum data type for quantum registers (`qreg`)
- Manipulation of quantum registers
- Easy C-style syntax

3.1.2 A Simple Example

In this simple example of a Q-HSK program, we take the measure of a quantum register `qreg` after it has been put into a superposition by the Hadamard transform.

```
int main()
{
   int a;
   qreg *q;
   q=create(5, ‘‘q’’);
   q = computeHadamard(q);
   a = Measure(q);
   printf(‘‘This is the measure:  %d’’, a);
   return 0;
}
```

3.2 Q-HSK Syntax

Q-HSK notation can be described with a context-free grammar. The complete grammar can be found in Appendix A. The program consists of lines, which can be either declarations or functions. Functions are composed of expressions, statements, and declarations. The Q-HSK language provides many of the same conventions as other common programming languages. It features conditional statements, a loop structure, and expressions. Further detail for each subset is provided below.

3.2.1 Statements

Q-HSK statements can be simple or complex. Statements may include a range of options such as function calls, simple commands, and control structures. An example of a statement is

```
if (x>5)
   printf("x is greater than 5");
```

3.2.2 Definitions

Definitions associate an identifier with a specified value. The identifier is a variable of type consistent with the value being assigned to it. For example,

```
complex i;
complex i = (2.0, 3.0);
```

3.2.3 Expressions

An expression is a syntactically correct combination of terminal symbols, as defined by the grammar rules found in Appendix A. For example, `a + b` is an expression in Q-HSK.

3.2.4 Comments

Only C-style comments are supported in Q-HSK. Comments must start with `/*` and end with `*/`. They may be of any length, but nested comments are not supported. A `*` or `/` is not allowed within the comment.

3.2.5 Include File

Files can be included with the command `#include` *filename*. All included files must be within the same directory as the program file. Included files must be declared at the beginning of the program file.

3.3 Classical Expressions

3.3.1 Classical Data Types

The Q-HSK programming language contains four classical data types: `int`, `real`, `complex`, and `char` and pointers to those types.

Type	**Detail**	**Example**
`int`	integer	3
`real`	real number	2.237
`complex`	complex number (real part, imaginary part)	(2.0, 3.0)
`char`	character	a
*type**	pointer to a type	`char *`

3.3.2 Integer

An integer is a whole number or zero with an optional positive or negative sign. If no sign is present, then the integer is assumed to be positive. An `int` can be up to 32 bits in size.

3.3.3 Real Number

A real number is made of a combination of an integer part and a fractional part expressed in decimal notation. At least one digit to the right and left of the decimal point is required. A `+` or `-` is optional. A typical size for a `real` is 64 bits.

3.3.4 Complex Number

The `complex` data type is represented with two `real` types. The syntax is (*real part*, *imaginary part*), where each part is a `real` type.

3.3.5 Character

The `char` type is any alphanumeric character.

3.4 Operators

3.4.1 Arithmetic Operators

There are several arithmetic operators in Q-HSK. Their precedence is the same as that of C++. The following table summarizes these operators.

Operator	Detail	Example	Type
`%`	integer modulus	`5 % 3`	`int`
`*`	multiplication	`5*3`	`int, real, complex`
`/`	division	`4/2`	`int, real, complex`
`+`	addition	`(1.0, 2.0) + (3.0, 4.0)`	`int, real, complex`
`-`	subtraction	`3.5 - 2.7`	`int, real, complex`

3.4.2 Conditional Operators

There are several conditional operators offered in Q-HSK. The precedence is consistent with that of C++. These are listed in the following table.

Operator	**Detail**	**Type**
`==`	equal	`int, real, complex, char`
`!=`	not equal	`int, real, complex, char`
`<`	less than	`int, real`
`>`	greater than	`int, real`
`<=`	less than or equal	`int, real`
`>=`	greater than or equal	`int, real`
||	logical or	`int, real`
`&&`	logical and	`int, real`
|	bit or	`int, real`
`&`	bit and	`int, real`

3.5 Functions

There are several functions offered in Q-HSK. They are listed below.

Function	**Detail**
`pow(x, y)`	x raised to the y power
`exp(x)`	e raised to the xth power
`log(x)`	natural logarithm of x
`log(x, n)`	logarithm of x base n
`sqrt(x)`	square root of x
`ceil(x)`	round up to nearest integer
`floor(x)`	round down to nearest integer
`max`$(x_0, x_1, ..., x_n)$	maximum of $(x_0, x_1, ..., x_n)$
`random()`	random value between [0, 1)
`sin(x)`	sine of x
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`cot(x)`	cotangent of x
`GCD(x, y)`	greatest common denom. of x, y

3.6 Terminals

3.6.1 Keywords

The following words are considered keywords in Q-HSK: `if`, `while`, `else`, `return`, `printf`, `create`, `int`, `real`, `complex`, `char`, `qreg`, `CBLAS`.

3.6.2 Identifiers

Identifiers in Q-HSK can begin with any letter, and can be followed by any number of alphanumeric characters or an underscore. Identifiers are case sensitive and cannot be a Q-HSK keyword.

3.6.3 Variables

A variable is a type-specific name which must be declared before its use. Variables must be declared and initialized in separate statements. A variable cannot be a Q-HSK keyword.

3.6.4 Scope

A variable is exists within the code block in which it is defined. If there are nested blocks of code, variables existing in the outer block exist in the inner blocks.

3.7 Pointers

Pointers are allowed in Q-HSK. There must be a space between the type and the `*` symbol. Pointers may be used with any type. To allocate memory for pointers of type `int`, `real`, `complex`, `char` use the command `malloc` as in C. When allocating memory for pointers of type `qreg` use the command `create`.

3.8 Classical Statements

3.8.1 Function Calls

Q-HSK supports recursive function calls.

3.8.2 Assignment

An assignment of a value to an identifier can be made with the operator `=`. The types of the value and of the identifier must both be consistent and valid.

3.9 Quantum Statements

3.9.1 Quantum Registers

There is a single data type for quantum registers called `qreg`. `qreg` is a quantum register of qubits.

3.9.2 Quantum Expressions

The following table lists the ways to reference a quantum register `qreg` and its qubits.

Expression	Detail	Register
`q`	quantum register	entire register
`q[i]`	qubit	single qubit

To dynamically create a qubit, use the function `create()`. The syntax can be found in Appendix C.

```
qreg *q;
q = create(5, ‘‘q’’);
```

3.9.3 Unitary Operations

A unitary operation U is an operation such that $U^\dagger U = I$, where $\dagger$ is the adjoint (transposition followed by complex conjugation) and I is the identity matrix. Any unitary operator is a valid quantum gate. They must be

reversible and invertible operations. Such gates include the Hadamard gate, the controlled-not gate, and the SWAP gate.

3.9.4 Non-Unitary Operations

The measurement operation is important in quantum computation. It converts a single qubit into a classical bit with a corresponding probability. This is irreversible and non-invertible. `Measure` measures a quantum register.

```
   int i;
qreg *q;
q = create(2, ``q'');
q = computeHadamard(q);
i=Measure(q);
```

3.10 Control Flow

3.10.1 Blocks

A block is a set of statements. Control flow in a block must continue from statement to statement.

3.10.2 Conditional Branch

Q-HSK contains `if` and `if-else` statements analagous to their definitions in C++. If the expression is true, then the statements within the `if` block are executed. Otherwise, the statements in the `else` block, if such a block exists, are executed.

3.10.3 Loops

There is only one type of loop in Q-HSK: the conditional `while` loop. As long as the given expressions remain true, the loop continues to execute.

3.11 Subroutines

Subroutines are offered for both classical and quantum settings. There are classical functions of types `int, real, complex,` and char, pseudo-classical operators of type `qreg`, general unitary operators of type `qreg`, algorithms of types `int, qreg`, and `CBLAS` matrix library function calls of type `complex` and `qreg`.

3.11.1 Restrictions

Subroutines can only be called from other routines. There is no need to include a header file to use the library functions of Q-HSK. However, functions within a C library must include their proper C header file. A complete list of Q-HSK library functions are listed in Appendix C.

3.11.2 Functions

Functions can be of type `int, real, complex, qreg` or `char`. They may require as many parameters as deemed necessary. Local variables are defined at the beginning of the function definition. To return a value from the function, a `return` statement is invoked. Recursion is allowed in Q-HSK and calling of other functions is allowed. Typecasting is permitted in Q-HSK.

3.11.3 Included Subroutines

Functions available in Q-HSK include prime testing, prime power testing, hadamard transform, quantum fourier transform, quantum measurement, the size of a register, and the number of qubits needed for a given operation. Refer to Appendix C for a more complete listing. `CBLAS` functions can also be called.

Chapter 4

Q-HSK Project Plan

4.1 Process

Implementation of the Q-HSK system began in September 2002. First, the lexer, tokenizer, and parser were designed and tested. We then continued development by translating a Q-HSK program into C++ with a robust translator and by adding a graphical interface was added to provide a learning tool.

4.2 Implementation Style Sheet

4.2.1 Indentation

Use tabs for indenting.

4.2.2 Tabs

Tabs shall be at five space intervals, except for the Yacc file which is tabbed for readability.

4.2.3 Braces

All braces shall be aligned vertically or horizontally. If matching braces are not readily identifiable, i.e., more than one screen apart, add numbered

comments to match them. For example, the braces enclosing the class block would be:{//[1] and } //[1]

4.2.4 Statements and Blocks

Blocks shall be indented one tab space beyond their calling statement.

4.2.5 Methods

Leave a blank line before and after all methods.

4.2.6 Function Names

All function names shall, whenever possible, describe the function's purpose. Names will be initial upper case with intermediate names capitalized.

4.2.7 Variables

Variable names shall describe their purpose. One letter variable names (names without any connotation) are reserved for loop indicies and counters. Variable names should have initial lower case letters.

Global Variables

Declare all global variables together at the top of the code when possible.

Local Variables

Declare all local variables at the beginning of their respective functions.

4.2.8 Constants

Constant names shall describe their purpose. All constants names are all UPPER CASE, with intermediate words separated by an underscore "_" character.

4.2.9 Comments

All code shall have a commented block at the very top of each function. The block shall include a description of the program's function and the name of the team-member who created the function.

Single-line Comments

Single-line comments should use the `// ...` format.

Multi-line Comments

Multi-line comments should use the `/* ... */` format (do not use the `//` format for multi-line commenting). Do not place stars or other characters at the beginning of each line of the intermediate lines. Instead, to clearly deliminate the commented section, the section should be surrounded by a line of stars at the beginning and end of the function, i.e. `/**********************`.

4.3 Timeline

Below is a timeline of our goal points during the implementation of the Q-HSK system.

Date	Goal Achieved
10/1/2002	•Preliminary ideas of design
10/10/2002	•Whitepaper completed
10/17/2002	•Parsing of simple expressions (no quantum types)
10/24/2002	•Parsing of complex expressions (no quantum types)
10/31/2002	•Reference manual rough draft •Removal of conflicts in Yacc
11/7/2002	•Reference manual completed •Complete parsing of non-quantum expressions
11/14/2002	•Implement basic graphics •Parsing of `complex`
11/17/2002	•Allow `CBLAS` calls •Parsing of `qreg`
11/19/2002	•Parsing of Shor's algorithm •Basic graphics working •In-class demo
11/21/2002	•Add mouse interaction to graphics
11/25/2002	•Complete interaction between translator and graphics
11/27/2002	•Finish code
12/1/2002	•Full testing •Finalize graphics •Final report rough draft
12/3/2002	•Project completed •Final report completed

4.4 Roles and Responsibilities

The design and development of the Q-HSK system was a cooperative, collaborative effort between all Q-HSK team members.

Each member of the Team worked together to get the rudimentary Lex, Yacc and C translation files running with a makefile. Establishing this foundation together helped the Team to better understand the limitations and abilities of these tools, which was paramount in deciding the future direction

of the project. Once these initial design plans had been laid out however, leadership responsibilities were delegated to different team members in order to expedite progress.

The Q-HSK system was divided into three subsections for development purposes. Katherine Heller led the effort to develop the robust Lex , Yacc and translation files. Krysta Svore led the effort to develop the Q-HSK quantum function "library" in C++, and Maryam Kamvar led the effort to develop the Java-based animation output of Q-HSK programs.

Leadership responsibilities primarily entailed programming development, and making design decisions internal to each member's sub-section of the Q-HSK system.

Although server space had been allocated for Q-HSK development on which integration could be performed remotely, Team Q-HSK preferred to be together for this process not only because it would be easier to identify which part of the system was causing the bug with three brains, but the identification of bugs in the system would often lead the team to re-consider a major design decision. These decisions could not have been made by one team member herself because they typically affected each part of the System.

4.5 Software Development Environment

The parsing files were written in Lex and Yacc using Emacs and the translation file was also written in C using Emacs. The quantum functions were written in C++, again using Emacs. The Q-HSK animator was written in Java using the FORTE IDE.

4.6 Project Log

Below is list of tasks both completed and uncompleted by date.

Date	**Achievements**
10/1/2002	•Developed an overview of the project
10/10/2002	•Solidified project ideas
10/17/2002	•Able to parse simple non-quantum expressions •Developed idea of graphical interface
10/24/2002	•Able to parse complex non-quantum expressions •Begin design of graphics •Need to test capabilities of parser
10/31/2002	•Completed a rough draft of the reference manual •Removed most of the conflicts in the Yacc file •Need to handle complex number arithmetic •Need to investigate CBLAS, BLAS
11/7/2002	•Completed the reference manual •Able to parse all non-quantum expressions •Need to decide on a matrix library
11/14/2002	•Finished basic graphics •Able to parse type `complex` •Included CBLAS matrix package •Need to incorporate type `qreg`

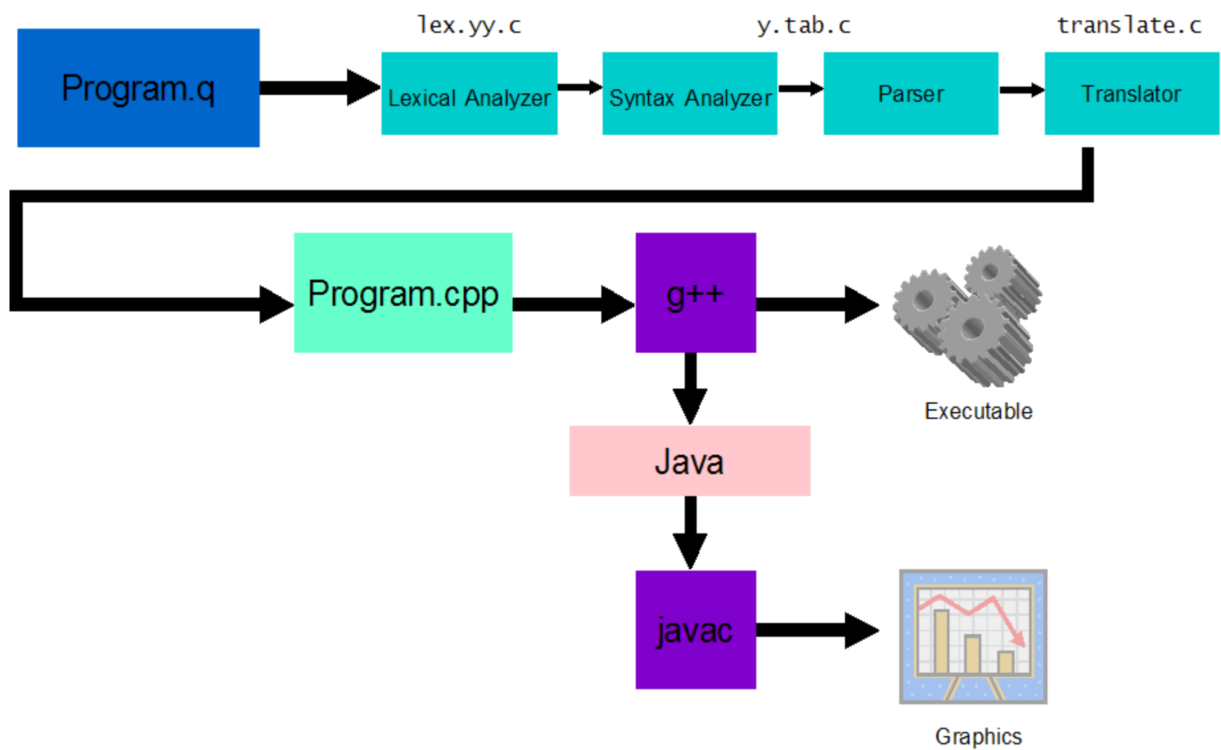
Date	**Achievements**
11/17/2002	•Able to handle CBLAS calls •Able to parse type `qreg` modified `complex` library
11/19/2002	•Able to parse Shor's algorithm •Basic graphics working •In-class demo •Need to automate graphic calls •Need to align g++ errors with Q-HSK program errors •Need to handle double-click on a quantum gate •Need to translate java calls
11/21/2002	•Need to continue to interpret mouse movement
11/25/2002	•Completed interaction between translator and graphics
11/27/2002	•Finished code
12/1/2002	•Full testing •Finalized graphics •Completed final report rough draft
12/3/2002	•Project completed •Final report completed

Chapter 5

Q-HSK Architecture

5.1 Structural Diagram

The diagram below shows the flow of a program when it enters the Q-HSK system.



5.2 Interfaces

The process of compiling a `.q program` is fairly straighforward. The user produces a `.q file` to be compiled. This file is then sent through the tokenizer written in Lex to be broken into tokens. The parser then parses the input from the Lex program and produces a C++ output file. This is directly passed through g++ for further compiling and error checking. g++ produces a compiled, runnable executable file `a.out`. When `a.out` is run, it produces both a textual output file and a Java file that can be compiled to produce a graphical representation of the quantum steps of the original `.q` program. All of this is done automatically when the command `qhsk` *program*`.q` is executed.

5.3 Team Responsibilities

The entire team worked on designing the various components of the Q-HSK system. Katherine Heller primarily developed tokenizing and parsing the `.q` file. She worked with the Yacc, Lex, and C translation files to help produce a concise and accurate system. Krysta Svore mainly developed the quantum simulation components of the system and asssited with the quantum translation. She mainly worked with the C translation programs and the connection between the translated program and the Java interface. Maryam Kamvar primarily led the efforts on the graphical interface. She devloped the Java program and the communication between the C++ and Java files which produced a visual guide to quantum algorithms.

Chapter 6

Q-HSK Test Plan

6.1 Examples

Below is a listing of the Q-HSK source code and its translation into C++.

6.1.1 Hadamard

The following program calculates the hadamard transform of a quantum register and then measures it.

Q-HSK

```
int main()
   {
      qreg *a; /*qbit a is an array of 5 registers |00000>*/
      a = create(5, “a”);
      int d;
      a = computeHadamard(a, RegSize(a)); /*compute the hadamard
transform of a*/
      d = Measure(a, RegSize(d), ‘‘d’’); /*measure the result*/
      printf("%d",d); /*print the result*/
      return 0;
   }
```

C++

```
int main()
   {
      complex *a; /*qbit a is an array of 5 registers |00000>*/
      a = (complex *)malloc(5*sizeof(complex));
      int d;
      a = computeHadamard(a, RegSize(a)); /*compute the hadamard
transform of a*/
      d = Measure(a, RegSize(d), ''d''); /*measure the result*/
      printf("%d",d); /*print the result*/
      return 0;
   }
```

6.1.2 Fourier Transform

The following program calculates the quantum fourier transform of a qubit on a quantum simulation. This requires a function call to `computeHadamard` and to `DFT`, the quantum fourier transform function.

Q-HSK

```
int main()
   {
      qreg *a; /*qbit a is an array of 7 registers |0000000>*/
   indent a = create(7, "a");
      int q;
      a = computeHadamard(a, RegSize(a)); /*compute the hadamard
transform of a*/
      a =DFT(a, GetQ(a), RegSize(a)); /*perform the discrete quantum
fourier transform*/
      q = Measure(a, RegSize(q), ''q''); /*measure the result*/
      printf(''%d'', q); /*print the result*/
      return 0;
   }
```

C++

```
int main()
   {
      complex *a; /*qbit a is an array of 7 registers |0000000>*/
   indent a = (complex *)malloc(7 * sizeof(complex));
      int q;
      a = computeHadamard(a, RegSize(a)); /*compute the hadamard
transform of a*/
      a =DFT(a, GetQ(a), RegSize(a)); /*perform the discrete quantum
fourier transform*/
      q = Measure(a, RegSize(q), ``q''); /*measure the result*/
      printf(``%d'', q); /*print the result*/
      return 0;
   }
```

6.2 Test Suites

Q-HSK testing mentality was to perform many incremental integration tests. Because of the sheer number of test files, the entire test suite will not be outlined in this document. However, the key tests that comprise the testing suite are outlined below.

6.2.1 Test 1

This is the first test program that was successfully run through Q-HSK system.

```
   int main()
{
   int a;
   a = 2+3;
}
```

6.2.2 Test 2

This is a test program that revealed to team Q-HSK members that although Q-HSK generated appropriate errors for mulitiple declarations of a single variable name, Q-HSK would erroneously report variable clashes for variables in different scopes. Note that as a result, the team decided that a C++ compiler would deal with complex error handling.

```
   int main()
{
   int a;
   a = 5;
   return 0;
}


int fun2()
{
   int a;
   return a;
}
```

6.2.3 Test 3

This is a test program which successfully parsed Q-HSK specific functions, and generated an accurate C++ translation.

```
   void main()
{
   int a;
   qreg *b;
   b = create(5, ‘‘b’’);
   a = 5;
   return 0;
}
```

6.2.4 Test 4

Shor's algorithm is a complete Q-HSK program whose output was successfully run through the Java animator. Shor's algorithm can be found in Appendix B.

6.3 Automation

To facilitate automation, Q-HSK had a makefile, with which it was simple to run each test file that comprised the test suite. Unfortunately, complete testing automation was never achieved over the entire suite.

6.4 Team Responsibilities

The testing responsibilities for the Q-HSK system was divided along the same lines leadership responsibilities were divided. Katherine Heller was responsible for ensuring compilation and accurate output of the lex, yacc and translation files, Krysta Svore was responsible for ensuring compilation and accurate output of the C++ file that contained quantum functions, and Maryam Kamvar was responsible for ensuring compilation and accurate output of the Java animation file. Integration testing, much like the integration itself, was the responsibility of the entire team and was performed together.

Chapter 7

Lessons Learned

Below, each team member describes her experience during the design and implementation of Q-HSK.

7.1 Katherine Heller

The most significant lesson I learned from this project is the importance of teamwork and collaboration. My fellow team members were extremely dedicated people and we all worked together extremely well, regardless of the extent of our undertaking. Without any one team member successful completion of this project would not have been possible. I also believe this project owes greatly to the enthusiasm, availability, and general guidance (especially at the early stages) of both our TA and professor. Everyone I worked with was full of positive energy and determination. I believe that is why this has been the most enjoyable (regardless of sleep deprivation) and probably most successful project I have ever worked on.

I also learned the importance of working hard towards ambitious goals, regardless of how far off or unattainable they may seem at the time. I learned that it helps to plan and think ahead, while at the same time it helps to start small and slowly build up to a final product.

Lastly, I learned that it is much more interesting to define your own project based on the interests of you and your teammates. This encourages creativity and inspires you to come up with novel ideas.

7.2 Krysta Svore

Over the course of the semester, I have learned the ins and outs of writing a robust programming language to suit a particular niche. But I have found the the more lasting aspect of the project is the skills I have acquired over the past few months.

I have learned that it is crucial to maintain a list of goals and corresponding dates to complete those goals in order to make consistent progress. Without weekly meetings and parallel programming sessions, I don't believe we would have accomplished near the amount that we have. When we attempted to finish different sections on our own, it did not work out as well as the parallel sessions. But as we learned from these experiences, our language grew and grew. We began the project with the belief that our language might never make it, and finished with more functionality and cool features than ever imagined!

I have also found that it is important to divide the work and delegate one person as the leader of each division. This helps things progress more quickly and more easily. Each person really became an expert in an area by the end of the semester, which really helped maintain the forward movement. We all worked together to solve problems and bugs, but we really tried to distinguish the divisions of work.

At the end, I learned that it is extremely important to keep testing the various components together. Sometimes we fell a bit behind on this, and realized that we had flaws only after combining the seperate entities. Overall, I have learned that an organized plan and a clear division of work is a large component of a successful project.

7.3 Maryam Kamvar

Most importantly, this project reinforced the immeasurable value of teamwork. From the inception of our team name, Team Q-HSK acted as a unified force, working towards a better quantum simulator. I believe much of the success of our project is due to the motivation and dedication of each team member, and moreover due to the ease with which we were able to work

together.

I also learned that although a picture can convey a thousand words, it might lack three vitally important and key descriptive words. I learned this lesson at about 2:30am one morning, while I was demoing a rudimentary implementation of the graphics to my other team members. I had been modeling the animation after pictures drawn on the whiteboard, which hadn't explicitly represented the non-concurrency in a quantum state. This realization resulted in a complete overhaul of the graphics system and objects. But this situation also gave rise to another valuable lesson: Implementing code twice inspires significant improvements! I believe that the second design of the graphics system is more extensible and manageable than the first.

As for future advice, I would encourage anyone embarking on a similar endeavor to find a project that is meaningful. Moreover, I believe the project will be multi-fold more exciting if you are able to find your own 'niche' into which you can use your past experiences and apply your knowledge. Although I had no background in Quantum Computing, I understood the novelty of our idea, and worked on the development of Q-HSK as a learning tool for Quantum Computing. I appreciated this on two levels: first, because I wanted to learn as much about Quantum Computing as I could through this project, and secondly, because I enjoy teaching and experimenting with new teaching tools and methods. Hence I found the Q-HSK project very relevant to my own goals, and I think that each team member was similarly able to find their own 'niche'.

7.4 Future Advice

Our advice for future groups stems from our own lessons learned over the semester.

7.4.1 Communication

We believe communication is key. Detailing the design early helps to eliminate later problems. We spent several meetings redesigning components because our early layout was not complete enough. Explicit definitions of the input and the output for each component is important and will save the

team valuable time.

7.4.2 Organization

It is important to be organized before, during, and after the project. If you organize team deadlines along the way, then the project will continue to progress. We had weekly team meetings and frequent parallel programming sessions to maintain organization. Also, beginning small and expanding over time will assist with the organization. It is always possible to add more features, but we found it too overwhelming to start with a large legacy system. We are still planning more features for our system, but by working along the way we have a finished product at each goal point.

7.4.3 Expansion

It is very important to demo the product to audiences at various points in development. Fruitful changes and improvements can thus be made because it will be based on user feedback.

Appendix A

Q-HSK Grammar

A.1 Notation

Literals are in `typewriter font`. Subexpressions are in *italic*. Expressions that can be repeated 1 or more times are followed by a $^+$. Expressions that can be repeated 0 or more times are followed by a *. A | represents an "or". Parentheses, (...) indicate that the grouping must be done together. [...] represents a regular expression.

A.2 Grammar

program→*lines*

lines→*comment lines*
→ *incls declist lines*
→*comment*
→ϵ

statements→*statement* | *statements statement*

statement→[*string*|*qreg*]*string* = *expr*;
→{ *statements* }
→`return` *expr*;
→`printf` (*quotedstring printlist*);
→`if` (*comparison*) *statement elses*

→`while` (*comparison*) *statement*
→*string* (*callist*);
→*decs*
→*CBLAS*
→*comment*
→ ϵ

$comparison \rightarrow (string \mid ints)\ (\mid \ \mid \ \& \mid == \mid >= \mid <= \mid > \mid < \mid !=)\ (string \mid ints)$

printlist→*printlist*, *somecall*
→ ϵ

incls→*incls* #`include` *includestring*
→*incls* #`include` *quotedstring*
→ ϵ

functions→*type string* (*arglist*) *statement*
functype string (*arglist*) *statement*

arglist→*args* | ϵ

args→*type string*
→*args*, type string

declist→*declist decs*
→ ϵ

decs→*type varlist*;

varlist→*string*
→*varlist*, *string*
→*qureg*
→*varlistqreg*, *qureg*

elses→`else` *statement*
→ ϵ

expr→*expr* $(+ \mid -)$ *expr*

$\rightarrow$*expr* (/ | $*$) *expr*
$\rightarrow$*expr* % *expr*
$\rightarrow$(*expr*)
$\rightarrow$*string* (*callist*)
$\rightarrow$ - *expr*
$\rightarrow$*ints*
$\rightarrow$*string*
$\rightarrow$*qreg*

callist$\rightarrow$*somecall*
$\rightarrow$*callist*, *somecall*
$\rightarrow\epsilon$

somecall$\rightarrow$*expr* | *quotedstring* | *type*

ints$\rightarrow$*integer* | *real* | *complex*

comment$\rightarrow$/* [$\wedge$/ *] */

`int`$\rightarrow$[0-9]$^+$

`real`$\rightarrow$[0-9]$^+$.[0-9]$^+$

`complex`$\rightarrow$(*real*, *real*)

`char`$\rightarrow$[a-zA-Z][a-zA-Z0-9]*

quotedstring$\rightarrow$“ [$\wedge$\n]* ”

string$\rightarrow$[a-zA-Z]$^+$

includestring$\rightarrow$[a-zA-Z]$^+$

type$\rightarrow$`int` | `real` | `complex` | `qreg` | `char`

`qreg`$\rightarrow$*string* [`int`]
$\rightarrow$*string* [*string*]
$\rightarrow$*string*[]

Appendix B

Shor's Prime Factorization Algorithm

Below is an example of Shor's algorithm for prime factorization in the Q-HSK programming language. It is a modified version of the code found in [2] and [3].

```
#include <iostream.h>
#include <math.h>
#include <time.h>
#include ‘‘complex’’
#include <stdlib.h>
#include <stdio.h>
int flag;
int main(int argc, char * argv[]) {
   flag=0;
/*Establish a random seed.*/
   srand(time(NULL));
   printf("Shor’s Algorithm Restrictions are:  1) The number to be
factored must be >= 15.  2) The number to be factored must be odd.
3) The number must not be prime.  4) The number must not be a prime
power");

   /*n is the number we are going to factor, get n.*/    int n;

   n = atoi(argv[1]);
   printf("n is %d\n", n);
```

```
   /*Test to see if n is factorable by Shor's algorithm.  Exit if
the number is even.*/
   if (n%2 == 0) {
      printf("Error, the number must be odd!\n");
      exit(0);
   } /*Exit if the number is prime.*/
   if (TestPrime(n)==1) {
      printf("Error, the number must not be prime!\n");
      exit(0);
   }
/*Prime powers are prime numbers raised to integral powers.  Exit
if the number is a prime power.*/
   if (TestPrimePower(n)==1) {
      printf("Error, the number must not be a prime power!\n");
      exit(0);
   }

   /*Now we must pick a random integer x, coprime to n.  Numbers
are coprime when their greatest common denominator is one.  One is
not a useful number for the algorithm.*/
   int x;
   x=0;
   x = 1 + (int)((n-1)*(real)rand()/(real)RAND_MAX);
   while (GCD(n,x) != 1 || x == 1) {
      x = 1 + (int)((n-1)*(real)rand()/(real)RAND_MAX);    }
   printf("Found x to be %d.\n", x);

   /*Now we must figure out how big a quantum register we need for
our input, n.  We must establish a quantum register big enough to
hold an equal superposition of all integers 0 through q - 1 where
q is the power of two such that n^2 <= q < 2n^2.*/
   int q;
   q = GetQ(n);
   printf("Found q to be %d.\n", q);

   /*Create the register.  QuReg star reg1 = new QuReg(RegSize(q)
- 1);*/
```

```
   qreg *reg1;
   reg1 = create(RegSize(q)-1, "reg1");

   printf("Made register 1 with register size = %d\n", (int)pow(2,
RegSize(q)-1));

   /*This array will remember what values of q produced for x^q mod
n.  It is necessary to retain these values for use when we collapse
register one after measuring register two.  In a real quantum computer
these registers would be entangled, and thus this extra bookkeeping
would not be needed at all.  The laws of quantum mechanics dictate
that register one would collapse as well, and into a state consistent
with the measured value in resister two.*/

   int *modex;
   modex = (int *)malloc(q*sizeof(int));

   /*This array holds the probability amplitudes of the collapsed
state of register one, after register two has been measured it is
used to put register one in a state consistent with that measured
in register two.*/

   complex * collapse;
   collapse = (complex *)malloc(q*sizeof(complex));

   /*This is a temporary value.*/
   complex tmp;

   /*This is a new array of probability amplitudes for our second
quantum register, that populated by the results of x^a mod n.*/

   complex * mdx;
   mdx = (complex *)malloc(((int)pow(2, RegSize(n)))*sizeof(complex));

   qreg *reg2;
   reg2 = create(RegSize(n),"reg2");

   printf("Created register 2 of size %d\n", (int)pow(2,RegSize(n)));
```

```
   /*This is a temporary value.*/
   int tmpval;

   /*This is a temporary value.*/
   int value;

   /*c is some multiple lambda of q div r, where q is q in this program,
and r is the period we are trying to find to factor n.  m is the
value we measure from register one after the Fourier transformation.*/
   real c,m;

   /*This is used to store the denominator of the fraction p div
den where p div den is the best approximation to c with den <= q.*/
   int den;

   /*This is used to store the numerator of the fraction p div den
where p div den is the best approximation to c with den <= q.*/
   int p;

   /*The integers e, a, and b are used in the end of the program
when we attempts to calculate the factors of n given the period it
measured.  Factor is the factor that we find.*/
   int e,a,b, factor;

   /*Shor's algorithm can sometimes fail, in which case you do it
again.  The done variable is set to 0 when the algorithm has failed.
Only try a maximum number of tries.*/
   int done;
   done = 0;
   int tries;
   tries = 0;
   while (done==0) {
      if (tries >= 5) {
         printf("There have been five failures, giving up.\n");
         exit(0);
      }
```

```
   /*Now populate register one in an even superposition of the integers
0 -> q - 1.*/

      reg1 = computeHadamard(reg1, (q-1));

   /*Now we preform a modular exponentiation on the superposed elements
of reg 1.  That is, perform x^a mod n, but exploiting quantum parallelism
a quantum computer could do this in one step, whereas we must calculate
it once for each possible measurable value in register one.  We store
the result in a new register, reg2, which is entangled with the first
register.  This means that when one is measured, and collapses into
a base state, the other register must collapse into a superposition
of states consistent with the measured value in the other..  The
size of the result modular exponentiation will be at most n, so the
number of bits we will need is therefore less than or equal to log2
of n.  At this point we also maintain a array of what each state
produced when modularly exponised, this is because these registers
would actually be entangled in a real quantum computer, this information
is needed when collapsing the first register later.*/

   /*This counter variable is used to increase our probability amplitude.*/
      tmp = (1.0,0.0);

   /*This for loop ranges over q, and puts the value of x^a mod n
in modex[a].  It also increases the probability amplitude of the
value of mdx[x^a mod n] in our array of complex probabilities.*/
      int i;
      i=0;
      while(i < q) {
/*We must use this version of modexp instead of C++ builtins as they
overflow when x^i > 2^31.*/
         tmpval = modexp(x,i,n);
         modex[i] = tmpval;
         mdx[tmpval] = mdx[tmpval] + tmp;
         i=i+1;
      }

   /*Set the state of register two to what we calculated it should
```

```
be.*/

      int k;
      k=0;

      while(k < pow(2, RegSize(n))) {
         reg2[k] = mdx[k];
         k=k+1;
      }

   /*Normalise register two, so that the probability of measuring
a state is given by summing the squares of its probability amplitude.*/

      reg2 = Norm(reg2, RegSize(n));

   /*Now we measure reg2.  */
      value = Measure(reg2, RegSize(n),"reg2");

   /*Now we must using the information in the array modex collapse
the state of register one into a state consistent with the value
we measured in register two.*/

      i=0;
      while(i < q) {
         if (modex[i] == value) {
            collapse[i] = (1.0,0.0);
         }
         else {
            collapse[i] = (0.0,0.0);
         }
         i=i+1;
      }

   /*Now we set the state of register one to be consistent with what
we measured in state two, and normalise the probability amplitudes.*/

      int r;
      r=0;
```

```
      while(r < pow(2, RegSize(q)-1)) {
         reg1[r] = collapse[r];
         r=r+1;
      }

      reg1 = Norm(reg1, RegSize(q)-1);

   /*Here we do our Fourier transformation.  */
      printf("\nBegin Discrete Fourier Transformation!\n");
      reg1 = DFT(reg1, q, RegSize(q)-1);

   /*Next we measure register one, due to the Fourier transform the
number we measure, m will be some multiple of lambda div r, where
lambda is an integer and r is the desired period.*/
      m = Measure(reg1, RegSize(q)-1, "reg1");

   /*If nothing goes wrong from here on out we are done.*/
      done = 1;

   /*If we measured zero, we have gained no new information about
the period, we must try again.*/
      if (m == 0) {
         printf("Measured 0 this trial a failure!\n");
         done = 0;
      }

   /*The DecMeasure subroutine will return -1 as an error code, due
to rounding errors it will occasionally fail to measure a state.*/
      if (m == -1) {
         printf("We failed to measure anything, this trial a failure!
Trying again.\n");
         done = 0;
      }

   /*If nothing has gone wrong, try to determine the period of our
function, and get factors of n.*/
      if (done==1) {
/*Now c =  lambda div r for some integer lambda.*/
```

```
        c = (real)m / (real)q;

   /*Calculate the denominator of the best rational approximation
to c with den < q.  Since c is lambda div r for some integer lambda,
this will provide us with our guess for r, our period.*/
        den = denominator(c, q);

   /*Calculate the numerator from the denominator.*/
        p = (int)floor(den * c + 0.5);

   /*Give user information.*/
        printf("measured %f, approximation for %f is %d / %d\n",
m, c, p, den);

   /*The denominator is our period, and an odd period is not useful
as a result of Shor's algorithm.  If the denominator times two is
still less than q we can use that.*/
        if (den % 2 == 1 && 2 * den < q ) {
           printf("Odd denominator, expanding by 2\n");
           p = 2 * p;
           den = 2 * den;
        }

   /*Initialise helper variables.*/
        e = 0;
        a = 0;
        b = 0;
        factor = 0;

   /* Failed if odd denominator.*/
        if (den % 2 == 1) {
           printf("Odd period found.  This trial failed.  Trying
again.\n");
        ]indent done = 0;
        }
        else {
/*Calculate candidates for possible common factors with n.*/
           printf("possible period is %d\n", den);
```

```
            e = modexp(x, den / 2, n);
            a = (e + 1) % n;
            b = (e + n - 1) % n;
            printf("%d ^ %d + 1 mod %d = %d,\n%d ^ %d - 1 mod %d
= %d\n", x, den / 2, n, a, x, den / 2, n, b);
            factor = max(GCD(n,a),GCD(n,b));
         }
      }

   /*GCD will return a -1 if it tried to calculate the GCD of two
numbers where at some point it tries to take the modulus of a number
and 0.*/
      if (factor == -1) {
         printf("Error, tried to calculate n mod 0 for some n.  Trying
again.\n");
         done = 0;
      }

      if ((factor == n || factor == 1) && done == 1) {
         printf("Found trivial factors 1 and %d.  Trying again.\n",
n);
         done = 0;
      }

   /*If nothing else has gone wrong, and we got a factor we are finished.
Otherwise start over.*/
      if (factor != 0 && done == 1) {
         printf("%d = %d * %d\n",n, factor, n / factor);       }
      else if (done == 1) {
         printf("Found factor to be 0, error.  Trying again.\n");
         done = 0;
      }
      tries=tries+1;
   }
   free(reg1);
   free(reg2);
   free(modex);
   free(collapse);
```

```
   free(mdx);
   return 1;
}
```

Appendix C

Q-HSK Library Functions

C.1 Classical Functions

C.1.1 `TestPrime`

The Q-HSK function `TestPrime` has the form:
`int TestPrime(int n)`

n must be the number to be tested. `TestPrime` checks if n is a prime number and returns 0 if true and 1 if false.

C.1.2 `TestPrimePower`

The Q-HSK function `TestPrimePower` has the form:
`int TestPrimePower(int n)`

n must be the number to be tested. `TestPrime` checks is n is a power of a prime number and returns 0 if true and 1 if false.

C.1.3 `RegSize`

The Q-HSK function `RegSize` has the form:
`int RegSize(int a)`

`RegSize` takes the integer a and returns the number of bits needed to represent that integer.

C.1.4 `GetQ`

The Q-HSK function `GetQ` has the form:
`int GetQ(int n)`

`GetQ` takes an integer n and returns the power required by the quantum computation.

C.1.5 `denominator`

The Q-HSK function `denominator` has the form:
`int denominator(real c, int qmax)`

`denominator` returns the denominator q of the best rational denominator q for approximating p/q for c with $q < qmax$.

C.2 Quantum Functions

All quantum library functions require the Q-HSK library `complex`.

C.2.1 `modexp`

The Q-HSK function `modexp` has the form:
`int modexp(int x, int a, int n)`

`modexp` takes x, a, and n and returns $x^a modn$.

C.2.2 `computeHadamard`

The Q-HSK function `computeHadamard` has the form:
`qreg * computeHadamard(qreg *z, int size)`

z must be a pointer to a type `qreg`. *size* must be an integer containing the desired size of the `qreg`. It computes the hadamard transform on the quantum register z of size *size*, but will output a register of size 2^size.

C.2.3 `DFT`

The Q-HSK function `DFT` has the form:
`qreg * DFT(qreg *reg, int q, int reg_size)`

The computes the quantum fourier transform of the quantum register *reg* and returns a pointer to a quantum register with the values after the transformation.

C.2.4 `Norm`

The Q-HSK function `Norm` has the form:
`qreg * Norm(qreg *State, int reg_size)`

State must be a pointer to a type `qreg`. *reg_size* must be an integer containing the size of the quantum register. `Norm` normalizes the states of the quantum register.

C.2.5 `Measure`

The Q-HSK function `Measure` has the form:
`int Measure(qreg *State, int reg_size, char s[])`

State must be a pointer to a type `qreg`. *reg_size* must contain the size of the quantum register. *s[]* must contain the name of the register *State*. `Measure` measures the quantum register and collapses it to a classical state.

Appendix D

Implementation Source Code

D.1 The Lex File

D.2 The Yacc File

D.3 The Translation File

D.4 The Java File

Bibliography

[1] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, UK, 2000

[2] B. Omer. *A Procedural Formalism for Quantum Computing*. `http://tph.tuwien.ac.at/ oemer`, 1998

[3] M. Hayward. `alumni.imsa.edu/ matth/cs299/node25.html`. 2002