

QLang: Qubit Language (Final Report)

Christopher Campbell

Clément Canonne

Sankalpa Khadka

Winnie Narang

Jonathan Wong

December 17, 2014

Contents

1	An Introduction to the Language	4
1.1	Background: Quantum Computing	4
1.1.1	Dirac notation for quantum computation	5
1.1.2	Quantum Algorithms	6
1.2	Goal and objectives	6
2	QLang in practice: a Tutorial	7
2.1	Basics and syntax	7
2.2	Control structures, built-in functions and conversions	8
2.3	Diving in: Deutsch–Jozsa Algorithm	10
3	Reference Manual	13
3.1	Lexical conventions	13
3.1.1	Character set	13
3.1.2	Literals	13
3.1.3	Constants	13
3.1.4	Identifier (names)	13
3.1.5	Keywords	14
3.1.6	Expression Operators	14
3.1.7	Seperators	14
3.1.8	Elementary operations and spacing	14
3.2	Objects and types	14
3.2.1	Objects and lvalues	14
3.2.2	Valid types	14
3.3	Conversions	15
3.4	Expressions	15
3.4.1	Operator Precedence	15
3.4.2	Literals	15
3.4.3	Primary Expressions	15
3.4.4	Unary Operators	17
3.4.5	Binary Operators	18
3.4.6	Assignment Operators	20
3.5	Declarations	20
3.5.1	Type Specifiers	20
3.5.2	Declarator List	21

3.5.3	Meaning of Declarators	21
3.6	Statements	21
3.6.1	Expression statements	21
3.6.2	The if-else statement	22
3.6.3	The for loop	22
3.6.4	The while loop	23
3.7	Scope rules	23
3.8	Constant expressions	23
3.9	Examples	24
3.9.1	Solving Quantum Computation Problem	24
3.9.2	Simulation of Quantum Algorithm	25
4	Project Plan and Organization	29
4.1	Project Management	29
4.1.1	Planning	29
4.1.2	Specification	29
4.1.3	Development	29
4.1.4	Testing	30
4.2	Style Guide	30
4.3	Project Timeline	30
4.4	Roles and Responsibilities	30
4.5	Software Development Environment	30
4.6	Project Log	31
5	Architectural Design	39
5.0.1	Block Diagram	39
5.0.2	Components	39
6	Test Plan	41
6.1	Testing Phases	41
6.1.1	Unit Testing	41
6.1.2	Integration Testing	41
6.1.3	System Testing	41
6.2	Automation and Implementation	41
6.3	Sample test programs	42
7	Lesson Learned	50
7.1	Christopher Campbell	50
7.2	Sankalpa Khadka	50
7.3	Winnie Narang	51
7.4	Jonathan Wong	51
A	More on Quantum Computing	52
A.1	Common quantum gates	52
A.2	Tensor product and its properties	53

B	Source Code	54
B.1	Scanner	54
B.2	Parser	55
B.3	AST	58
B.4	Analyzer	61
B.5	SAST	72
B.6	Generator	75
B.7	Scripts	80
	B.7.1 Makefile	80
	B.7.2 Compilation script	81
	B.7.3 Testing script	81
B.8	Programs	84
	B.8.1 Demo	84
	B.8.2 Successful Test cases	88
	B.8.3 Execution output of successful cases	96
	B.8.4 Failed cases	100
	B.8.5 Output for failed cases	104
B.9	C++ Helper files	105
	B.9.1 qlang.cpp	105
	B.9.2 qlang.hpp	107

Chapter 1

An Introduction to the Language

The **QLang** language is a scientific tool that enables easy and simple simulation of quantum computing on classical computers. Featuring a clear and intuitive syntax, **QLang** makes it possible to take any quantum algorithm and implement it seamlessly, while conserving both the overall structure and syntactical features of the original pseudocode. The **QLang** code is then compiled to C++, allowing for an eventual high-performance execution – a process made simple and transparent to the user, who can focus on the algorithmic aspects of the quantum simulation.

1.1 Background: Quantum Computing

In classical computing, data are stored in the form of binary digits or bits. A *bit* is the basic unit of information stored and manipulated in a computer, which in one of two possible distinct states (for instance: two distinct voltages, on and off state of electric switch, two directions of magnetization, etc.). The two possible values/states of a system are represented as binary digits, 0 and 1. In a quantum computer, however, data are stored in the form of *qubits*, or quantum bits. A quantum system of n qubits is a Hilbert space of dimension 2^n ; fixing any orthonormal basis, any *quantum state* can thus be uniquely written as a linear combination of 2^n orthogonal vectors $\{|i\rangle\}_i$ where i is an n -bit binary number.

Example 1.1.1. A 3 qubit system has a canonical basis of 8 orthonormal states denoted $|000\rangle$, $|001\rangle$, $|010\rangle$, $|011\rangle$, $|100\rangle$, $|101\rangle$, $|110\rangle$, $|111\rangle$.

To put it briefly, while a classical bit has only two states (either 0 or 1), a qubit can have states $|0\rangle$ and $|1\rangle$, or any linear combination of states also known as a *superposition*:

$$|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where $\alpha, \beta \in \mathbb{C}$ are any complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$.

Similarly, one may recall that logical operations, also known as *logical gates*, are the basis of computation in classical computers. Computers are built with circuit that is made up of logical gates. The examples of logical gates are AND, OR, NOT, NOR, XOR, etc. The analogue for quantum computers, *quantum gates*, are operations which are a *unitary transformation* on qubits. The quantum gates are represented by matrices, and a gate acts on n qubits is represented by $2^n \times 2^n$ unitary matrix¹. Analogous to the classical computer which is built from an electrical circuit

¹That is, a matrix $U \in \mathbb{C}^{2^n \times 2^n}$ such that $U^\dagger U = I_{2^n}$, where \cdot^\dagger denotes the Hermitian conjugate.

containing wires and logic gates, quantum computers are built from quantum circuits containing “wires” and quantum gates to carry out the computation.

More on this, as well as the definition of the usual quantum gates, can be found in [Appendix A](#).

1.1.1 Dirac notation for quantum computation

In quantum computing, *Dirac notation* is generally used to represent qubits. This notation provides concise and intuitive representation of complex matrix operations.

More precisely, a column vector $\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$ is represented as $|\psi\rangle$, also read as “ket psi”. In particular,

the computational basis states, also known as *pure states* are represented as $|i\rangle$ where i is a n -bit binary number. For example,

$$|000\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |001\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |010\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \dots, |101\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |110\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, |111\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Similarly, the row vector $\begin{bmatrix} c_1^* & c_2^* & \dots & c_n^* \end{bmatrix}$, which is also complex conjugate transpose of $|\psi\rangle$, is represented as $\langle\psi|$, also read as “bra psi”.

The inner product of vectors $|\varphi\rangle$ and $|\psi\rangle$ is written $\langle\varphi, \psi\rangle$. The tensor product of vectors $|\varphi\rangle$ and $|\psi\rangle$ is written $|\varphi\rangle \otimes |\psi\rangle$ and more commonly $|\varphi\rangle|\psi\rangle$. We list below a few other mathematical notions that are relevant in quantum computing:

- z^* (complex conjugate of elements)
if $z = a + ib$, then $z^* = a - ib$.
- A^* (complex conjugate of matrices)
if $A = \begin{bmatrix} 1 & 6i \\ 3i & 2 + 4i \end{bmatrix}$ then $A^* = \begin{bmatrix} 1 & -6i \\ -3i & 2 - 4i \end{bmatrix}$.
- A^T (transpose of matrix A)
if $A = \begin{bmatrix} 1 & 6i \\ 3i & 2 + 4i \end{bmatrix}$ then $A^T = \begin{bmatrix} 1 & 3i \\ 6i & 2 + 4i \end{bmatrix}$.
- A^\dagger (Hermitian conjugate (adjoint) of matrix A)
Defined as $A^\dagger = (A^T)^*$; if $A = \begin{bmatrix} 1 & 6i \\ 3i & 2 + 4i \end{bmatrix}$ then $A^\dagger = \begin{bmatrix} 1 & -3i \\ -6i & 2 - 4i \end{bmatrix}$
- $\| |\psi\rangle \|$ (ℓ_2 norm of vector $|\psi\rangle$)
 $\| |\psi\rangle \| = \sqrt{\langle\psi|\psi\rangle}$. (This is often used to normalize $|\psi\rangle$ into a unit vector $\frac{|\psi\rangle}{\| |\psi\rangle \|}$.)

- $\langle \varphi | A | \psi \rangle$ (inner product of $|\varphi\rangle$ and $A|\psi\rangle$).
Equivalently², inner product of $A^\dagger|\varphi\rangle$ and $|\psi\rangle$

1.1.2 Quantum Algorithms

A quantum algorithm is an algorithm that, in addition to operations on bits, can apply quantum gates to qubits and measure the outcome, in order to perform a computation or solve a search problem. Inherently, the outcome of such algorithms will be probabilistic: for instance, a quantum algorithm is said to *compute a function f on input x* if, for all x , the value $f(x)$ it outputs is correct with high probability. The representation of a quantum computation process requires an input register, output register and unitary transformation that takes a computational basis states into linear combination of computational basis states. If x represents an n qubit input register and y represents an m qubit output register, then the effect of a unitary transformation U_f on the computational basis $|x\rangle_n |y\rangle_m$ is represented as follows:

$$U_f(|x\rangle_n |y\rangle_m) = |x\rangle_n |y \oplus f(x)\rangle_m, \quad (1.1)$$

where f is a function that takes an n qubit input register and returns an m qubit output and \oplus represents mod-2 bitwise addition.

1.2 Goal and objectives

QLang has been designed with a handful of key characteristics in mind:

Intuitive. Any student or researcher familiar with quantum computing should be able to trans-
pose and implement their algorithms easily and quickly, without wasting time struggling to
understand idiosyncrasies of the language.

Specific. The language has one purpose – implementing quantum algorithms. Though, the language
supports many linear algebraic computation, it is mainly aims for quantum computation.
Anything that is not related to nor useful for this purpose should not be – and is not – part of
QLang (e.g., the language does not support strings).

Simple. Matrices, vector operations are pervasive in quantum computing – thus, they must be
easy to use and understand. All predefined structures and functions are straightforward to
use, and have no puzzling nor counter-intuitive behavior.

In a nutshell, QLang is simple, includes everything it should – and nothing it should not.

²Recall that we work in a complex Hilbert space: the inner product is a sesquilinear form.

Chapter 2

QLang in practice: a Tutorial

2.1 Basics and syntax

A QLang file (extension `.ql` by convention) comprises several functions, each of them having its own variables. Once compiled, a program will start by calling the `compute()` function that must appear in the `.ql` file, and whose prototype is as follows:

```
1 def compute(): int trial {  
    trial = 10;  
3 }
```

In particular, the main entry point `compute()` receives no argument and, automatically prints the return variable defined in the function declaration. The execution of above program prints 10. Note also that QLang is case-sensitive: `compute` and `Compute` would be two different functions (however, indentation is completely unrestricted).

Comments in the language are single-line, and start with a `#`: everything following this symbol, until the next line return, will be ignored by the compiler. Furthermore, a function is defined (and declared – there is no forward declaration) by the keyword `def` followed by the details of the function:

```
1 def function_name(type1 param1, type2 param2, ..., typek paramk): type returnvar {  
    # variable declarations  
3    # body of the function  
}
```

The valid types in QLang for parameters, return variables and variables are `int`, `float`, `comp`, `mat`: respectively integers, real numbers, complex numbers and matrices (the latter including, as we shall see, qubits). In the above, the return variable `returnvar` is available in the body of the function, and its value will be returned at the end of the function call. All other local variables must be declared, at the beginning of the function body: in particular, it is not possible to mix variable declaration and assignment:

```
def foo( mat bar ): mat blah {  
2   int bleh;           # OK  
   int bluh = 0;        # Not OK: parsing error  
4   comp blih, bloh;     # Not OK: one variable at a time  
   comp blih; comp bloh; #OK
```



```

6   bleh = 5;                                # OK
   bleh = bleh+1                             # Not OK: missing semicolon
8   bleh = bleh * 4 +
   2^bleh;                                   # OK: statements can span several lines
10  bleh = bleh-1; bleh = 2*bleh;             # OK: several statements per line
   blah = bleh * bar;                        # OK: blah is the returned variable
12 }

```

As exemplified above, each statement (declaration, assignment, operation) can span any number of lines, and end with a semicolon.

Qubits, matrices and vectors. Before turning to the flow control structures, recall that QLang is designed specifically for the sake of implementing quantum algorithms; as such, it supports the usual quantum notations for bra and kets (although it stores and recognizes them as of type `mat`):

```

   mat idt;
2  mat vct;
   mat qub;
4  qub = |11>;                               # this is a ket of dirac notation
   qub = <01|;                               # this is a bra of dirac notation
6  idt = [(1,0)(0,1)];                       # this is a matrix
   vct = [(1,2,C(3.2 + 1.I))];               # this is a vector (with complex entries)
8  vct = qub;                                # this is OK

```

In the above, the 3 variables have the same type – the difference is only syntactical, in order to provide the user with an intuitive way to program the quantum operations.

2.2 Control structures, built-in functions and conversions

Now that the basic syntax of the language has been described, it is time to have a look at the fundamental blocks of any algorithm: the control structures, such as loops and conditional statements.

Loops. QLang supports two sorts of loop, the `for` and `while` statements. While their behaviors are illustrated below, it is important to remember two features of the `for` loop: namely, that (a) the loop index must be a variable declared beforehand; and (b) that the (optional) keyword `by` allows to set the increment size by any integer, even negative.

```

   int i; # Will be used as 'for' loop index
2  int a;

4  for( i from 0 to 2 by 1 ) # OK
   a=a+5;

6  for( i from 2 to 0 by -1 ) # OK
8  {
   a=a*10;
10  continue; # going to next iteration: the next instruction will never be executed.
   print(a);
12 }

14 for( i from 1 to 10 ) # OK: missing "by 1" is implicit

```

```

16 {
    a=a-3;
    break; # leaves the loop.
18 }

20 while( a leq 10 ) # OK
    a=a+1;
22
24 while( a neq 0 ) # OK
    {
        a = (a+1) ;
        continue;
        print(0); # never reached
28 }

```

As shown above, braces are optional when the body of the loop comprises only one line.

Conditional constructs. As in many languages, QLang supports a C-like if...else construct:

```

2     if( predicate )
    {
4         # Do something
    }
    else
6     {
        # Do something else
8     }

```

The predicate can be any expression evaluating to an integer: if non-zero, the if statement is entered; otherwise, the (optional) else statement is entered, if it exists. Note that QLang does not provide a builtin construct `elseif`, but instead relies on a nested combination of `if` and `else`:

```

2     if(z eq 5) a = 0;
    a = a - 2;
4     if( z leq 5 )
    {
6         a = 0;
    }
8     else
    {
10         a = 10;
        b = 24;
12     }

14     if( a gt 100 )
    {
16         print(b); # a > 100
    }
18     else if( a eq 10 )
    {
20         print(a);
    }

```

Builtin functions and operators. As shown in the previous two examples, QLang provides builtin constructs to perform basic or fundamental tasks:

Comparison operators: `gt`, `lt`, `geq`, `leq`, `eq`, `neq` take two operands a , b , and return 0 (resp. 1) if respectively $a > b$, $a < b$, $a \geq b$, $a \leq b$, $a = b$ and $a \neq b$;

Builtin functions: these are convenient functions such as `print`, `printq` (for qubit syntax), or mathematical ones applying to matrices such as `norm`, `adj`, to complex values (`sin`, `im`, ...) or to 0/1 integers (“Booleans”) such as `and`, `xor`.

Operators: the language supports the usual unary (negation `-`, logical negation `not`), binary (addition `+`, subtraction `-`, exponentiation `^...`) operators, as well as some more specific ones (tensor product `@`).

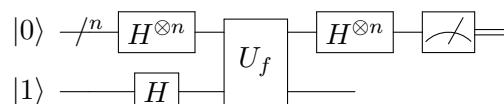
The complete list of these functions, operators and builtin constants can be found in [Chapter 3](#).

Implicit conversions. Implicit conversions for some operators such as `eq` is possible, according to the following rule: `int` \rightsquigarrow `float` \rightsquigarrow `comp` \rightsquigarrow `mat`. However, the language is otherwise strongly typed: it is not possible to assign a complex number to a variable of type `mat`, for instance.

2.3 Diving in: Deutsch–Jozsa Algorithm

To illustrate and describe the process of writing in QLang, this section will walk the reader through the implementation of one of the most emblematic quantum examples, namely *Deutsch-Jozsa Algorithm*. The goal of this algorithm is to answer the following question: given query access to an unknown function $f: \{0, 1\}^n \rightarrow \{0, 1\}$, promised to be either constant or balanced¹, which of the two holds? Classically, it is easy to see that this requires (in the worst case) querying just over half the solution space, that is $2^{n-1} + 1$ queries. Quantumly, the Deutsch-Jozsa algorithm enables us to answer this question with just *one* query!

The circuit performing the algorithm is given below:



To implement it in QLang, we first have to implement the n -fold Hadamard gate $H^{\otimes n}$; recalling that the Hadamard gate H is a built-in operator of the language, this can be done as follows:

```

1  def hadamard(int n): mat gate{
2      #returns Hadamard gate of 2^n dimensions
3      int i;
4      gate = H;
5      for(i from 1 to n-1 by 1){
6          gate = gate @ H;
7      }
8  }
```

¹ f is said to be balanced if $f(x) = 0$ for exactly half of the inputs $x \in \{0, 1\}^n$; or, equivalently, if $\mathbb{E}_x[f(x)] = \frac{1}{2}$.

Now, to implement the measurement gate (or, more precisely, to return the measurement matrix), we write the following code that takes a ket $|x\rangle$ and returns the matrix $|x\rangle\langle x|$:

```

def measure(mat top): mat result{
2     # returns the measurement matrix
    mat ad;
4     mat ad = Adj(top);
    result = top * ad;
6 }

```

(Note that $|x\rangle\langle x|$ was written as $|x\rangle$ adjoint($|x\rangle$), which is performed above using the transparent conversion between vectors and matrices provided by the language.)

Since the qubit in the top register is n -bit, we can write a function that allows us to create such a qubit for any n .

```

def topqubit(int n): mat input{
2     # n-bit qubit
    int i;
4     input = |0>;
    for(i from 1 to n-1 by 1){
6         input = input @ |0>;
    }
8 }

```

Once all the “building blocks” (gates) of the algorithm have been implemented, we can write down the algorithm *as it appears from the circuit above*: the function takes as argument the parameter size n , as well as the unitary matrix implementing the quantum gate U_f (the access to the unknown function f), and returning either 0 or 1, depending on whether the function is constant or balanced.

```

def deutsch(int n, qubit top, mat U): float outcomeZero{
2     mat bottom; mat top; mat input;
    mat hadtop; mat meas;
4
    bottom = |1>;
6     top = topqubit(n);
    input = top @ bottom;
8
    hadtop = hadamard(n);
10    input = (hadtop @ H)*input;
    input = U * input;
12    input = (hadtop @ IDT)*input;
    meas = measure(top);
14
    input = (meas @ IDT)* input;
16    outcomeZero = norm(input);
18 }

```

Finally, we can call (and test) our algorithm by defining two unitary transformations (here U_b and U_c) and testing our function on them – and print the output. This is done by writing the entry point function, `compute`:

```

def compute (): float outcome{
2     int n; mat Ub; mat Uc;

4     n = 1;
    Ub = [(1,0,0,0) (0,1,0,0) (0,0,0,1) (0,0,1,0)];
6     Uc = [(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)];

8     outcome = deutsch(n, Ub);
    print(outcome);

10    outcome = deutsch(n, Uc);
    print(outcome);

12

14    n = 2;
    Ub = [(1,0,0,0,0,0,0,0)
16          (0,1,0,0,0,0,0,0)
          (0,0,1,0,0,0,0,0)
18          (0,0,0,1,0,0,0,0)
          (0,0,0,0,0,1,0,0)
20          (0,0,0,0,1,0,0,0)
          (0,0,0,0,0,0,0,1)
22          (0,0,0,0,0,0,1,0)];

24    outcome = deutsch(n, Ub);
}

```

The above program will print 0, 1, 0 for balanced function, constant function and balanced function respectively.

Chapter 3

Reference Manual

3.1 Lexical conventions

There are five kinds of tokens in the language, namely (1) literals, (2) constants, (3) identifiers, (4) keywords, (5) expression operators, and (6) 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.

3.1.1 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`.

3.1.2 Literals

There are three sorts of literals in the language, namely *integer*, *float*, and *complex*. All three can be negative or positive (negation is achieved by applying the unary negative operator to them). Integers are given by the regular expression `['0'-'9'] +`, floats are given by `['0'-'9'] + ' . ' ['0'-'9'] *`, and complex are given by `C(F) | C(F+FI) | C(FI)`, where F is any floating point number.

3.1.3 Constants

There are several built-in numerical constants that can be treated as literals, they include:

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

pi the constant π . Has type `float`.

3.1.4 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.

3.1.5 Keywords

The following identifiers are reserved for keywords, using them as function of variable name will result in an error at compilation time.

```
int float comp mat C I def return eq neq lt gt leq geq
true false not and or xor norm trans det adj conj unit @
im re sin cos tan if else for from to by while break continue
```

3.1.6 Expression Operators

Expression operators are discussed in detail in section 3.4, Expressions.

3.1.7 Seperators

Commas are used to separator lists of actual and formal parameters, colons are used to separate the rows of matrices, semi-colons are used to terminate statements, and the hash-symbol (#) is used to begin a comment. Comments extends until the next carriage return. Multi-line comments are not supported.

3.1.8 Elementary operations and spacing

An operation, or language elementary unit, starts from the end of the previous one, and ends whenever a semicolon is encountered. Whitespace does not play any role, except as separators between tokens; in particular, indentation is arbitrary.

3.2 Objects and types

3.2.1 Objects and lvalues

As in C, “an object is a manipulatable region of storage; an lvalue is an expression referring to an object.”

3.2.2 Valid types

The language features 4 elementary types, namely `int`, `float`, `comp`, `mat`. Is also valid, any type that inductively can be built from a valid type as follows:

- elementary types are valid;
- an *matrix* of a valid type is valid. Matrices have fixed size (that must be declared at compilation time), and are comprised of any elements of any type (that is, a matrix can have elements of non-necessarily identical types);
- a *function* taking as input a fixed number of elements from (non-necessarily identical) valid types, and returning a valid type.

3.3 Conversions

Applying unary or binary operators to some values may cause an implicit conversion of their operands. In this section, we list the possible conversions, and their expected result – any conversion not listed here is impossible, and attempting to force it would generate a compilation error.

- `int` \rightarrow `float`
- `float` \rightarrow `comp`
- `int` \rightarrow `comp`

The equality and comparison operators (`eq`, `leq`, `geq`, `lt`, `gt`) will perform the implicit conversions above, when they make sense. For instance, `0 eq C(0.0 + 0.0I)` is valid, and the comparison will be between two complex numbers (after the first operand is converted into a `comp`). Similarly, `1 lt 2.5` is valid, the integer left-hand side being cast into a float (note that `leq`, `geq`, `lt`, `gt` are not defined for complex numbers, but only `int` and `float`).

3.4 Expressions

3.4.1 Operator Precedence

Unary operators have the highest precedence, followed by binary operators, and then assignment. The precedence and associativity within each type of operator is given in the table below. The lists of operators are read left to right in order of descending precedence. Also, the `|` symbol is used to group operators of the same precedence.

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

3.4.2 Literals

Literals are integers, floats, complex numbers, 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 `comp`, qubits 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.

3.4.3 Primary Expressions

identifier

Identifiers are primary expression. 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`).

literals

Literals are primary expression. They are described above.

(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).

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.

[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 parenthesized. Commas separate expressions into matrix columns and parentheses group expressions into matrix rows. The expressions can be of type **int**, **float**, and **comp** and need not be identical. Additionally, the number of expressions in each row of the matrix must be the same. An example matrix is shown below.

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

<expression|

Expressions with a less than sign on the left and a bar on the right are primary expression. Such expressions are used to define qubits and therefore are of type **mat**. 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.

```
1 mat b_qubit = <0100|;
```

$|expression\rangle$

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

```
1 int a = 001;
   mat k_qubit = |a>;
```

3.4.4 Unary Operators

not $expression$

The result is a 1 or 0 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.

re $expression$

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

im $expression$

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

norm $expression$

The result is the norm of the $expression$. The type of the expression must be mat. The result has type float, and corresponds to the 2-norm; in the case of comp or float.

unit $expression$

The result is a 1 or 0 indicating whether the expression is a unit matrix. The type of the expression must be mat.

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 $expression$.

det $expression$

The result is the determinant of the $expression$. The type of the expression must be mat. The result has type comp.

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*.

conj *expression*

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

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 `comp`. The result has type `float` if the expression is of type `int` or `float` and type `comp` if the expression is of type `comp`.

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 `comp`. The result has type `float` if the expression is of type `int` or `float` and type `comp` if the expression is of type `comp`.

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 `comp`. The result has type `float` if the expression is of type `int` or `float` and type `comp` if the expression is of type `comp`. (If an error occurred because of a division by zero, a runtime exception is raised.)

3.4.5 Binary Operators

expression* ^ *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 `comp`. 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 `comp`, the result is of type `comp`; if neither expressions are `comp`, but at least one is `float`, the result is of type `float`.

expression* * *expression

The result is the product of the *expressions*. The type considerations are the same as they are for *expression* ^ *expression* except that it also allows for matrices.

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* ^ *expression*. Integer division is rounded towards 0 and truncated. (If an error occurred because of a division by zero, a runtime exception is raised.)

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* ^ *expression*.

expression + expression

The result is the sum of the *expressions*. The types of the expressions must be of type `int`, `float`, `comp`, or `mat`. If at least one *expression* is a `comp`, the result is of type `comp`; 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).

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*.

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*.

expression eq expression

The result is a 1 or 0 indicating if it is true or false that the two *expression* are equivalent. The type of the expressions must either be the same, or one of the two should be implicitly convertible to the other type (e.g., `0.2 eq 1`, where the right-hand side is an `int` that can be cast into a `float`).

expression lt expression

The result is a 1 or 0 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`.

expression gt expression

The result is a 1 or 0 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`.

expression leq expression

The result is a 1 or 0 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`.

expression geq expression

The result is a 1 or 0 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`.

expression or expression

The result is a 1 or 0 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**.

expression and expression

The result is a 1 or 0 indicating the logical *and* of the *expressions*. The type considerations are the same as they are for *expression* or *expression*.

expression xor expression

The result is a 1 or 0 indicating the logical *xor* of the *expressions*. The type considerations are the same as they are for *expression* or *expression*.

3.4.6 Assignment Operators

Assignment operators have left associativity

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`, and `mat`.

3.5 Declarations

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

declaration:
type-specifier declarator-list

3.5.1 Type Specifiers

There are five main type specifiers:

type-specifier:

`int`
`float`
`comp`
`mat`

3.5.2 Declarator List

The declarator-list consist of either a single declarator, or a series of declarators separated by commas.

declarator-list:
declarator
declarator , declarator-list

A declarator refers to an object with a type determined by the type-specifier in the overall declaration. Declarators can have the following form

declarator:
identifier
declarator ()
(declarator)

3.5.3 Meaning of Declarators

Each declarator that appears in an expression is a call to create an object of the specified type. Each declarator has one identifier, and it is this identifier that is now associated with the created object.

If declarator D has the form

D ()

then the contained identifier has the type "function" that is returning an object. This object has the type which the identifier would have had if the declarator had just been D.

Parentheses in declarators do not change the the type of contained identifier, but can affect the relations between the individual components of the declarator.

Not all possible combinations of the above syntax are permitted. There are certain restrictions such as how array of functions cannot be declared.

3.6 Statements

3.6.1 Expression statements

Expression statements are the building blocks of an executable program. As the name suggests, expression statements are nothing but expressions, delimited by semicolons. Expressions can actually be declarations, assignments, operations or even function calls. For example,

² `x = a + 3;`

is a valid expression statement, and so is

```
2 print(a);
```

3.6.2 The if-else statement

The **if-else** statement is used for selectively executing statements based on some condition. Essentially, if the condition following the **if** keyword is satisfied, the specified statements get executed. To specify what happens if the condition does not evaluate to true, we have the **else** keyword. In case we want to evaluate more than one condition at a time, **if-else** can be nested.

```
2
4
6
8
10
12
14
16
if( condition ){
}
else{
}

Example:
if ( x eq 5) {
    print(5);
} else if (x eq 3) {
    print(3);
} else {
    print(0);
}
```

3.6.3 The for loop

The **for** statement is used for executing a set of statements a specified number of times. The statements within the for loop are executed as long as the value of the variable is within the specified range. As soon as the value goes out of range, control comes out of the for loop. To ensure termination, each iteration of the for loop increments/decrements the value of the variable, bringing it one step closer to the final value that is to be achieved.

By default, increment or decrement is by 1. However, if the desired increment is something other than one, the optional keyword **by** lets you specify that explicitly.

An example of for loop, increment by 2 is as follows:

```
2
int k;
for( k from 1 to 10 by 2 ) {
}
```

The two keywords **break** and **continue** can be used inside the body of the loop to respectively exit it prematurely, or skip to the next iteration.

3.6.4 The while loop

The **while** statement is used for executing a set of statements as long as a predicate (condition) is true. As soon as the predicate is no longer satisfied, control comes out of the **while** loop. An example of **while** loop is given below:

```
1  while( k leq 100 ) {  
3    k = k^2;  
    }
```

The two keywords **break** and **continue** can be used inside the body of the loop to respectively exit it prematurely, or skip to the next iteration.

3.7 Scope rules

Name bindings have a block scope. That is to say, the scope of a name binding is limited to a section of code that is grouped together. That name can only be used to refer to associated entity in that block of code. Blocks of code in QLang are delimited by the opening curly brace ('{') at the start of the block, and the closing curly brace ('}') at the end of the block.

Within a program, variables may be declared and/or defined in various places. The scope of each variable is different, depending on where it is declared. There are three primary scope rules.

If a variable is defined at the outset/outer block of a program, it is visible everywhere in the program.

If a variable is defined as a parameter to a function, or inside a function/block of code, it is visible only within that function.

Declarations made after a specific declaration are not visible to it, or to any declarations before it.

For instance, consider the following snippet.

```
1  int x = 5;  
3  int y = x + 10; # this works  
5  int z = a + 100; # this does not  
7  int a = 200;
```

3.8 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.

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

The *Hadamard gate* is defined by the matrix:

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

3.9 Examples

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

3.9.1 Solving Quantum Computation Problem

Problem1

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

```

2 def compute() : mat evaluate () {
    mat a;
4   a = |00>;
    evaluate = (H @ X) * a;
6   printq(evaluate);
8 }

```

Problem 2

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

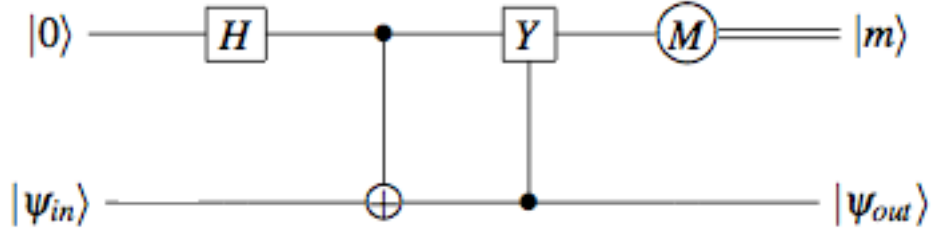


Figure 3.1: Quantum Circuit

```

1 def measure(mat top): mat outcome{
    mat ad;
3
    ad = adj(top);
5    outcome = top*ad;
    }
7
9 def outcomezero(mat bottom) : float probability{
11
    mat top; mat input;
    mat had; mat cnot; mat ynot;

```

```

13     mat output;  mat meas;

15     top = |0>;
    input = top @ bottom;

17     had = H @ IDT;
    cnot = [(1,0,0,0)
19           (0,1,0,0)
           (0,0,0,1)
21           (0,0,1,0)];

23

    ynot = [(1,0,0,0)
25           (0,0,0,-1)
           (0,0,1,0)
27           (0,-1,0,0)];

29     output = (ynot*(cnot*(had*input)));

31     printq(output);

33     probability = norm(output);

35 }

37 def compute() : float outcome{

39     mat bottom;

41     bottom = |1>;
    outcome = outcomezero(bottom);
43     print(outcome);

45 }

```

Output

```

1 (0.707107)|10> + (-0.707107)|11>
1

```

3.9.2 Simulation of Quantum Algorithm

Deutsch Jozsa Algorithm

```

def measure (mat top) : mat outcome{
2
    mat ad;

4
    ad = adj(top);
    outcome = top * ad;
6
}

8
def hadamard (int n) : mat gate{
10
    int i;
    gate = H;
12

    for (i from 0 to n-1 by 1){
14

```

```

16         gate = gate @ H;
17     }
18 }
19
20 def topqubit (int n) : mat input{
21
22     int i;
23     input = |0>;
24
25     for (i from 0 to n-1 by 1){
26         input = input @ |0>;
27     }
28 }
29
30 def deutsch (int n, mat U) : float outcomeZero{
31
32     mat bottom; mat top; mat input;
33     mat hadtop; mat meas;
34
35     bottom = |1>;
36     top = topqubit(n);
37     input = top @ bottom;
38
39     hadtop = hadamard(n);
40     input = (hadtop @ H)*input;
41     input = U * input;
42     input = (hadtop @ IDT)*input;
43     meas = measure(top);
44
45     input = (meas @ IDT)* input;
46     outcomeZero = norm(input);
47 }
48
49 def compute () : float outcome{
50
51     int n; mat Ub; mat Uc;
52
53     n = 1;
54     Ub = [(1,0,0,0) (0,1,0,0) (0,0,0,1) (0,0,1,0)];
55     Uc = [(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)];
56
57     outcome = deutsch(n, Ub);
58     print(outcome);
59
60     outcome = deutsch(n, Uc);
61     print(outcome);
62
63     n = 2;
64     Ub = [(1,0,0,0,0,0,0,0)
65           (0,1,0,0,0,0,0,0)
66           (0,0,1,0,0,0,0,0)
67           (0,0,0,1,0,0,0,0)
68           (0,0,0,0,0,1,0,0)
69           (0,0,0,0,1,0,0,0)
70           (0,0,0,0,0,0,0,1)
71           (0,0,0,0,0,0,1,0)];
72
73     outcome = deutsch(n, Ub);
74 }

```

Output

```

1 0
2 1
3 0

```

Grover's Search Algorithm

The following program implements special case of Grover's Search Algorithm for $f(0) = 1$.

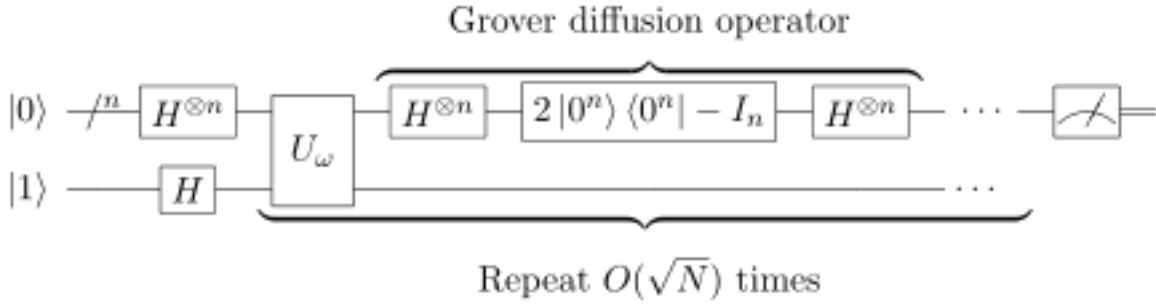


Figure 3.2: Grover Algorithm Circuit

```

1 def measure (mat top) : mat outcome{
3     mat ad;
5     ad = adj(top);
6     outcome = top * ad;
7 }
9 def ntensor (int n, mat k) : mat gate{
11     int i;
12     gate = k;
13
14     for (i from 0 to n-1 by 1){
15         gate = gate @ k;
16     }
17 }
19 def prepareU (int n) : mat gate {
20     mat i;
21     mat u;
22
23     i = [(1,0)
24          (0,0)];
25
26     u = ntensor(n+1, i);
27     gate = ntensor(n+1, IDT) - 2*u;
28 }
29
30 def prepareG (int n) : mat gate{
31     mat s; mat sa; mat i; mat h;
32
33     s = ntensor(n, |0>);
34     sa = adj(s);

```

```

35     i = ntensor(n, IDT);
36     gate = 2*s*sa - i;
37     h = ntensor(n, H);
38     gate = h*gate*h;
39     gate = gate @ IDT;
40 }
41
42 def grover (int n) : float outcomeZero{
43
44     mat bottom; mat top; mat input;
45     mat hadtop; mat u; mat g; mat go; mat meas;
46     int i;
47
48     bottom = |1>;
49     top = ntensor(n, |0>);
50     input = top @ bottom;
51
52     hadtop = ntensor(n, H);
53     input = (hadtop @ H)*input;
54     u = prepareU(n);
55     g = prepareG(n);
56
57     go = g*u;
58
59     for (i from 0 to n by 1){
60         input = go*input;
61     }
62
63     meas = measure(top);
64     input = (meas @ IDT)* input;
65     outcomeZero = norm(input);
66 }
67
68 def compute () : float outcome{
69     #simulate the grover for f(0)=1
70
71     int n; mat Ub; mat Uc;
72     n = 1;
73
74     outcome = grover(n);
75     print(outcome);
76
77     n = 2;
78     outcome = grover(n);
79 }

```

Output

```

0.707107
2 0.5

```

Chapter 4

Project Plan and Organization

The majority of our initial meetings consisted of creating a rough outline of how we envisioned our language. Much of the concept for the language was decided upon by Sankalpa, who was originally the one who suggested designing a quantum computing language. This strong foundation is what allowed us to create qlang.

4.1 Project Management

4.1.1 Planning

Throughout the semester we met regularly to keep everyone up to date on the overall progress of the project. Initially, it was twice a week after class for short meetings, but as the semester went on, we began to meet nearly everyday. At the end of every week, there was a short session reviewing what was accomplished that week, as well as our goals for the upcoming week.

4.1.2 Specification

Upon creation, the LRM was the manifestation of our vision. However, it was almost immediately upon submitting the LRM that we realized that there were some changes that had to be made. This was a common theme throughout the development process. Even though we had a set ideal of what we wanted, the specification of the implementation varied during the course of our work. However, constantly thinking about how certain things would affect, or be influenced by, the LRM caused us to think more critically about our code. Though our LRM changed during the project lifetime, QLang evolved as well.

4.1.3 Development

To ensure the group as a whole was able to coordinate their independent work, we used Git as a distributed version control system. Each team member worked on an individual feature. When they were satisfied that their section was working and had passed unit tests, it was pushed into the master branch. Once it was pushed, the other team members looked over the feature and made suggestions as well as pointed out any bugs that were missed. This iterative process was repeated the entire project.

4.1.4 Testing

We continuously performed unit tests throughout the development process. However, it was not until the end that we completed more rigorous acceptance testing. This was due to the continued evolution of our language as well as features. One constant throughout the project was a configurable test script that allowed us to complete the compilation process to a certain point. This allowed us to isolate tests for the individual parts of the compiler such as the AST or code generator.

4.2 Style Guide

The following coding guidelines were generally followed while coding:

- One statement per a line
- Each block of code following a “let” statement is indented
- Helper functions are written for commonly reused code

4.3 Project Timeline

Commits to master, excluding merge commits



The above graph shows the project timeline for the QLang compiler. It represents the number of commits over the course of the project, with a total of 397 commits. Work was generally centered around large project deadlines but slowing down near the end of the project as we were wrapping up.

4.4 Roles and Responsibilities

Christopher Campbell - System Architect (coded the greater part of the semantics)
Sankalpa Khadka - Language Guru (designed the majority of the features of our language)
Winnie Narang – Testing Verification and Validation (created the bulk of the test suite)
Jonathan Wong – Manager (built the QLang C++ library)
Cément Canonne - LaTeX

4.5 Software Development Environment

The QLang project was built on a combination of OS X and Arch Linux platforms. As stated above, Git was used as a distributed version control system. The compiler itself was written using both

vim and sublime. The project was done mostly in OCaml, but a QLang C++ library was created to augment the C++ Matrix library Eigen that is used for much of the linear algebra. Since our code was compiled to C++, g++ was used to compile the code into an executable. Lastly, Bash/shell scripts and makefiles were used to automate compilation and testing.

4.6 Project Log

Below is an excerpt from our git log in the format of “<YYYY-MM-DD>: <Author> - <Commit Message>”.

```
2014-12-17: khadka - main
2014-12-17: Christopher Campbell - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: Christopher Campbell - removed vestigial tokens
2014-12-17: khadka - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: khadka - main
2014-12-17: Winnie Narang - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: Winnie Narang - tex
2014-12-17: Christopher Campbell - lessons learned
2014-12-17: khadka - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: khadka - lessons learned
2014-12-17: Jonathan Wong - script to compile to execution file for single .ql file
2014-12-17: Jonathan Wong - cleaned up directory and minor change to Makefile
2014-12-17: Winnie Narang - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: Winnie Narang - main.tex
2014-12-17: khadka - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: khadka - demo2
2014-12-17: Winnie Narang - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: Winnie Narang - PPT
2014-12-17: khadka - grover
2014-12-17: Winnie Narang - merge
2014-12-17: khadka - grover
2014-12-17: khadka - lessons, examples
2014-12-17: khadka - revised tutorial and introduction
2014-12-17: khadka - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: khadka - demo 2 for probability
2014-12-17: khadka - demo 3 Deutsch
2014-12-17: Winnie Narang - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: Winnie Narang - tex files
2014-12-17: Christopher Campbell - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-17: Christopher Campbell - negative numbers in floats and funcitons with no params working
2014-12-17: khadka - demo 1 added
2014-12-16: Jonathan Wong - fixed up rows/cols
2014-12-16: Jonathan Wong - added cpp compilation to Makefile in Compiler directory
2014-12-16: Jonathan Wong - just kidding didn't get rid of them all
2014-12-16: Jonathan Wong - got rid of extraneous ; in generator print
2014-12-16: Jonathan Wong - added compile cpp to runTest script
2014-12-16: Jonathan Wong - cleaned directory
2014-12-16: Jonathan Wong - fixed vectorToBraket
2014-12-16: Jonathan Wong - Added double endl to print
2014-12-16: Winnie Narang - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-16: Christopher Campbell - fixed qlang.hpp
2014-12-16: Winnie Narang - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-16: Christopher Campbell - should be fixed
2014-12-16: Winnie Narang - Removed conflict
2014-12-16: Winnie Narang - Merge
2014-12-16: Christopher Campbell - Merge branch 'master' of https://github.com/thejonathanwong/PLT
2014-12-16: Christopher Campbell - added break and continue
2014-12-16: khadka - Merge branch 'master' of https://github.com/thejonathanwong/PLT
```


2014-12-16: khadka - introduciton
 2014-12-16: Winnie Narang - Formatted result in exec_output a little
 2014-12-16: Winnie Narang - Comp n float matrix binop tests
 2014-12-16: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-16: Christopher Campbell - more updates
 2014-12-16: Christopher Campbell - more updates to presentation
 2014-12-16: Christopher Campbell - more updates to presentation
 2014-12-16: Christopher Campbell - more updates2
 2014-12-16: Christopher Campbell - more updates
 2014-12-16: Christopher Campbell - working on powerpoint2
 2014-12-16: Christopher Campbell - working on powerpoint2
 2014-12-16: Christopher Campbell - working on powerpoint
 2014-12-16: Christopher Campbell - working on powerpoint
 2014-12-16: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-15: Jonathan Wong - duplicate Eigen, fixed cpp makefile
 2014-12-15: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - fixed tensor product
 2014-12-14: khadka - merge
 2014-12-14: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Winnie Narang - Better test for im,not and neg
 2014-12-14: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - for some reason Eigen was deleted. Fixed vectorToBraket to
 handle float coefficients
 2014-12-14: Winnie Narang - Fixup for assertion failed issue for determinant
 2014-12-14: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Winnie Narang - Fixed adjoint
 2014-12-14: Jonathan Wong - minor changes to norm test
 2014-12-14: Jonathan Wong - minor change
 2014-12-14: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Winnie Narang - Merge
 2014-12-14: Jonathan Wong - added constants test
 2014-12-14: Winnie Narang - All tests passing execution except for those with printq
 2014-12-14: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Christopher Campbell - added rows, cols, and elem builtin funcs
 2014-12-14: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - added some mat operators and qubit printing
 2014-12-14: Winnie Narang - Fixed else if keyword bug in generator.ml
 2014-12-14: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Christopher Campbell - updated test suite
 2014-12-14: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - accidentally removed if_stmt.ql
 2014-12-14: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Christopher Campbell - updated test suite
 2014-12-14: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - moved includes directory into Compiler dir, since that is
 the directory we are going to submit
 2014-12-14: Christopher Campbell - analyzer
 2014-12-14: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: khadka - chaning deutsch
 2014-12-14: Jonathan Wong - minor changes to merge
 2014-12-14: Christopher Campbell - updated test suite again
 2014-12-14: Christopher Campbell - updated test suite
 2014-12-14: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - fixed tests for norm and det to reflect them returning comp
 2014-12-14: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-14: Christopher Campbell - fixed det
 2014-12-14: Jonathan Wong - reorganized cpp directory and eigen lib. Added compilation to
 runTests.sh.
 2014-12-14: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-14: Jonathan Wong - some modifications to qlang.cpp

2014-12-13: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-13: Winnie Narang - Refined failures folder and added powerpoint
 2014-12-13: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-13: khadka - row and column
 2014-12-13: Clement Canonne - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-13: Clement Canonne - LRM, changes to get it consistent with the language.
 2014-12-13: Clement Canonne - LRM, changes to get it consistent with the language.
 2014-12-13: Christopher Campbell - really fixed it this time
 2014-12-13: Christopher Campbell - fixed comp comparisons
 2014-12-12: Christopher Campbell - script updates
 2014-12-12: Christopher Campbell - updated test script
 2014-12-12: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-12: Christopher Campbell - updated test script to take folder param
 2014-12-12: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-12: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-12: Christopher Campbell - syntax changes to analyzer
 2014-12-12: Jonathan Wong - changed const I to IDT
 2014-12-12: Winnie Narang - Refined failure test cases
 2014-12-12: Christopher Campbell - run tests update
 2014-12-12: Christopher Campbell - updated run tests
 2014-12-12: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-12: Winnie Narang - Failure test cases refined
 2014-12-12: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-12: Christopher Campbell - updated run tests script
 2014-12-12: Jonathan Wong - fix to if else
 2014-12-12: Jonathan Wong - Removed merge tokens from generator
 2014-12-12: Clement Canonne - Including the previous LRM, roughly (un)modified for now.
 2014-12-12: Clement Canonne - Tutorial: finished for now (i.e., not finished: DS algo and some others (?) still to add. Turning to the reference manual.
 2015-12-12: Christopher Campbell - fixed by x in for loop'
 2014-12-12: Clement Canonne - Added tests for while, for and if
 2014-12-12: Clement Canonne - Tutorial: if, loops, etc
 2014-12-12: Clement Canonne - Going through the tutorial: added basics
 2014-12-12: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-12: khadka - parts by parts
 2014-12-12: Clement Canonne - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-12: Clement Canonne - Fixing parsing errors.
 2014-12-12: Christopher Campbell - changed norm, det, and equality/inequality analysis
 2014-12-12: khadka - generator
 2014-12-11: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-11: Winnie Narang - Added some meaningful failures
 2014-12-11: Christopher Campbell - makefile for compiling our test output cpp
 2014-12-11: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-11: Christopher Campbell - fixed mod
 2014-12-11: Winnie Narang - generating outputs for qland programs complete
 2014-12-11: Winnie Narang - Got cpp code compilation working in general
 2014-12-11: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-11: Jonathan Wong - added multiple qubit functionality to qubitToString -> vectorToBracket
 2014-12-11: khadka - small change in function call
 2014-12-11: Winnie Narang - runTests.sh working
 2014-12-11: Clement Canonne - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-11: Winnie Narang - Resolving merge issues
 2014-12-11: Clement Canonne - Updated code for the tutotial, improved syntax for the code highlighting.
 2014-12-11: Clement Canonne - Started fixing deutsch.q1, not valid yet (parsing errors)
 2014-12-11: Winnie Narang - Updated runTests.sh
 2014-12-11: Clement Canonne - Adding stuff to the tutorial. TODO: check the Deutsch algo .q1 file in the tests, it seems to be buggy.
 2014-12-11: Christopher Campbell - small updates
 2014-12-11: Christopher Campbell - print working
 2014-12-10: Christopher Campbell - working

2014-12-10: Christopher Campbell - updated
 2014-12-10: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-10: Christopher Campbell - updated
 2014-12-10: Winnie Narang - Cleaning up temp cpp and ql files
 2014-12-10: Winnie Narang - Merge
 2014-12-10: Winnie Narang - Updated testing
 2014-12-10: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-10: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-10: Christopher Campbell - adding support for polymorphing print function
 2014-12-10: khadka - print stuff
 2014-12-10: Christopher Campbell - commit
 2014-12-10: Christopher Campbell - removed qub
 2014-12-10: khadka - print qubit
 2014-12-10: Jonathan Wong - enforced 1 dimensionality of qubitToString
 2014-12-10: khadka - test cases
 2014-12-10: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-10: khadka - generator to handle print and equals
 2014-12-10: Jonathan Wong - removed main from qlang.cpp
 2014-12-10: Jonathan Wong - moved eigen lib into directory
 2014-12-10: Jonathan Wong - added qubitToString to generate string representation ($|>$ & $<|$) of a qubit
 2014-12-10: Clement Canonne - Tutorial
 2014-12-10: Clement Canonne - Tutorial
 2014-12-10: Clement Canonne - iAdd tutorial file.
 2014-12-10: Clement Canonne - Add package for quantum circuits in Latex.
 2014-12-10: Clement Canonne - Starting to add final report, first attempt (wip)
 2014-12-10: Clement Canonne - Starting the first commit for the final report.
 2014-12-10: Jonathan Wong - Fixed qlang.cpp so test1.cpp and test2.cpp compiles
 2014-12-09: Christopher Campbell - trying to get test algorithms to work
 2014-12-06: khadka - algorithms to test for
 2014-12-06: Christopher Campbell - if else working
 2014-12-06: Christopher Campbell - implemented if else
 2014-12-06: Christopher Campbell - matrices and complex working
 2014-12-05: Christopher Campbell - fixed matrices
 2014-12-05: Christopher Campbell - blah
 2014-12-05: khadka - gen
 2014-12-05: khadka - function call defined
 2014-12-05: khadka - mat definition changed
 2014-12-05: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-05: khadka - print statement and qlc file output
 2014-12-05: Christopher Campbell - reading this?
 2014-12-05: khadka - additional test cases
 2014-12-05: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-05: Christopher Campbell - still no one reading this
 2014-12-05: Christopher Campbell - no one is reading this
 2014-12-05: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-05: Christopher Campbell - you suck more
 2014-12-05: Christopher Campbell - you suck
 2014-12-05: khadka - damn
 2014-12-05: khadka - qubit def
 2014-12-05: Christopher Campbell - fixed qubits
 2014-12-05: Christopher Campbell - fixed qubits
 2014-12-05: khadka - new qub
 2014-12-05: Christopher Campbell - updated
 2014-12-04: Jonathan Wong - changed all matrix gen to matrixXcf
 2014-12-03: Jonathan Wong - midfix of cpp qubit
 2014-12-03: Jonathan Wong - code gen qubit bra ket functionality
 2014-12-03: Jonathan Wong - attempt to add qubit func
 2014-12-03: Christopher Campbell - started final report document
 2014-12-03: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-03: Christopher Campbell - fixed qubit and return variable issue with analyzer

2014-12-03: khadka - working generator
 2014-12-03: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-03: khadka - merge conflict resolution
 2014-12-03: Winnie Narang - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-03: Winnie Narang - Semantic checks error output formatted
 2014-12-03: Jonathan Wong - Fixed semicolons and added initializer for return
 2014-12-03: Winnie Narang - Test Script and few cases
 2014-12-03: Christopher Campbell - remove test.sh
 2014-12-03: Christopher Campbell - fixed weird matrix output issue
 2014-12-03: khadka - test2
 2014-12-03: khadka - example test1
 2014-12-03: khadka - working qlc.ml
 2014-12-03: khadka - almost complete code generator; works
 2014-12-03: khadka - working qlc
 2014-12-03: khadka - working code generator with fixes
 2014-12-03: khadka - changes in test1.ql
 2014-12-03: khadka - working qlc with entire pipeline
 2014-12-03: khadka - Working makefile with all the requirements
 2014-12-02: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-02: Christopher Campbell - implemented matrix checking with the analyzer and printing
 with the ast and sast pretty printer
 2014-12-02: Winnie Narang - semantic testing; not working yet
 2014-12-02: Jonathan Wong - updated generator
 2014-12-02: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-02: Christopher Campbell - analyzer is working. also made changes to qubits across all
 files that may affect your work. please review them and we
 can talk about it
 2014-12-01: Jonathan Wong - removed extraneous headers
 2014-12-01: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-01: Jonathan Wong - Some minor fixes
 2014-12-01: khadka - vardecl
 2014-12-01: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-01: khadka - vardecl
 2014-12-01: Jonathan Wong - Added return variable initializer
 2014-12-01: khadka - vardecl
 2014-12-01: Winnie Narang - Merged
 2014-12-01: Winnie Narang - Call
 2014-12-01: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-12-01: Jonathan Wong - writeQubit and changes to header
 2014-12-01: Jonathan Wong - Consolidated qlang.h and constants.h into one file
 2014-12-01: khadka - merging
 2014-12-01: khadka - writeMatrix included
 2014-12-01: Winnie Narang - Lit_comp
 2014-12-01: Winnie Narang - cppExpr
 2014-12-01: Jonathan Wong - Finished writeUnop
 2014-12-01: Christopher Campbell - removing unnecessary comments now that we have a better
 understanding of how everything works
 2014-12-01: Christopher Campbell - analyzer compiles, but not complete and not tested
 2014-12-01: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-12-01: Christopher Campbell - big progress on analyzer, but still not compiling
 2014-11-30: khadka - more generator
 2014-11-30: Jonathan Wong - Added cpp directory with qubit gen
 2014-11-30: Winnie Narang - Working on Unop
 2014-11-30: Winnie Narang - Worked on cppExpr
 2014-11-30: Winnie Narang - Fixed cppStmt
 2014-11-30: Jonathan Wong - Merged conflicts, includes most of the controlflow
 2014-11-30: Jonathan Wong - Initial merge
 2014-11-30: Winnie Narang - Merged While and For
 2014-11-30: Winnie Narang - generator - added codegen skeleton for while
 2014-11-30: khadka - conflict solved
 2014-11-30: khadka - sast with updated statements

2014-11-30: khadka - changes with generator
 2014-11-30: khadka - qlc with generator
 2014-11-30: khadka - additional generator.ml
 2014-11-30: Jonathan Wong - start writeIfStmt in code gen
 2014-11-30: Jonathan Wong - Fixed minor typing mistakes
 2014-11-30: khadka - code generator starting point
 2014-11-30: Christopher Campbell - more updates...
 2014-11-29: Christopher Campbell - more updates to analyzer
 2014-11-29: Christopher Campbell - cleaned up analyzer
 2014-11-29: Christopher Campbell - sast is back
 2014-11-29: Christopher Campbell - updates to analyzer
 2014-11-26: Christopher Campbell - successfully able to parse full programs
 2014-11-26: Christopher Campbell - got statement lists working, program will be next
 2014-11-26: Christopher Campbell - getting further along with the testing
 2014-11-26: Christopher Campbell - compile script
 2014-11-26: Christopher Campbell - basic testing
 2014-11-26: Christopher Campbell - small changes
 2014-11-26: Christopher Campbell - merged
 2014-11-26: Christopher Campbell - updates to analyzer
 2014-11-23: Winnie Narang - Merge with exampleCPP
 2014-11-23: Winnie Narang - Pretty printing working...
 2014-11-23: Jonathan Wong - Made initial fixes to grover search
 2014-11-23: Jonathan Wong - Added control, updated problems, more examples
 2014-11-23: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-11-23: Jonathan Wong - Updated examples. Created constants and tensorProd
 2014-11-23: khadka - test1 program
 2014-11-23: Jonathan Wong - Fixed example 3
 2014-11-23: Jonathan Wong - Possible fix to example 3
 2014-11-23: Jonathan Wong - Made initial changes to LRM based on TA feedback
 2014-11-22: Christopher Campbell - small change
 2014-11-22: Christopher Campbell - added more to an analyzer, but I know it's not compiling
 right now so don't even try
 2014-11-20: Christopher Campbell - updated analyzer
 2014-11-20: Christopher Campbell - completed unop and binop checks for analyzer nad made
 small changes to the other files
 2014-11-19: Christopher Campbell - still working on analyzer
 2014-11-19: Christopher Campbell - merging
 2014-11-19: Christopher Campbell - working analyzer - far from done
 2014-11-19: khadka - a first working sast
 2014-11-19: khadka - included sast.mli in make
 2014-11-19: Jonathan Wong - Added C++ code for examples in LRM. Prob 3 broken
 2014-11-17: Winnie Narang - Pretty printer for AST added, not complete yet
 2014-11-16: Christopher Campbell - added files for sast, analyzer, and compiler
 2014-11-16: Christopher Campbell - fixed shift/reduce conflicts for our 'for' statements
 and complex numbers ; '
 2014-11-12: Christopher Campbell - added project examples
 2014-11-10: Christopher Campbell - finished ast, parse, and scanner for the most part, but
 need to be carefully reviewed and tested
 2014-11-09: Christopher Campbell - updated ast, parser, and scanner
 2014-11-09: Christopher Campbell - updated ast and parser
 2014-11-09: Christopher Campbell - updated ast and parser
 2014-11-09: Christopher Campbell - updated scanner
 2014-11-09: Christopher Campbell - updated ast and parser
 2014-11-04: Christopher Campbell - Updated Ast and scanner
 2014-11-04: khadka - new tokens added
 2014-11-04: khadka - fix merge conflicts
 2014-11-04: khadka - updated with all the token from the scanner
 2014-11-04: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-11-04: Christopher Campbell - Finished AST, although it will almost certainly need
 revision
 2014-11-04: Jonathan Wong - Added some tokens to parser

2014-10-27: khadka - build main.pdf
 2014-10-27: Jonathan Wong - turned off colour and notes in main.tex
 2014-10-27: Jonathan Wong - redo scope commit
 2014-10-27: khadka - examples
 2014-10-27: Winnie Narang - Refined Statements and Scope
 2014-10-27: Winnie Narang - Added statements and scope rules
 2014-10-27: Jonathan Wong - Fixed rolled back changes in sec-declarations
 2014-10-27: Clement Canonne - Added matrix and array access [i] and [i,j]
 2014-10-27: Clement Canonne - Changes (fixed inconsistencies and types; added valid types and conversions).
 2014-10-27: Clement Canonne - Added files for new sections.
 2014-10-27: Clement Canonne - Second round of change: fixed some inconsistencies.
 2014-10-27: Clement Canonne - First round of changes: (lexical conventions updated soon).
 2014-10-26: khadka - main
 2014-10-26: khadka - grover circuit
 2014-10-26: khadka - new examples
 2014-10-26: Christopher Campbell - Finished sec-expressions.text
 2014-10-26: Jonathan Wong - Merge <https://github.com/thejonathanwong/PLT>
 2014-10-26: khadka - main
 2014-10-26: Jonathan Wong - Fixed sec-declarations.tex
 2014-10-26: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-10-26: khadka - main
 2014-10-26: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-10-26: Christopher Campbell - Updating sec-expressions.tex
 2014-10-26: khadka - troubleshooting
 2014-10-26: khadka - updated examples
 2014-10-26: khadka - section on constant expressions
 2014-10-26: khadka - main with updated sections
 2014-10-26: Christopher Campbell - Added sec-expressions-table.tex
 2014-10-26: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-10-26: Christopher Campbell - Fixed formatting and compile issues with sec-declarations.tex
 2014-10-26: khadka - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-10-26: khadka - section for examples
 2014-10-26: khadka - just the main
 2014-10-26: thejonathanwong - Added declarations section
 2014-10-26: khadka - new section for constant expressions
 2014-10-26: khadka - new package for graphics
 2014-10-26: khadka - new section examples
 2014-10-26: khadka - images
 2014-10-26: khadka - updated packages with listling for code formatting
 2014-10-26: khadka - example section
 2014-10-24: Christopher Campbell - Completed a large part of 'expressions' section and small changes other places
 2014-10-20: Christopher Campbell - Made small changes to the scanner and added package and preamble files
 2014-10-15: Clement Canonne - Filled section 2.
 2014-10-15: Clement Canonne - Filled lexical conventions, added packages and preamble files.
 2014-10-15: Clement Canonne - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-10-15: Clement Canonne - (latest changes)
 2014-10-14: Christopher Campbell - Completed most of the scanner
 2014-10-13: Christopher Campbell - added . to symbols
 2014-10-13: Christopher Campbell - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-10-13: Christopher Campbell - Started scanner
 2014-10-13: Clement Canonne - It goes on: started filling the reference manual, added a file for the keywords.
 2014-10-13: Clement Canonne - Merge branch 'master' of <https://github.com/thejonathanwong/PLT>
 2014-10-13: Clement Canonne - First push: baby reference manual, take #1.
 2014-10-13: Christopher Campbell - Added folder and documents for our compiler
 2014-10-13: Christopher Campbell - Adding microc to resources
 2014-10-13: khadka - starter for language referece manual
 2014-10-13: Sankalpa Khadka - starter ast

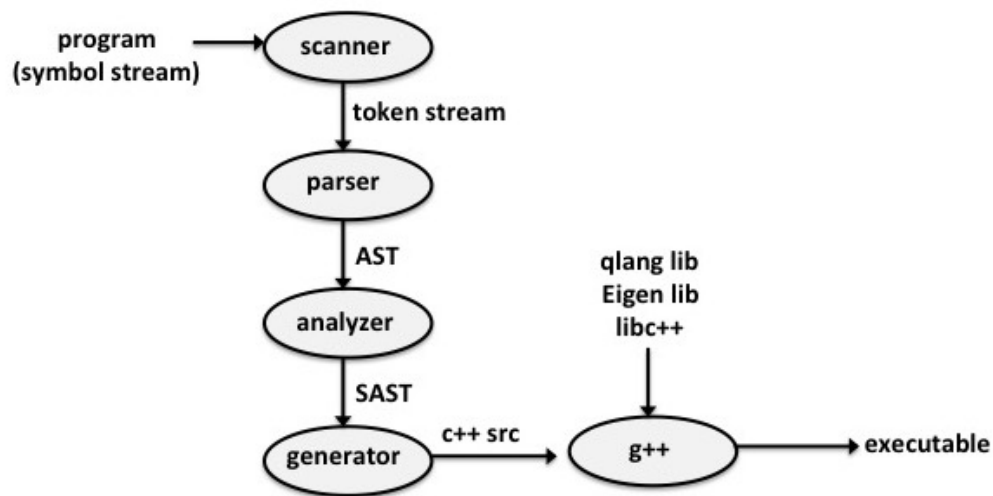
2014-09-30: Christopher Campbell - Adding a Resources folder. It already contains two quantum computing resources that are pretty helpful.

2014-09-29: thejonathanwong - Initial commit

Chapter 5

Architectural Design

5.0.1 Block Diagram



5.0.2 Components

1. Scanner

The scanner was implemented using ocamellex - the associated file is scanner.mll. It was chiefly implemented by Christopher Campbell and Winnie Narang.

The scanner takes a program (symbol stream) as input and tokenizes it to produce a token stream. The tokenization process provides basic syntax checking, rejecting programs that contain illegal symbols and illegal combinations of symbols (e.g. the \$ symbol). Additionally, it discards information that is unnecessary for the remainder of the compilation process such as white space and comments.

2. Parser & Abstract Syntax Tree

The parser was implemented using `ocamlyacc` - the associated files are `ast.ml` and `parser.mly`. It was chiefly implemented by Christopher Campbell and Sankalpa Khadka.

The parser takes the token stream produced by the scanner as input and parses it to produce an abstract syntax tree (AST), which describes the overall structure of the program. `ast.ml` provides `parser.mly` with the acceptable structure of the AST. The parsing process provides further syntax checking, rejecting programs that do not strictly meet the syntactic requirements of the AST (e.g. a malformed for statement).

3. Analyzer & Semantically Analyzed Syntax Tree

The analyzer was implemented in OCaml - the associated files are `analyzer.ml` and `sast.ml`. Additionally, `analyzer.ml` utilizes `ast.ml` in order to be able to analyze its input. It was chiefly implemented by Christopher Campbell.

The analyzer takes the `ast` produced by the parser and analyzes it to produce a semantically analyzed abstract syntax tree (SAST). Like the AST, the SAST describes the overall structure of the program, but it also includes type information that was attached during the analysis process. `sast.ml` provides `analyzer.ml` with the acceptable structure of the SAST. The analysis process provides rigorous semantic checking, rejecting programs that violate type requirements (e.g. assigning a complex number to a variable declared as an integer), declaration requirements (e.g. using a variable that was not declared or attempting to declare a variable more than once), scope requirements (e.g. using a variable declared in another function), order requirements (e.g. calling a function before it is declared), and other language-specific requirements (e.g. not declaring a compute function). Additionally, the analyzer adds built-in information (i.e. built-in variables and functions) to the `sast`.

4. Generator

The generator was implemented in OCaml - the associated file is `generator.ml`. Additionally, `generator.ml` utilizes `sast.ml` in order to be able to process its input. It was chiefly implemented by Sankalpa Khadka, Jonathan Wong, and Winnie Narang.

The generator takes the `sast` produced by the analyzer and generates `c++` code from it. Most of the code it generates is hard coded into `generator.ml`, but it also draws on code from our standard library - `qlanglib`, `libc++`, and `Eigen` (a third-party library).

5. QLang Library

The QLang Library was implemented in `c++` - the associated files are `qlang.hpp` and `qlang.cpp`. It was chiefly implemented by Jonathan Wong. The QLang library contains `c++` code for carrying out some of the more complex conversions from `qlang` code to `c++` code in the generator (e.g. generating qubits and carrying out the tensor product).

Chapter 6

Test Plan

6.1 Testing Phases

6.1.1 Unit Testing

Unit testing was done at very point essentially, as we were in the coding phase. Every building block was tested rigorously using multiple cases. We tested for recognition of datatypes, variables , expression statements and functions initially, and then moved on to AST generation.

6.1.2 Integration Testing

In this phase,the various modules were put together and tested incrementally again. So once the AST could be generated, we moved on to test the semantic analysis and code generation.

6.1.3 System Testing

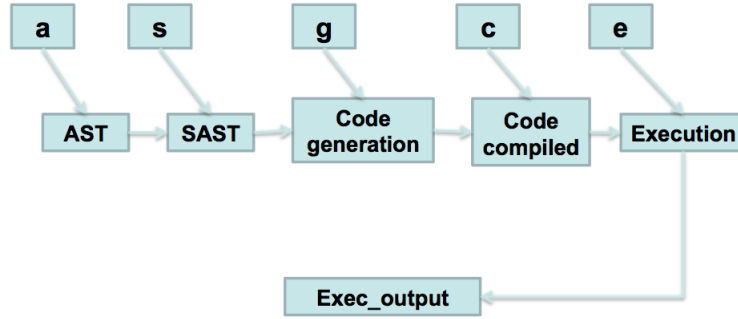
System testing entailed end to end testing of our entire language framework. The input program written in QLang is fed to the compiler and it gives out the final output of the program, having passed through the parsing, scanning, compiling, code generation and execution phases. The final results were piped to an output file where we could see all the outputs.

6.2 Automation and Implementation

A shell script was written in order to automate the test cases at each level, syntax, semantic, code generation and accurate execution. Our file is called runTests.sh, located in the 'test' folder. It takes a folder having QLang program files, and the operation to be done on them as arguments. The outputs of the respective operation can be seen in the corresponding output file.

The operation options available are :

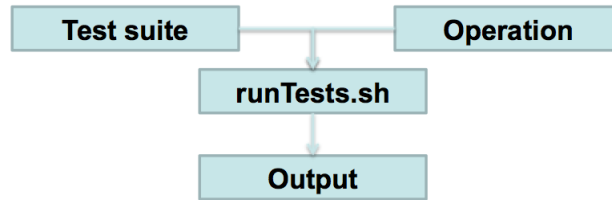
- a : Parsing, scanning and AST generation.
- s : SAST generation.
- g : Code generation.
- c : Generated code is compiled.
- e : Generated executable is run, to generate the program's outputs.



The operations mentioned above are each inclusive of the operations mentioned above them. That means, if you enter the 'g' option, runTests.sh will perform the tasks under 'a','s' and then the operations specific to 'g' as well.

The second argument is the folder that has the input program files. We have acronyms for two folder that are standard to our implementation, the SemanticSuccess and the SemanticFailures. So to run the sast generation on the files in SemanticSuccess folder, we would write :

```
sh runTests.sh s ss.
```



The entire code of this script can be seen in the appendix. The Test Suites were chiefly created by Winnie Narang, and everyone else also contributed test cases. The script runTests.sh was created by Winnie Narang and Christopher Campbell.

6.3 Sample test programs

The effort has been to exhaustively test every kind of execution scenario, in what can be a typical user program. We have created many test files to showcase varied kinds of programs that can be written in QLang, as can be seen in the contents of the SemanticSuccess and SemanticFailures folders.

The rationale is to make sure that syntactically or semantically incorrect programs are not compiled and echo corresponding meaningful error messages to the user, and that correct programs are accepted and executed correctly.

Hence, we have separate test programs to test all kinds of unary and binary operations on all datatypes that our language supports, and also for all kinds of statements and possible combinations of expressions. Though the test suite is too large to be included in this section, here are a few sample success and failure cases that showcase different applications of our language :

For instance, break_continue.q1 is a QLang program as follows :

```

1 def func_test(int a) : int ret_name {
3
5     int i;
7     for(i from 0 to 2 by 1)
8         a=a+5;
9     for(i from 2 to 0 by -1)
10    {
11        a=a*10;
12        print(a);
13        break;
14    }
15    for(i from 1 to 5)
16    {
17        print(a);
18        continue;
19        a=a*10;
20    }
21
22    ret_name = a;
23 }
24
25 def compute(): int trial {
26
27     trial = func_test(20);
28 }

```

It generates break_continue.cpp as below upon passing it through the code generation code

```

2 #include <iostream>
3 #include <complex>
4 #include <cmath>
5 #include <Eigen/Dense>
6 #include <qlang>
7 using namespace Eigen;
8 using namespace std;
9
10 int func_test (int a )
11 {
12     int i;
13     int ret_name;
14
15     for (int i = 0; i < 2; i = i + 1){
16         a = a + 5;
17     }
18     for (int i = 2; i < 0; i = i + -1){
19     {
20         a = a * 10;
21
22         cout << a << endl;
23
24         break;
25     }
26 }

```

```

30     }
31     for (int i = 1; i < 5; i = i + 1){
32     {
33     cout << a << endl;
34     continue;
35     a = a * 10;
36     }
37     } ret_name = a;
38     return ret_name;
39 }
40 int main ()
41 {
42     int trial;
43     trial = func_test(20);
44     std::cout << trial << endl;
45     return 0;
46 }

```

and the generated output of this is :

```

30
2 30
30
4 30
30

```

Another example we consider is mat_qubit.q1

```

1 def func_test(mat a, mat b) : mat ret_name {
2
3     ret_name = a*b;
4
5 }
6
7 def compute(int a):mat trial {
8
9     mat zero;
10    mat one;
11
12    zero = |0>;
13    one  = |1>;
14
15    trial = func_test(H, zero);
16    printq(trial);
17
18    trial = func_test(H, one);
19    printq(trial);
20
21 }

```

It generates mat_qubit.cpp as below :

```
1 #include <iostream>
2 #include <complex>
3 #include <cmath>
4 #include <Eigen/Dense>
5 #include <qlang>
6 using namespace Eigen;
7 using namespace std;
8
9 MatrixXcf func_test (MatrixXcf a,MatrixXcf b )
10 {
11     MatrixXcf ret_name;
12
13     ret_name = a * b;
14
15     return ret_name;
16 }
17 int main ()
18 {
19     MatrixXcf zero;
20     MatrixXcf one;
21     MatrixXcf trial;
22
23     zero = genQubit("0",0);
24     one = genQubit("1",0);
25     trial = func_test(H,zero);
26     cout << vectorToBraket(trial) << endl;
27     trial = func_test(H,one);
28     cout << vectorToBraket(trial) << endl;
29
30     std::cout << trial << endl;
31
32     return 0;
33 }
```

and it generates the qubits in the output as well, like :

```
1 (0.707107)|0> + (0.707107)|1>
2 (0.707107)|0> + (-0.707107)|1>
3 (0.707107,0)
4 (-0.707107,0)
```

One more program we can show here is a demonstration of the capacity of QLang to emulate Quantum algorithms. The following program runs the Deutsch-Jozsa algorithm.

```
1 def measure (mat top) : mat outcome{
2
3     mat ad;
4
5     ad = adj(top);
6     outcome = top * ad;
7 }
8
9 def hadamard (int n) : mat gate{
10
11     int i;
12     gate = H;
13
14     for (i from 0 to n-1 by 1){
```

```

16         gate = gate @ H;
17     }
18 }
19
20 def topqubit (int n) : mat input{
21
22     int i;
23     input = |0>;
24
25     for (i from 0 to n-1 by 1){
26         input = input @ |0>;
27     }
28 }
29
30 def deutsch (int n, mat U) : float outcomeZero{
31
32     mat bottom; mat top; mat input;
33     mat hadtop; mat meas;
34
35     bottom = |1>;
36     top = topqubit(n);
37     input = top @ bottom;
38
39     hadtop = hadamard(n);
40     input = (hadtop @ H)*input;
41     input = U * input;
42     input = (hadtop @ IDT)*input;
43     meas = measure(top);
44
45     input = (meas @ IDT)* input;
46     outcomeZero = norm(input);
47 }
48
49 def compute () : float outcome{
50
51     int n; mat Ub; mat Uc;
52
53     n = 1;
54     Ub = [(1,0,0,0) (0,1,0,0) (0,0,0,1) (0,0,1,0)];
55     Uc = [(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)];
56
57     outcome = deutsch(n, Ub);
58     print(outcome);
59
60     outcome = deutsch(n, Uc);
61     print(outcome);
62
63     n = 2;
64     Ub = [(1,0,0,0,0,0,0,0)
65           (0,1,0,0,0,0,0,0)
66           (0,0,1,0,0,0,0,0)
67           (0,0,0,1,0,0,0,0)
68           (0,0,0,0,0,1,0,0)
69           (0,0,0,0,1,0,0,0)
70           (0,0,0,0,0,0,0,1)
71           (0,0,0,0,0,0,1,0)];
72
73     outcome = deutsch(n, Ub);
74 }

```

It creates the C++ code as follows :

```

2 #include <iostream>
  #include <complex>
4 #include <cmath>
  #include <Eigen/Dense>
6 #include <qlang>
  using namespace Eigen;
8 using namespace std;

10 MatrixXcf measure (MatrixXcf top )
  {
12   MatrixXcf ad;
    MatrixXcf outcome;

14   ad = top.adjoint();
16   outcome = top * ad;

18   return outcome;
  }
20 MatrixXcf hadamard (int n )
  {
22   int i;
    MatrixXcf gate;

24   gate = H;

26   for (int i = 0; i < n - 1; i = i + 1){
28     {
30     gate = tensor(gate, H);
32     }
34   }
    return gate;
36 }
MatrixXcf topqubit (int n )
38 {
40   int i;
    MatrixXcf input;

42   input = genQubit("0",0);

44   for (int i = 0; i < n - 1; i = i + 1){
46     {
48     input = tensor(input, genQubit("0",0));
50     }
52   }
    return input;
  }
54 float deutsch (int n, MatrixXcf U )
  {
56   MatrixXcf bottom;
    MatrixXcf top;
58   MatrixXcf input;
    MatrixXcf hadtop;
60   MatrixXcf meas;
    float outcomeZero;

62   bottom = genQubit("1",0);
64   top = topqubit(n);
    input = tensor(top, bottom);

```



```
13 }  
14 def compute( int a):int trial {  
15     trial = func_test1(4);  
16 }  
17 }
```

gives the error :

```
Fatal error: exception Analyzer.Except("Invalid function declaration: func_test1 was already  
declared")
```

whereas the sample program

```
1 def func_test(float z) : float ret_name {  
2     float a;  
3     a = 5.8;  
4     ret_name = z;  
5 }  
6 }
```

would give the error :

```
1 Fatal error: exception Analyzer.Except("Missing 'compute' function")
```

More such pass and fail test cases can be found in the appendix and in our project folder.

Chapter 7

Lesson Learned

7.1 Christopher Champbell

I learned many lessons from this project, most of which were related to group dynamics. I learned that, depending on how they are managed and leveraged, every group member's differences (i.e. differences in ideas, opinions, abilities, etc.) can either be beneficial or detrimental to the group and the project. In order to effectively leverage differences in opinion, all group member's opinions should be heard and considered by the group, and if a clear winner does not emerge the leader for that part of the project should make a decisive decision. This situation highlights another aspect of group dynamics that I learned - leaders are important. In order to keep the different parts of the project focused and progressing, each part should have a group member that leads its development. Leading the development of a part of the project entails having expertise in the associated domain, resolving tough issues and questions with it, and driving its development from beginning to end. In addition to the lessons I learned involving group dynamics, I also learned lessons, and re-learned lessons that I should have already known, that apply to project work in general. Among these lessons learned were: start early and manage your time well, thoroughly research ideas before you begin implementing them, and maintain a big picture view of the project.

7.2 Sankalpa Khadka

I realized that one of the important aspects of doing a big project is to make incremental progress, however small, over time. In the beginning, it is not always possible to have a global view of how each component of project fits in together. This can be a discouraging factor at times, however this should not deter anyone from building the components of the project. Teamwork is very crucial to the success of the project. From the very beginning of the project, it is important to delegate responsibilities and making sure that each member of team is contributing to the project. Any disruption to this can affect the work balance.

Finally, it is a very fulfilling experience to design a programming language from CS perspective. This experience draws from both theory and application aspect of CS. Everyone doing similar projects in future should try to participate, contribute and enjoy the process.

7.3 Winnie Narang

I learned that one should always work while keeping in the mind the shape of the end result. That helps in making sure your efforts are not wasted, and helps you make decisions more easily. Also, start early. And always test every change as you go. If you code everything at once and then it doesn't work, it gets very hard to debug.

Also, since we were using git as our version control system, we had to deal with numerous merge conflicts. So I learnt that one should keep committing changes, as you code as soon as you can be sure however much you have written is correct, no matter how small the change. this helps makes sure you are not causing any faults or conflicts for the other team members and also for you as an individual.

7.4 Jonathan Wong

At the start of the semester, the task of creating a new programming language seemed to be an impossibility. However, over the course of the semester, I learned and a gained an appreciation for the amount of work and though that goes into the creation of languages.

I have learned that communication is key. Without our regular meetings, I would have been floundering whenever I was off working independently. It is necessary to have a clear picture of what is needed to be done everytime you sit down to work on the project. Effective communication also allowed us to flesh out the gritty details of the language. What could have been improved is how quickly a consensus on specifics could be reached. It would have been better to have one person decide on these things and work could have been started immediatedly.

I know this is probably a common sentiment, but starting this project early is key. We did not really start pushing until the start of November. Ideally, once we covered the lectures on the structure of the compiler we should have hit the ground running. I have also learned that even though git can be great at what it does, improper use of it can lead to a great deal of frustration if one pushes bad code, or someone else deletes good working code.

Appendix A

More on Quantum Computing

A.1 Common quantum gates

Pauli Operators

The *Pauli operators* are the special single qubit gates which are represented by the Pauli matrices $\{I, X, Y, Z\}$ as follows

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

For example, the application of X causes bit-flip in following ways:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle.$$

Hadamard Gate

The *Hadamard gate* is defined by the matrix:

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

The Hadamard gate maps the computational basis states into superposition of states. The Hadamard gate is significant since it produces maximally entangled states from basis states in the following ways:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

Controlled-U Gates

A *controlled-U gate* is the quantum gate in which the U operator acts on the n^{th} n -qubit only if the value of the preceding qubit is 1.

For example: In a Controlled-NOT gate, the NOT operator flips the second qubit if the first qubit is 1.

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\text{CNOT}|00\rangle = |00\rangle$$

$$\text{CNOT}|01\rangle = |01\rangle$$

$$\text{CNOT}|10\rangle = |11\rangle$$

$$\text{CNOT}|11\rangle = |10\rangle.$$

A.2 Tensor product and its properties

Let $A = (a_{i,j})$ be a matrix with respect to the ordered basis $\mathcal{A} = (u_1, \dots, u_n)$ and $B = (b_{i,j})$ be a matrix with respect to the ordered basis $\mathcal{B} = (v_1, \dots, v_m)$. Consider the ordered basis $\mathcal{C} = (u_i \otimes v_j)$ ordered by lexicographic order, that is $u_i \otimes v_j \leq u_l \otimes v_k$ if if $i < l$ or $i = l$ and $j < k$. The matrix of $A \otimes B$ with respect to \mathcal{C} is :

$$A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \dots & a_{1,n}B \\ a_{2,1}B & a_{2,2}B & \dots & a_{2,n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}B & a_{n,2}B & \dots & a_{n,n}B \end{bmatrix}$$

This matrix is called the tensor product of the matrix A with the matrix B .

- $A \otimes B \otimes C = (A \otimes B) \otimes C = A \otimes (B \otimes C)$
- $a(|x\rangle \otimes |y\rangle) = a|x\rangle \otimes |y\rangle = |x\rangle \otimes a|y\rangle$
- $(A \otimes B) \cdot (|y\rangle|z\rangle) = A|y\rangle \otimes B|z\rangle$
- $(A \otimes B) \cdot (C \otimes D) = AC \otimes BD$
- $(A \otimes B)^H = A^H \otimes B^H$
- If A and B unitary, $A \otimes B$ is unitary.
- If $|x\rangle = |x_1\rangle|x_2\rangle$ and $|y\rangle = |y_1\rangle|y_2\rangle$ then $\langle x|y\rangle = \langle x_1|y_1\rangle\langle x_2|y_2\rangle$

Appendix B

Source Code

B.1 Scanner

scanner.mll

```
1 (* Christopher Campbell, Winnie Narang*)
   { open Parser }
3
   let whitespace = [' ' '\t' '\r' '\n']
5 let name = ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_' ]*
   let integers = ['0'-'9']+
7 let floats = ['0'-'9']+ '.' ['0'-'9']*
9
   rule token = parse
       whitespace { token lexbuf }
11 | '#'         { comment lexbuf }
13 | "int"        { INT }      (* Integer type *)
14 | "float"      { FLOAT }    (* Float type *)
15 | "comp"       { COMP }     (* Complex type *)
16 | "mat"        { MAT }      (* Matrix *)
17
18 | "C"          { C }        (* Start of complex number *)
19 | "I"          { I }        (* Imaginary component *)
20
21 | "def"        { DEF }      (* Define function *)
22
23 | '='          { ASSIGN }    (* Assignment *)
24 | ','          { COMMA }     (* Separate list elements *)
25 | ':'          { COLON }     (* Separate matrix rows *)
26 | ';'          { SEMI }      (* Separate matrix columns *)
27 | '('          { LPAREN }    (* Surround expression *)
28 | ')'          { RPAREN }
29 | '['          { LBRACK }     (* Surround vectors/matricies *)
30 | ']'          { RBRACK }
31 | '{'          { LBRACE }     (* Surround blocks *)
32 | '}'          { RBRACE }
33 | '<'          { LCAR }      (* Open bra- *)
34 | '>'          { RCAR }      (* Close -ket *)
35 | '|'          { BAR }       (* Close bra- and Open -ket *)
36
37 | '+'          { PLUS }      (* Addition *)
38 | '-'          { MINUS }     (* Subtraction *)
39 | '*'          { TIMES }     (* Multiplication *)
40 | '/'          { DIV }       (* Division *)
41 | '%'          { MOD }       (* Modulus *)
```

```

43 | ' ^ ,      { EXPN }      (* Exponentiation *)
|
| "eq"       { EQ }       (* Equal to (structural) *)
45 | "neq"      { NEQ }      (* Not equal to (structural) *)
| "lt"       { LT }       (* Less than *)
47 | "gt"       { GT }       (* Greater than *)
| "leq"      { LEQ }      (* Less than or equal to *)
49 | "geq"      { GEQ }      (* Greater than or equal to *)
|
51 | "not"      { NOT }      (* Boolean not *)
| "and"      { AND }      (* Boolean and *)
53 | "or"       { OR }       (* Boolean or *)
| "xor"      { XOR }      (* Boolean xor *)
55 |
| "norm"     { NORM }     (* Get norm *)
57 | "trans"    { TRANS }   (* Get transpose *)
| "det"      { DET }      (* Get determinant *)
59 | "adj"      { ADJ }      (* Get adjoint *)
| "conj"     { CONJ }     (* Get complex conjugate *)
61 | "unit"     { UNIT }     (* Is unit matrix? *)
| '@'        { TENS }     (* Tensor product *)
63 | "im"       { IM }       (* Is imaginary number? *)
| "re"       { RE }       (* Is real number *)
65 | "sin"      { SIN }      (* Sine *)
| "cos"      { COS }      (* Cosine *)
67 | "tan"      { TAN }      (* Tangent *)
|
69 | "if"       { IF }       (* If statement *)
| "else"      { ELSE }     (* Else statement *)
71 | "for"      { FOR }      (* For loop - for(i from x to y by z) *)
| "from"      { FROM }
73 | "to"       { TO }
| "by"       { BY }
75 | "while"    { WHILE }    (* While loop *)
| "break"     { BREAK }    (* Break For or While loop *)
77 | "continue" { CONT }     (* Continue to For or While loop *)
|
79 | name      as lxm { ID(lxm) }
| integers as lxm { INT_LIT(lxm) }
81 | floats    as lxm { FLOAT_LIT(float_of_string lxm) }
|
83 | eof        { EOF }
| _ as char    { raise (Failure("illegal character " ^ Char.escaped char)) }
85 |
and comment = parse
87 | ['\r' '\n'] { token lexbuf }
| _

```

B.2 Parser

parser.mly

```

(* Christopher Campbell, Sankalpa Khadka *)
2 %{ open Ast %}
|
4 %token C I
%token INT FLOAT COMP MAT
6 %token DEF
%token ASSIGN
8 %token COMMA COLON SEMI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE LCAR RCAR BAR

```



```

%token PLUS MINUS TIMES DIV MOD EXPN
10 %token EQ NEQ LT GT LEQ GEQ
%token NOT AND OR XOR
12 %token TENS UNIT NORM TRANS DET ADJ CONJ IM RE SIN COS TAN
%token IF ELIF ELSE FOR FROM TO BY WHILE BREAK CONT
14 %token EOF

16 %token <string> ID
%token <string> INT_LIT
18 %token <float> FLOAT_LIT
%token <string> COMP_LIT
20
%nonassoc NOELSE
22 %nonassoc ELSE
%right ASSIGN
24 %left OR XOR
%left AND
26 %right NOT
%left EQ NEQ
28 %left LT GT LEQ GEQ
%left PLUS MINUS
30 %left TIMES DIV MOD TENS
%right EXPN
32 %nonassoc RE IM NORM TRANS DET ADJ CONJ UNIT SIN COS TAN

34 %start program
%type <Ast.program> program
36
%%

38
vtype:
40   INT      { Int  }
   | FLOAT   { Float }
42   | COMP    { Comp }
   | MAT      { Mat  }
44
vdecl:
46   vtype ID SEMI { { typ = $1;
                      name = $2 } }
48
vdecl_list:
   /* nothing */ { [] }
50 | vdecl_list vdecl { $2 :: $1 }

52
formal_params:
   /* nothing */ { [] }
54 | formal_params_list { List.rev $1 }

56
formal_params_list:
   vtype ID { [{ typ = $1;
58               name = $2; }] }
   | formal_params_list COMMA vtype ID { { typ = $3;
60               name = $4; } :: $1 }

actual_params:
62   /* nothing */ { [] }
   | actual_params_list { List.rev $1 }
64
actual_params_list:
66   expr { [$1] }
   | actual_params_list COMMA expr { $3 :: $1 }
68
fdecl:
70   DEF ID LPAREN formal_params RPAREN COLON vtype ID LBRACE vdecl_list stmt_list RBRACE
   { { func_name = $2;
72       formal_params = $4;
       ret_typ = $7;

```

```

74         ret_name = $8;
75         locals = List.rev $10;
76         body = List.rev $11; } }

78 mat_row:
79     expr { [$1] }
80 | mat_row COMMA expr { $3 :: $1 }

82 mat_row_list:
83     LPAREN mat_row RPAREN { [List.rev($2)] }
84 | mat_row_list LPAREN mat_row RPAREN { List.rev($3) :: $1 }

86 inner_comp:
87     FLOAT_LIT { [$1; 0.] }
88 | FLOAT_LIT I { [0.; $1] }
89 | FLOAT_LIT PLUS FLOAT_LIT I { [$1; $3] }

90 expr:
91     ID { Id($1) }
92 | INT_LIT { Lit_int(int_of_string $1) }
93 | FLOAT_LIT { Lit_float($1) }
94 | C LPAREN inner_comp RPAREN { Lit_comp(List.hd $3, List.hd (List.rev $3)) }
95 | LCAR INT_LIT BAR { Lit_qub($2, 0) }
96 | BAR INT_LIT RCAR { Lit_qub($2, 1) }
97 | LBRACK mat_row_list RBRACK { Mat(List.rev($2)) }
98 | LPAREN expr RPAREN { $2 }
99 | ID ASSIGN expr { Assign($1, $3) }
100 | ID LPAREN actual_params RPAREN { Call($1, $3) }
101 | MINUS expr { Unop(Neg, $2) }
102 | NOT LPAREN expr RPAREN { Unop(Not, $3) }
103 | RE LPAREN expr RPAREN { Unop(Re, $3) }
104 | IM LPAREN expr RPAREN { Unop(Im, $3) }
105 | NORM LPAREN expr RPAREN { Unop(Norm, $3) }
106 | TRANS LPAREN expr RPAREN { Unop(Trans, $3) }
107 | DET LPAREN expr RPAREN { Unop(Det, $3) }
108 | ADJ LPAREN expr RPAREN { Unop(Adj, $3) }
109 | CONJ LPAREN expr RPAREN { Unop(Conj, $3) }
110 | UNIT LPAREN expr RPAREN { Unop(Unit, $3) }
111 | SIN LPAREN expr RPAREN { Unop(Sin, $3) }
112 | COS LPAREN expr RPAREN { Unop(Cos, $3) }
113 | TAN LPAREN expr RPAREN { Unop(Tan, $3) }
114 | expr PLUS expr { Binop($1, Add, $3) }
115 | expr MINUS expr { Binop($1, Sub, $3) }
116 | expr TIMES expr { Binop($1, Mult, $3) }
117 | expr DIV expr { Binop($1, Div, $3) }
118 | expr MOD expr { Binop($1, Mod, $3) }
119 | expr EXPN expr { Binop($1, Expn, $3) }
120 | expr TENS expr { Binop($1, Tens, $3) }
121 | expr EQ expr { Binop($1, Eq, $3) }
122 | expr NEQ expr { Binop($1, Neq, $3) }
123 | expr LT expr { Binop($1, Lt, $3) }
124 | expr GT expr { Binop($1, Gt, $3) }
125 | expr LEQ expr { Binop($1, Leq, $3) }
126 | expr GEQ expr { Binop($1, Geq, $3) }
127 | expr OR expr { Binop($1, Or, $3) }
128 | expr AND expr { Binop($1, And, $3) }
129 | expr XOR expr { Binop($1, Xor, $3) }

132 by:
133     /* nothing */ { Noexpr }
134 | BY expr { $2 }

136 stmt:
137     expr SEMI { Expr($1) }
138 | LBRACE stmt_list RBRACE { Block(List.rev $2) }

```

```

140 | FOR LPAREN expr FROM expr TO expr by RPAREN stmt { For($3, $5, $7, $8, $10) }
    | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
    | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Ast.Expr(Ast.Noexpr)) }
142 | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
    | BREAK SEMI { BreakCont(0) }
144 | CONT SEMI { BreakCont(1) }

146 stmt_list:
    /* nothing */ { [] }
148 | stmt_list stmt { $2 :: $1 }

150 rev_program:
    /* nothing */ { [] }
152 | rev_program fdecl { $2 :: $1 }

154 program:
    rev_program { List.rev $1 }

```

B.3 AST

ast.ml

```

1 (* Christopher Campbell, Winnie Narang *)
2 (* Elementary Data Types *)
3 type data_type =
4     Int
5     | Float
6     | Comp
7     | Mat

9 (* Unary Operators *)
10 type un_op =
11     Neg
12     | Not
13     | Re
14     | Im
15     | Norm
16     | Trans
17     | Det
18     | Adj
19     | Conj
20     | Unit
21     | Sin
22     | Cos
23     | Tan

25 (* Binary Operators *)
26 type bi_op =
27     Add
28     | Sub
29     | Mult
30     | Div
31     | Mod
32     | Expn
33     | Tens
34     | Eq
35     | Neq
36     | Lt
37     | Gt
38     | Leq

```

```

39 |   Geq
40 |   Or
41 |   And
42 |   Xor
43
44 (* Expressions *)
45 type expr =
46   Lit_int of int
47 |   Lit_float of float
48 |   Lit_comp of float * float
49 |   Lit_qub of string * int
50 |   Mat of expr list list
51 |   Id of string
52 |   Unop of un_op * expr
53 |   Binop of expr * bi_op * expr
54 |   Assign of string * expr
55 |   Call of string * expr list
56 |   Noexpr
57
58 (* Statements *)
59 type stmt =
60   Expr of expr
61 |   Block of stmt list
62 |   If of expr * stmt * stmt
63 |   For of expr * expr * expr * expr * stmt
64 |   While of expr * stmt
65 |   BreakCont of int
66
67 (* Statement Lists *)
68 type stmt_list =
69   stmt list
70
71 (* Variables Declaration *)
72 type var_decl =
73   {
74     typ : data_type;
75     name : string;
76   }
77
78 (* Function Declaration *)
79 type func_decl =
80   {
81     ret_typ : data_type;
82     ret_name : string;
83     func_name : string;
84     formal_params : var_decl list;
85     locals : var_decl list;
86     body : stmt list;
87   }
88
89 (* Program *)
90 type program =
91   func_decl list
92
93 (* Pretty Printer *)
94 let rec string_of_expr = function
95   Lit_int(n) -> string_of_int n
96 |   Lit_float(n) -> string_of_float n
97 |   Lit_comp(f1,f2) -> string_of_float f1 ^ " + " ^ string_of_float f2 ^ "i"
98 |   Lit_qub(s,t) -> let typ = string_of_int t in (match typ with
99     | 0 -> "Qub-bra of " ^ s
100    | _ -> "Qub-ket of " ^ s)
101 |   Mat(l) -> string_of_mat l
102 |   Id(s) -> s
103 |   Unop(unl,exp1) ->

```

```

105   (match un1 with
106     | Neg -> " - "
107     | Not -> " ! "
108     | Re -> " Re "
109     | Im -> " Im "
110     | Norm -> " Norm "
111     | Trans -> " Trans "
112     | Det -> " Det "
113     | Adj -> " Adj "
114     | Conj -> " Conj "
115     | Unit -> " Unit "
116     | Sin -> " Sin "
117     | Cos -> " Cos "
118     | Tan -> " Tan ") ^ string_of_expr expl

119 | Binop(ex1, binop, ex2) -> string_of_expr ex1 ^
  (match binop with
120    | Add -> " + " | Sub -> " - " | Mult -> " * "
121    | Div -> " / " | Mod -> " % " | Expn -> " ^ " | Tens -> " @ "
122    | Eq -> " = " | Neq -> " != " | Lt -> " < "
123    | Leq -> " <= " | Gt -> " > " | Geq -> " >= "
124    | Xor -> " XOR " | And -> " && " | Or -> " || ") ^ string_of_expr ex2
125 | Assign(str, expr) -> str ^ " = " ^ string_of_expr expr
126 | Call(str, expr_list) -> "Calling " ^ str ^ " on " ^ string_of_exprs expr_list
127 | Noexpr -> ""

128
129 and string_of_mat l =
130   let row_strs =
131     List.map string_of_row l
132   in
133     "[" ^ String.concat " " row_strs ^ "]"

134
135 and string_of_row r =
136   let row_str =
137     String.concat " " (List.map string_of_expr r)
138   in
139     "(" ^ row_str ^ ")"

140
141 and string_of_exprs exprs =
142   String.concat "\n" (List.map string_of_expr exprs)

143
144 and string_of_stmt = function
145   Expr(exp) -> string_of_expr exp ^ "\n"
146   | Block(stmt_list) -> "{" ^ string_of_stmts stmt_list ^ "\n}"
147   | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
148   | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^
149     string_of_stmt s2
150   | For(ex1, ex2, ex3, ex4, stmt) -> "For args : " ^ string_of_expr ex1 ^ " " ^ string_of_expr
151     ex2 ^ " " ^ string_of_expr ex3 ^ " " ^ string_of_expr ex4 ^ "\nstatement : \n" ^
152     string_of_stmt stmt
153   | While(expr, stmt) -> "While condition : " ^ string_of_expr expr ^ "\nstatement : " ^
154     string_of_stmt stmt
155   | BreakCont(t) -> string_of_breakcont t

156
157 and string_of_breakcont t =
158   if (t = 0) then
159     "break"
160   else
161     "continue"

162
163 and string_of_stmts stmts =
164   String.concat "\n" (List.map string_of_stmt stmts)

165
166 and string_of_var_decl var_decl =

```

```

165   "vdecl: typ: " ^
      (match var_decl.typ with
167     | Int -> "int", " ^ " name: " ^ var_decl.name ^ " "
      | Float -> "float", " ^ " name: " ^ var_decl.name ^ " "
169     | Comp -> "comp", " ^ " name: " ^ var_decl.name ^ " "
      | Mat -> "mat", " ^ " name: " ^ var_decl.name ^ " ")
171
172 and string_of_fdecl fdecl =
173   "\nfdecl:\nret_typ: " ^
      (match fdecl.ret_typ with
175     | Int -> " int "
      | Float -> " float "
177     | Comp -> " comp "
      | Mat -> " mat ") ^
179     "\nret_name: " ^ fdecl.ret_name ^ "\nfunc_name: " ^ fdecl.func_name ^ "\n(" ^
      String.concat " " (List.map string_of_var_decl fdecl.formal_params) ^ ")\n{" ^
181     String.concat " " (List.map string_of_var_decl fdecl.locals) ^ "\n" ^
      String.concat " " (List.map string_of_stmt fdecl.body) ^ "}"
183
184 and string_of_program (funcs) =
185   "program:\n" ^ String.concat "\n" (List.map string_of_fdecl funcs)

```

B.4 Analyzer

analyzer.ml

```

1 (* Christopher Campbell *)
   open Ast
3   open Sast

5   (*****
    * Environment *
7   *****)

9   type symbol_table =
      { ret_typ : Sast.sdata_type;
11     ret_nam : string;
      func_nam : string;
13     mutable formal_param : svar_decl list;
      mutable local : svar_decl list;
15     builtin : svar_decl list; }

17   type environment =
      { scope : symbol_table;
19     mutable functions : Sast.sfunc_decl list; }

21   let builtin_vars =
      [ { styp = Sast.Float; sname = "e"; builtinv = true; };
23     { styp = Sast.Float; sname = "pi"; builtinv = true; };
      { styp = Sast.Mat; sname = "X"; builtinv = true; };
25     { styp = Sast.Mat; sname = "Y"; builtinv = true; };
      { styp = Sast.Mat; sname = "Z"; builtinv = true; };
27     { styp = Sast.Mat; sname = "H"; builtinv = true; };
      { styp = Sast.Mat; sname = "IDT"; builtinv = true; }; ]

29   let builtin_funcs =
31     [ { sret_typ = Sast.Void;
        sret_name = "null";
33         sfunc_name = "print";
        sformal_params = [{ styp = Sast.Poly; sname = "print_val"; builtinv = true; }];

```

```

35     slocals  = [];
36     sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
37     builtinf = true; };

39 { sret_ttyp = Sast.Void;
40   sret_name = "null";
41   sfunc_name = "printq";
42   sformal_params = [{ styp = Sast.Mat; sname = "printq_val"; builtinv = true; }];
43   slocals  = [];
44   sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
45   builtinf = true; };

47 { sret_ttyp = Sast.Int;
48   sret_name = "null";
49   sfunc_name = "rows";
50   sformal_params = [{ styp = Sast.Mat; sname = "rows_val"; builtinv = true; }];
51   slocals  = [];
52   sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
53   builtinf = true; };

55 { sret_ttyp = Sast.Int;
56   sret_name = "null";
57   sfunc_name = "cols";
58   sformal_params = [{ styp = Sast.Mat; sname = "rows_val"; builtinv = true; }];
59   slocals  = [];
60   sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
61   builtinf = true; };

63 { sret_ttyp = Sast.Comp;
64   sret_name = "null";
65   sfunc_name = "elem";
66   sformal_params = [{ styp = Sast.Mat; sname = "elem_mat"; builtinv = true; };
67                     { styp = Sast.Int; sname = "elem_row"; builtinv = true; };
68                     { styp = Sast.Int; sname = "elem_col"; builtinv = true; }];
69   slocals  = [];
70   sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
71   builtinf = true; }; ]

73 let root_symbol_table =
74   { ret_ttyp = Sast.Void;
75     ret_nam = "";
76     func_nam = "";
77     formal_param = [];
78     local = [];
79     builtin = builtin_vars; }

81 let root_environment =
82   { scope = root_symbol_table;
83     functions = builtin_funcs; }

85 (*****
86  * Exceptions *
87  *****)

89 exception Except of string

91 let matrix_error t = match t with
92   0 -> raise (Except("Invalid matrix: incorrect type"))
93   | _ -> raise (Except("Invalid matrix"))

95 let qub_error t = match t with
96   0 -> raise (Except("Invalid qubit: incorrect use of |expr>"))
97   | 1 -> raise (Except("Invalid qubit: incorrect use of <expr|"))
98   | _ -> raise (Except("Invalid qubit"))
99

```

```

101 let assignment_error s =
    raise (Except("Invalid assignment to variable: " ^ s))

103 let var_error s =
    raise (Except("Invalid use of a variable: " ^ s ^ " was not declared" ))

105 let func_error s =
    raise (Except("Invalid function call: " ^ s ^ " was not declared" ))

107 let var_decl_error s =
    raise (Except("Invalid variable declaration: " ^ s ^ " was already declared" ))

109 let func_decl_error s =
    raise (Except("Invalid function declaration: " ^ s ^ " was already declared" ))

111 let unop_error t = match t with
112 | Ast.Neg -> raise (Except("Invalid use of unop: '-expr'"))
113 | Ast.Not -> raise (Except("Invalid use of unop: 'Not(expr)'))
114 | Ast.Re -> raise (Except("Invalid use of unop: 'Re(expr)'))
115 | Ast.Im -> raise (Except("Invalid use of unop: 'Im(expr)'))
116 | Ast.Norm -> raise (Except("Invalid use of unop: 'Norm(expr)'))
117 | Ast.Trans -> raise (Except("Invalid use of unop: 'Trans(expr)'))
118 | Ast.Det -> raise (Except("Invalid use of unop: 'Det(expr)'))
119 | Ast.Adj -> raise (Except("Invalid use of unop: 'Adj(expr)'))
120 | Ast.Conj -> raise (Except("Invalid use of unop: 'Conjexpr'"))
121 | Ast.Unit -> raise (Except("Invalid use of unop: 'Unit(expr)'))
122 | Ast.Sin -> raise (Except("Invalid use of unop: 'Sin(expr)'))
123 | Ast.Cos -> raise (Except("Invalid use of unop: 'Cos(expr)'))
124 | Ast.Tan -> raise (Except("Invalid use of unop: 'Tan(expr)'))

125 let binop_error t = match t with
126 | Ast.Add -> raise (Except("Invalid use of binop: 'expr + expr'"))
127 | Ast.Sub -> raise (Except("Invalid use of binop: 'expr - expr'"))
128 | Ast.Mult -> raise (Except("Invalid use of binop: 'expr * expr'"))
129 | Ast.Div -> raise (Except("Invalid use of binop: 'expr / expr'"))
130 | Ast.Mod -> raise (Except("Invalid use of binop: 'expr % expr'"))
131 | Ast.Expn -> raise (Except("Invalid use of binop: 'expr ^ expr'"))
132 | Ast.Or -> raise (Except("Invalid use of binop: 'expr or expr'"))
133 | Ast.And -> raise (Except("Invalid use of binop: 'expr and expr'"))
134 | Ast.Xor -> raise (Except("Invalid use of binop: 'expr xor expr'"))
135 | Ast.Tens -> raise (Except("Invalid use of binop: 'expr @ expr'"))
136 | Ast.Eq -> raise (Except("Invalid use of binop: 'expr eq expr'"))
137 | Ast.Neq -> raise (Except("Invalid use of binop: 'expr neq expr'"))
138 | Ast.Lt -> raise (Except("Invalid use of binop: 'expr lt expr'"))
139 | Ast.Gt -> raise (Except("Invalid use of binop: 'expr gt expr'"))
140 | Ast.Leq -> raise (Except("Invalid use of binop: 'expr leq expr'"))
141 | Ast.Geq -> raise (Except("Invalid use of binop: 'expr geq expr'"))

142 let expr_error t = match t with
143 | _ -> raise (Except("Invalid expression"))

144 let call_error t = match t with
145 | 0 -> raise (Except("Invalid function call: function undeclared"))
146 | 1 -> raise (Except("Invalid function call: incorrect number of parameters"))
147 | 2 -> raise (Except("Invalid function call: incorrect type for parameter"))
148 | _ -> raise (Except("Invalid function call"))

149 let stmt_error t = match t with
150 | 0 -> raise (Except("Invalid use of statment: 'if'"))
151 | 1 -> raise (Except("Invalid use of statment: 'for'"))
152 | 2 -> raise (Except("Invalid use of statment: 'while'"))
153 | _ -> raise (Except("Invalid statement"))

154 let program_error t = match t with
155 | 0 -> raise (Except("Missing 'compute' function"))

```



```

165 | 1 -> raise (Except("'compute' function must be of type int"))
166 | _ -> raise (Except("Invalid program"))
167
168 (*****
169  * Utility Functions *
170  *****)
171
172 let var_exists name scope =
173   if (List.exists (fun vdecl -> name = vdecl.sname) scope.formal_param) then true
174   else if (List.exists (fun vdecl -> name = vdecl.sname) scope.formal_param) then true
175   else List.exists (fun vdecl -> name = vdecl.sname) scope.builtin
176
177 let func_exists name env =
178   List.exists (fun fdecl -> name = fdecl.sfunc_name) env.functions
179
180 let lookup_var name scope =
181   let vdecl_found =
182     try List.find (fun vdecl -> name = vdecl.sname) scope.formal_param
183   with Not_found ->
184     try List.find (fun vdecl -> name = vdecl.sname) scope.local
185   with Not_found ->
186     try List.find (fun vdecl -> name = vdecl.sname) scope.builtin
187   with Not_found -> var_error name in
188   vdecl_found
189
190 let lookup_func name env =
191   let fdecl_found =
192     try
193       List.find (fun fdecl -> name = fdecl.sfunc_name) env.functions
194     with Not_found -> func_error name
195   in
196   fdecl_found
197
198 (*****
199  * Checks *
200  *****)
201
202 let rec check_qub_expr i =
203   let r = i mod 10 in
204   if (r = 0 || r = 1) then
205     let i = i / 10 in
206     if (i != 0)
207     then
208       check_qub_expr i
209     else 1
210   else 0
211
212 and check_qub i t =
213   let int_expr =
214     int_of_string i
215   in
216   if (check_qub_expr int_expr = 1) then
217     (match t with
218      0 -> Sast.Expr(Sast.Lit_qub(i, 1), Sast.Mat)
219      | 1 -> Sast.Expr(Sast.Lit_qub(i, 0), Sast.Mat)
220      | _ -> qub_error 2)
221   else
222     qub_error t
223
224 and check_mat l env =
225   let mat =
226     List.map (fun row -> check_mat_rows row env) l
227   in
228   Sast.Expr(Sast.Mat(mat), Sast.Mat)
229

```

```

231 and check_mat_rows l env =
    let row =
        List.map (fun e -> check_mat_row e env) l
233    in row

235 and check_mat_row e env =
    let se =
237        check_expr env e
    in
239    match se with
        Sast.Expr(_, t) ->
241        match t with
            Sast.Int -> se
243            | Sast.Float -> se
            | Sast.Comp -> se
245            | _ -> matrix_error 0

247 and check_id name env =
    let vdecl =
249        lookup_var name env.scope
    in
251    let typ = vdecl.styp in
        Sast.Expr(Sast.Id(name), typ)
253

255 and check_unop op e env =
    let e = check_expr env e in
    match e with
257    Sast.Expr(q, t) ->
        (match op with
259        Ast.Neg ->
            (match t with
261            Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
            | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
263            | Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
            | _ -> unop_error op)
265        | Ast.Not ->
            (match t with
267            Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
            | _ -> unop_error op)
269        | Ast.Re ->
            (match t with
271            Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
            | _ -> unop_error op)
273        | Ast.Im ->
            (match t with
275            Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
            | _ -> unop_error op)
277        | Ast.Unit ->
            (match t with
279            Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
            | _ -> unop_error op)
281        | Ast.Norm ->
            (match t with
283            Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
            | _ -> unop_error op)
285        | Ast.Det ->
            (match t with
287            Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
            | _ -> unop_error op)
289        | Ast.Trans | Ast.Adj ->
            (match t with
291            Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Mat)
            | _ -> unop_error op)
293        | Ast.Conj ->
            (match t with

```

```

295         Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
      | Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Mat)
297     | _ -> unop_error op)
| Ast.Sin ->
299     (match t with
      Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
301     | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
      Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
303     | _ -> unop_error op)
| Ast.Cos ->
305     (match t with
      Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
307     | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
      Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
309     | _ -> unop_error op)
| Ast.Tan ->
311     (match t with
      Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
313     | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
      Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
315     | _ -> unop_error op))

317 and check_binop e1 op e2 env =
    let e1 = check_expr env e1 and e2 = check_expr env e2 in
319     match e1 with
      Sast.Expr(_, t1) ->
321     (match e2 with
      Sast.Expr(_, t2) ->
323     (match op with
      Ast.Add | Ast.Sub ->
325     (match t1 with
      Sast.Int ->
327     (match t2 with
      Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
329     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
      Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
331     | _ -> binop_error op)
      | Sast.Float ->
333     (match t2 with
      Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
335     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
      Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
337     | _ -> binop_error op)
      | Sast.Comp ->
339     (match t2 with
      Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
341     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
      Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
343     | _ -> binop_error op)
      | Sast.Mat ->
345     (match t2 with
      Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
347     | _ -> binop_error op)
      | _ -> binop_error op)
      Ast.Mult | Ast.Div ->
349     (match t1 with
      Sast.Int ->
351     (match t2 with
      Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
353     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
      Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
355     | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
357     | _ -> binop_error op)
      | Sast.Float ->
359     (match t2 with

```

```

361         Sast.Int | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.
Float)
362         | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
363         | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
364         | _ -> binop_error op)
365     | Sast.Comp ->
366     (match t2 with
367     Sast.Int | Sast.Float | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op,
e2), Sast.Comp)
368     | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
369     | _ -> binop_error op)
370     | Sast.Mat ->
371     (match t2 with
372     Sast.Int | Sast.Float | Sast.Comp | Sast.Mat -> Sast.Expr(Sast.
Binop(e1, op, e2), Sast.Mat)
373     | _ -> binop_error op)
374     | _ -> binop_error op)
375     | Ast.Mod | Ast.Expn ->
376     (match t1 with
377     Sast.Int ->
378     (match t2 with
379     Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
380     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
381     | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
382     | _ -> binop_error op)
383     | Sast.Float ->
384     (match t2 with
385     Sast.Int | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.
Float)
386     | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
387     | _ -> binop_error op)
388     | Sast.Comp ->
389     (match t2 with
390     Sast.Int | Sast.Float | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op,
e2), Sast.Comp)
391     | _ -> binop_error op)
392     | _ -> binop_error op)
393     | Ast.Tens ->
394     (match t1 with
395     Sast.Mat ->
396     (match t2 with
397     Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
398     | _ -> binop_error op)
399     | _ -> binop_error op)
400     | Ast.Eq | Ast.Neq ->
401     (match t1 with
402     Sast.Int ->
403     (match t2 with
404     Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
405     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
406     | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
407     | _ -> binop_error op)
408     | Sast.Float ->
409     (match t2 with
410     Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
411     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
412     | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
413     | _ -> binop_error op)
414     | Sast.Comp ->
415     (match t2 with
416     Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
417     | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
418     | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
419     | _ -> binop_error op)
420     | Sast.Mat ->

```

```

421         (match t2 with
422           Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
423           | _ -> binop_error op)
424       | _ -> binop_error op)
425   | Ast.Lt | Ast.Gt | Ast.Leq | Ast.Geq ->
426     (match t1 with
427       Sast.Int ->
428         (match t2 with
429           Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
430           | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
431           | _ -> binop_error op)
432       | Sast.Float ->
433         (match t2 with
434           Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
435           | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
436           | _ -> binop_error op)
437       | _ -> binop_error op)
438   | Ast.Or | Ast.And | Ast.Xor ->
439     (match t1 with
440       Sast.Int ->
441         (match t2 with
442           Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
443           | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
444           | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
445           | _ -> binop_error op)
446       | Sast.Float ->
447         (match t2 with
448           Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
449           | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
450           | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
451           | _ -> binop_error op)
452       | Sast.Comp ->
453         (match t2 with
454           Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
455           | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
456           | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
457           | _ -> binop_error op)
458       | _ -> binop_error op)))
459 and check_assign name e env =
460   let vdecl = lookup_var name env.scope in
461   let e = check_expr env e in
462   match e with
463   | Sast.Expr(_, t1) ->
464     let t2 = vdecl.styp in
465     if (t1 = t2) then
466       Sast.Expr(Sast.Assign(name, e), t1)
467     else
468       assignment_error name
469 and check_call_params formal_params params =
470   if ((List.length formal_params) = 0)
471   then true
472   else
473     let fdecl_arg = List.hd formal_params in
474     let param = match (List.hd params) with
475     | Sast.Expr(_, t) -> t in
476     if (fdecl_arg.styp = Sast.Poly || (fdecl_arg.styp = param))
477     then check_call_params (List.tl formal_params) (List.tl params)
478     else false
479 and check_call name params env =
480   let fdecl =
481     try
482       lookup_func name env

```

```

485 with Not_found -> call_error 0 in
486 let params = List.map (check_expr env) params in
487   if ((List.length fdecl.sformal_params) != (List.length params))
488     then call_error 1
489   else
490     if ((check_call_params fdecl.sformal_params params) = true)
491       then Sast.Expr(Sast.Call(name, params), fdecl.sret_ttyp)
492     else
493       call_error 2
494
495 and check_expr env = function
496   Ast.Lit_int(i) -> Sast.Expr(Sast.Lit_int(i), Sast.Int)
497 | Ast.Lit_float(f) -> Sast.Expr(Sast.Lit_float(f), Sast.Float)
498 | Ast.Lit_comp(f1, f2) -> Sast.Expr(Sast.Lit_comp(f1, f2), Sast.Comp)
499 | Ast.Lit_qub(i, t) -> check_qub i t
500 | Ast.Mat(l) -> check_mat l env
501 | Ast.Id(s) -> check_id s env
502 | Ast.Unop(op, e) -> check_unop op e env
503 | Ast.Binop(e1, op, e2) -> check_binop e1 op e2 env
504 | Ast.Assign(s, e) -> check_assign s e env
505 | Ast.Call(s, l) -> check_call s l env
506 | Ast.Noexpr -> Sast.Expr(Sast.Noexpr, Sast.Void)
507
508 and check_block stmts env =
509   let sstmts = List.map (fun stmt -> check_stmt env stmt) stmts in
510   Sast.Block(sstmts)
511
512 and check_if e s1 s2 env =
513   let se = check_expr env e in
514   match se with
515   | Sast.Expr(_, t) ->
516     (match t with
517     | Sast.Int ->
518       let ss1 = check_stmt env s1 in
519       let ss2 = check_stmt env s2 in
520       Sast.If(se, ss1, ss2)
521     | _ -> stmt_error 0)
522
523 and check_for e1 e2 e3 e4 s env =
524   let se1 = check_expr env e1 in
525   match se1 with
526   | Sast.Expr(Sast.Id(_), Sast.Int) ->
527     let se2 = check_expr env e2 in
528     (match se2 with
529     | Sast.Expr(_, Sast.Int) ->
530       let se3 = check_expr env e3 in
531       (match se3 with
532       | Sast.Expr(_, Sast.Int) ->
533         let se4 = check_expr env e4 in
534         (match se4 with
535         | Sast.Expr(_, t) ->
536           (match t with
537           | Sast.Int ->
538             let ss = check_stmt env s in
539             Sast.For(se1, se2, se3, se4, ss)
540         | Sast.Void ->
541           let ss = check_stmt env s in
542           Sast.For(se1, se2, se3, Sast.Expr(Sast.Lit_int(1), Sast.Int), ss)
543         | _ -> stmt_error 1))
544       | _ -> stmt_error 1)
545     | _ -> stmt_error 1)
546   | _ -> stmt_error 1
547
548 and check_while e s env =
549   let se = check_expr env e in

```

```

match se with
551   Sast.Expr(Sast.Binop(_, op, _), Sast.Int) ->
      (match op with
553       Ast.Eq | Ast.Neq | Ast.Lt | Ast.Gt | Ast.Leq | Ast.Geq ->
          let ss = check_stmt env s in
          Sast.While(se, ss)
555       | _ -> stmt_error 2)
557   | _ -> stmt_error 2

559 and check_stmt env = function
      Ast.Expr(e) -> Sast.Sexpr(check_expr env e)
561   | Ast.Block(l) -> check_block l env
563   | Ast.If(e, s1, s2) -> check_if e s1 s2 env
565   | Ast.For(e1, e2, e3, e4, s) -> check_for e1 e2 e3 e4 s env
567   | Ast.While(e, s) -> check_while e s env
569   | Ast.BreakCont(t) -> Sast.BreakCont(t)

567 and vdecl_to_sdecl vdecl =
      match vdecl.typ with
569       Ast.Int -> { styp = Sast.Int; sname = vdecl.name; builtinv = false; }
571       | Ast.Float -> { styp = Sast.Float; sname = vdecl.name; builtinv = false; }
573       | Ast.Comp -> { styp = Sast.Comp; sname = vdecl.name; builtinv = false; }
575       | Ast.Mat -> { styp = Sast.Mat; sname = vdecl.name; builtinv = false; }

577 and formal_to_sformal scope formal_param =
      let found = var_exists formal_param.name scope in
      if found then var_decl_error formal_param.name
579   else let sdecl = vdecl_to_sdecl formal_param in
      let new_formals = sdecl :: scope.formal_param in
      let new_scope =
581         { ret_typ = scope.ret_typ;
          ret_nam = scope.ret_nam;
          func_nam = scope.func_nam;
          formal_param = new_formals;
          local = scope.local;
          builtin = scope.builtin; } in
      new_scope

587 and formals_to_sformals scope formal_params =
      let new_scope =
589         if (formal_params = []) then scope
591         else List.fold_left formal_to_sformal scope (List.rev formal_params) in
      new_scope

593 and local_to_slocal scope local =
      let found = var_exists local.name scope in
      if found then var_decl_error local.name
597   else let sdecl = vdecl_to_sdecl local in
      let new_locals = sdecl :: scope.local in
      let new_scope =
599         { ret_typ = scope.ret_typ;
          ret_nam = scope.ret_nam;
          func_nam = scope.func_nam;
          formal_param = scope.formal_param;
          local = new_locals;
          builtin = scope.builtin; } in
      new_scope

607 and locals_to_slocals scope locals =
      let new_scope = List.fold_left local_to_slocal scope (List.rev locals) in
      new_scope

611 and ret_to_sret scope ret_typ =
613   let sret_typ =
      match ret_typ with

```

```

615     Ast.Int -> Sast.Int
        | Ast.Float -> Sast.Float
617     | Ast.Comp -> Sast.Comp
        | Ast.Mat -> Sast.Mat
619   in
620   let new_scope =
621     { ret_typ = sret_typ;
622       ret_nam = scope.ret_nam;
623       func_nam = scope.func_nam;
624       formal_param = scope.formal_param;
625       local = scope.local;
626       builtin = scope.builtin; } in
627   new_scope

629 and rname_to_sname scope ret_name =
630   let new_scope = { ret_typ = scope.ret_typ;
631                     ret_nam = ret_name;
632                     func_nam = scope.func_nam;
633                     formal_param = scope.formal_param;
634                     local = scope.local; builtin = scope.builtin; } in
635   new_scope

637 and fname_to_sfname scope func_name =
638   let new_scope = { ret_typ = scope.ret_typ;
639                     ret_nam = scope.ret_nam;
640                     func_nam = func_name;
641                     formal_param = scope.formal_param;
642                     local = scope.local;
643                     builtin = scope.builtin; } in
644   new_scope

645 and ret_to_slocal scope name typ =
646   let vdecl = { typ = typ; name = name; } in
647   let sdecl = vdecl_to_sdecl vdecl in
648   let new_locals = sdecl :: scope.local in
649   let new_scope = { ret_typ = scope.ret_typ;
650                     ret_nam = scope.ret_nam;
651                     func_nam = scope.func_nam;
652                     formal_param = scope.formal_param;
653                     local = new_locals;
654                     builtin = scope.builtin; } in
655   new_scope

657 and fdecl_to_sdecl fdecl env =
658   let new_scope = ret_to_slocal env.scope fdecl.ret_name fdecl.ret_typ in
659   let new_scope = formals_to_sformals new_scope fdecl.formal_params in
660   let new_scope = locals_to_slocals new_scope fdecl.locals in
661   let new_scope = ret_to_sret new_scope fdecl.ret_typ in
662   let new_scope = rname_to_sname new_scope fdecl.ret_name in
663   let new_scope = fname_to_sfname new_scope fdecl.func_name in
664   let new_env = { scope = new_scope; functions = env.functions; } in
665   let stmts = List.map (fun stmt -> check_stmt new_env stmt) fdecl.body in
666   { sret_typ = new_scope.ret_typ;
667     sret_name = new_scope.ret_nam;
668     sfunc_name = new_scope.func_nam;
669     sformal_params = new_scope.formal_param;
670     slocals = new_scope.local;
671     sbody = stmts;
672     builtinf = false; }

675 and check_function env fdecl =
676   let found = func_exists fdecl.func_name env in
677   if found then func_decl_error fdecl.func_name
678   else let sfdecl = fdecl_to_sdecl fdecl env in
679   let new_env = { scope = env.scope; functions = sfdecl :: env.functions; } in

```



```

new_env
681
and check_compute_fdecl fdecls =
683   let fdecl = List.hd (List.rev fdecls) in
   let name = fdecl.func_name in
685   if (name = "compute") then fdecls
   else program_error 0
687
and check_program fdecls =
689   let fdecls = check_compute_fdecl fdecls in
   let env = List.fold_left check_function root_environment fdecls in
691   let sfdecls = List.rev env.functions in
   sfdecls

```

B.5 SAST

sast.ml

```

(* Sankalpa Khadka *)
2 open Ast

4 type sdata_type =
   Int
6   | Float
   | Comp
8   | Mat
   | Poly
10  | Void

12 type expr_wrapper =
   Expr of sexpr * sdata_type
14

and sexpr =
16   Lit_int of int
   | Lit_float of float
18   | Lit_comp of float * float
   | Lit_qub of string * int
20   | Mat of expr_wrapper list list
   | Id of string
22   | Unop of Ast.un_op * expr_wrapper
   | Binop of expr_wrapper * Ast.bi_op * expr_wrapper
24   | Assign of string * expr_wrapper
   | Call of string * expr_wrapper list
26   | Noexpr

28 and sstmt =
   Sexpr of expr_wrapper
30   | Block of sstmt list
   | If of expr_wrapper * sstmt * sstmt
32   | For of expr_wrapper * expr_wrapper * expr_wrapper * expr_wrapper * sstmt
   | While of expr_wrapper * sstmt
34   | BreakCont of int

36 and svar_decl =
   {
38     styp : sdata_type;
     sname : string;
40     builtinv : bool;
   }
42

```

```

and sfunc_decl =
44 {
    sret_type : sdata_type;
46    sret_name : string;
    sfunc_name : string;
48    sformal_params : svar_decl list;
    slocals : svar_decl list;
50    sbody : sstmt list;
    builtinf : bool;
52 }

54 type sprogram =
    sfunc_decl list

56 (* Pretty Printer *)
58 let rec string_of_unop op e =
    (match op with
60   Neg -> " -"
    | Not -> " !"
62   Re -> " Re "
    | Im -> " Im "
64   Norm -> " Norm "
    | Trans -> " Trans "
66   Det -> " Det "
    | Adj -> " Adj "
68   Conj -> " Conj "
    | Unit -> " Unit "
70   Sin -> " Sin "
    | Cos -> " Cos "
72   Tan -> " Tan ") ^ string_of_expr_wrapper e

74 and string_of_binop e1 op e2 =
    string_of_expr_wrapper e1 ^
76    (match op with
        Add -> " + " | Sub -> " - " | Mult -> " * "
78        | Div -> " / " | Mod -> " % " | Expn -> " ^ " | Tens -> " @ "
        | Eq -> " == " | Neq -> " != " | Lt -> " < "
80        | Leq -> " <= " | Gt -> " > " | Geq -> " >= "
        | Xor -> " XOR " | And -> " && " | Or -> " || ") ^ string_of_expr_wrapper e2

82 and string_of_mat l =
84   let row_strs =
        List.map string_of_row l
86   in
        "[" ^ String.concat " " row_strs ^ "]"

88 and string_of_row r =
90   let row_str =
        String.concat " " (List.map string_of_expr_wrapper r)
92   in
        "(" ^ row_str ^ ")"

94 and string_of_sexpr = function
96   Lit_int(i) -> string_of_int i
    | Lit_float(f) -> string_of_float f
98   | Lit_comp(f1, f2) -> string_of_float f1 ^ " + " ^ string_of_float f2 ^ "i"
    | Lit_qub(i, t) -> i
100  | Mat(l) -> string_of_mat l
    | Id(s) -> s
102  | Unop(op, e) -> string_of_unop op e
    | Binop(e1, op, e2) -> string_of_binop e1 op e2
104  | Assign(name, e) -> name ^ " = " ^ string_of_expr_wrapper e
    | Call(name, params) -> "Calling " ^ name ^ " on " ^ string_of_sexprs params
106  | Noexpr -> "noexpr"

```

```

108 and string_of_expr_wrapper w =
    let sexpr =
110         match w with
            Expr(Lit_int(i), Int) -> Lit_int(i)
112         | Expr(Lit_float(f), Float) -> Lit_float(f)
            | Expr(Lit_comp(f1, f2), Comp) -> Lit_comp(f1, f2)
114         | Expr(Mat(l), Mat) -> Mat(l)
            | Expr(Id(name), typ) -> Id(name)
116         | Expr(Unop(op, e), _) -> Unop(op, e)
            | Expr(Binop(e1, op, e2), _) -> Binop(e1, op, e2)
118         | Expr(Assign(name, e), t1) -> Assign(name, e)
            | Expr(Call(name, params), _) -> Call(name, params)
120         | Expr(Lit_qub(i, t), _) -> Lit_qub(i, t)
            | _ -> Noexpr
122     in
        string_of_sexpr sexpr
124
125 and string_of_svar_decl svar_decl =
126     "svdecl: styp: " ^
        (match svar_decl.styp with
128         | Int -> "int," ^ " name: " ^ svar_decl.sname ^ " "
            | Float -> "float," ^ " name: " ^ svar_decl.sname ^ " "
130         | Comp -> "comp," ^ " name: " ^ svar_decl.sname ^ " "
            | Mat -> "mat," ^ " name: " ^ svar_decl.sname ^ " "
132         | _ -> "")
134
135 and string_of_sexprs e =
    String.concat "\n" (List.map string_of_expr_wrapper e)
136
137 and string_of_sstmt = function
138     Sexpr(e) -> string_of_expr_wrapper e ^ "\n"
    | Block(l) -> "{\n" ^ string_of_sstmts l ^ "\n}"
140     | If(e, s, Block([])) -> "if (" ^ string_of_expr_wrapper e ^ ")\n" ^ string_of_sstmt s
    | If(e, s1, s2) -> "if (" ^ string_of_expr_wrapper e ^ ")\n" ^ string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
142     | For(e1, e2, e3, e4, s) -> "For args : " ^ string_of_expr_wrapper e1 ^ " " ^
        string_of_expr_wrapper e2 ^ " " ^ string_of_expr_wrapper e3 ^
        " " ^ string_of_expr_wrapper e4 ^ "\nstatement : \n" ^
        string_of_sstmt s
144     | While(e, s) -> "While condition : " ^ string_of_expr_wrapper e ^ "\nstatement : " ^
        string_of_sstmt s
    | BreakCont(t) -> string_of_breakcont t
146
147 and string_of_breakcont t =
148     if (t = 0) then
        "break"
150     else
        "continue"
152
153 and string_of_sstmts sstmts =
154     String.concat "\n" (List.map string_of_sstmt sstmts)
156
157 and string_of_sfdecl sfdecl =
    "\nsfdecl:\nsret_typ: " ^
158     (match sfdecl.sret_typ with
        Int -> "int"
160     | Float -> "float"
        | Comp -> "comp"
162     | Mat -> "mat"
        | _ -> "") ^
    "\nsret_name: " ^ sfdecl.sret_name ^ "\nsfunc_name: " ^ sfdecl.sfunc_name ^ "\n(" ^
164     String.concat "" (List.map string_of_svar_decl sfdecl.sformal_params) ^ ")\n{" ^
    String.concat "" (List.map string_of_svar_decl sfdecl.slocals) ^ "\n" ^
166     String.concat "" (List.map string_of_sstmt sfdecl.sbody) ^ "}"
168

```

```

170 and string_of_sprogram (l) =
    "program:\n" ^ String.concat "\n" (List.map string_of_sfdecl l)

```

B.6 Generator

generator.ml

```

(* Winnie Narang, Jonathan Wong, Sankalpa Khadka *)
2 open Sast
  open Printf
4 open String

6 let builtin_funcs = ["print"; "printw"; "rows"; "cols"; "elem"]

8 let is_builtin_func name =
    List.exists (fun func_name -> func_name = name) builtin_funcs
10
(* get type *)
12 let type_of (a : Sast.expr_wrapper) : Sast.sdata_type =
    match a with
14   | Expr(_,t)-> t

16 (* get expression from expression wrapper *)
  let expr_of (a : Sast.expr_wrapper) : Sast.sexpr =
18     match a with
        | Expr(e,_)-> e
20
(* generate type *)
22 let rec cpp_from_type (ty: Sast.sdata_type) : string =
    match ty with
24   | Int -> "int"
    | Float -> "float"
26   | Comp -> "complex<float>"
    | Mat -> "MatrixXcf"
28   | Poly | Void -> " "

30 (* write program to .cpp file *)
  and writeToFile fileName progString =
32     let file = open_out (fileName ^ ".cpp") in
        fprintf file "%s" progString
34
(* entry point for code generation*)
36 and gen_program fileName prog =
    let cppString = writeCpp prog in
38     let out = sprintf "
        #include <iostream>
        #include <complex>
        #include <cmath>
        #include <Eigen/Dense>
        #include <qlang>
        using namespace Eigen;
        using namespace std;
        %s" cppString in
46     writeToFile fileName out;
48
(* list of function declaration*)
50 and writeCpp funcList =
    let outStr =
52     List.fold_left (fun a b -> a ^ (cpp_funcList b)) "" funcList
    in

```

```

54     sprintf "%s" outStr
55
56 (* generate functions *)
57 and cpp_funcList func =
58     if func.builtinf then
59         ""
60     else
61         let cppFName = func.sfunc_name
62         and cppRtnType = cpp_from_type func.sret_type
63         and cppRtnValue = func.sret_name
64         and cppFParam = if (func.sformal_params = []) then "" else cppVarDecl func.
sformal_params ","
65         and cppFBody = cppStmtList func.sbody
66         and cppLocals = cppVarDecl func.slocals ";\n\t"
67         in
68         if cppFName = "compute" then
69             sprintf "\nint main ()\n{\n\t%s\n\t%s\n\tstd::cout << %s << endl;\n\n\
treturn 0;\n}" cppLocals cppFBody cppRtnValue
70         else
71             if (cppFParam = "") then
72                 sprintf "\n%s %s ()\n{\n\t%s\n\t%s\n\treturn %s;\n}" cppRtnType cppFName cppLocals
cppFBody cppRtnValue
73             else
74                 sprintf "\n%s %s (%s)\n{\n\t%s\n\t%s\n\treturn %s;\n}" cppRtnType cppFName cppFParam
cppLocals cppFBody cppRtnValue
75
76 (* generate variable declarations *)
77 and cppVarDecl vardecllist delim =
78     let varDecStr =
79         List.fold_left (fun a b -> a ^ (cppVar b delim)) "" vardecllist
80     in
81     let varDectrun = String.sub varDecStr 0 ((String.length varDecStr)-1)
82     in
83     sprintf "%s " varDectrun
84
85 (* generate variable declaration *)
86 and cppVar var delim =
87     if not var.builtinv then
88         let vartype =
89             cpp_from_type var.styp
90         in
91         sprintf "%s %s%s" vartype var.sname delim
92     else ""
93
94 (* generate list of statements *)
95 and cppStmtList astmtlist =
96     let outStr =
97         List.fold_left (fun a b -> a ^ (cppStmt b)) "" astmtlist
98     in
99     sprintf "%s" outStr
100
101 (* generate statement *)
102 and cppStmt stmts = match stmts with
103 | Sast.Sexpr(expr_wrap) -> "\t" ^ cppExpr (expr_of expr_wrap) ^ ";\n"
104 | Sast.Block(sstmt) -> cppStmtBlock sstmt
105 | Sast.If(expr_wrap, sstmt1, sstmt2) -> writeIfStmt (expr_of expr_wrap) sstmt1 sstmt2
106 | Sast.For(var, init, final, increment, stmt) -> writeForStmt var init final increment stmt
107 | Sast.While(expr_wrap, sstmt) -> writeWhileStmt (expr_of expr_wrap) sstmt
108 | Sast.BreakCont(t) -> writeBreakCont t
109
110 (* generate break/continue statement *)
111 and writeBreakCont t =
112     if (t = 0) then
113         sprintf "break;"
114     else

```

```

116     sprintf "continue;"
117
118 (* generate expression *)
119 and cppExpr expr = match expr with
120   | Lit_int(lit) -> string_of_int lit
121   | Lit_float(flit) -> string_of_float flit
122   | Lit_comp(re,im) -> " complex<float>(" ^ string_of_float re ^ "," ^ string_of_float im ^
123     ")" (* Not sure how to do this *)
124   | Unop(op, expr) -> writeUnop op expr
125   | Binop(expr1, op, expr2) -> writeBinop expr1 op expr2
126   | Lit_qub(vec, t) -> writeQubit vec t
127   | Mat (expr_wrap) -> writeMatrix expr_wrap
128   | Id(str) -> str
129   | Assign(name, expr) -> name ^ " = " ^ cppExpr (expr_of expr)
130   | Call(name,l) ->
131     if is_builtin_func name then
132       writeBuiltinFuncCall name l
133     else
134       name ^ "(" ^ writeFunCall l ^ ")"
135   | Noexpr -> ""
136
137 (* generate built-in function call *)
138 and writeBuiltinFuncCall name l =
139   match name with
140   | "print" -> writePrintStmt l
141   | "printq" -> writePrintqStmt l
142   | "rows" -> writeRowStmt l
143   | "cols" -> writeColStmt l
144   | "elem" -> writeElemStmt l
145   | _ -> ""
146
147 (* generate row statement *)
148 and writeRowStmt l =
149   let expr_wrap = List.hd l in
150   let expr = cppExpr (expr_of expr_wrap) in
151   sprintf "%s.rows()" expr
152
153 (* generate col statement *)
154 and writeColStmt l =
155   let expr_wrap = List.hd l in
156   let expr = cppExpr (expr_of expr_wrap) in
157   sprintf "%s.cols()" expr
158
159 (* generate elem statement *)
160 and writeElemStmt l =
161   let ew1 = List.hd l in
162   let e1 = cppExpr (expr_of ew1)
163   and ew2 = List.hd (List.tl l) in
164   let e2 = cppExpr (expr_of ew2)
165   and ew3 = List.hd (List.tl (List.tl l)) in
166   let e3 = cppExpr (expr_of ew3) in
167   sprintf "%s(%s,%s)" e1 e2 e3
168
169 (* generate print statement *)
170 and writePrintStmt l =
171   let expr_wrap = List.hd l in
172   let expr = cppExpr (expr_of expr_wrap) in
173   match expr_wrap with
174   | Sast.Expr(_,t) ->
175     (match t with
176     | Sast.Mat -> sprintf "cout << %s << endl" expr
177     | _ -> sprintf "cout << %s << endl" expr)
178
179 (* generate qubit print statement *)
180 and writePrintqStmt l =

```

```

180 let expr_wrap = List.hd 1 in
181 let expr = cppExpr (expr_of expr_wrap) in
182   match expr_wrap with
183   | Sast.Expr(_, t) ->
184     (match t with
185      | Sast.Mat -> sprintf "cout << vectorToBraket(%s) << endl" expr
186      | _ -> sprintf "cout << %s << endl" expr)
187
188 (* generate block *)
189 and cppStmtBlock sstmt1 =
190   let slist = List.fold_left (fun output element ->
191     let stmt = cppStmt element in
192     output ^ stmt ^ "\n") "" sstmt1 in
193   "\n\t{\n" ^ slist ^ "\t}\n"
194
195 (* generate if statement *)
196 and writeIfStmt expr stmt1 stmt2 =
197   let cond = cppExpr expr in
198   let body = cppStmt stmt1 in
199   let ebody = writeElseStmt stmt2 in
200   sprintf " if(%s)%s%s" cond body ebody
201
202 (* generate else statements *)
203 and writeElseStmt stmt =
204   let body =
205     cppStmt stmt
206   in
207   if ((String.compare body "\t;\n") = 0) then
208     sprintf "\n"
209   else
210     sprintf "\telse%s" body
211
212 (* generate while statement *)
213 and writeWhileStmt expr stmt =
214   let condString = cppExpr expr
215   and stmtString = cppStmt stmt in
216   sprintf "while (%s)\n%s\n" condString stmtString
217
218 (* generate for statements *)
219 and writeForStmt var init final increment stmt =
220   let varname = cppExpr (expr_of var)
221   and initvalue = cppExpr (expr_of init)
222   and finalvalue = cppExpr (expr_of final)
223   and incrementval = cppExpr (expr_of increment)
224   and stmtbody = cppStmt stmt
225   in
226   sprintf "
227   for (int %s = %s; %s < %s; %s = %s + %s){
228     %s
229   }" varname initvalue varname finalvalue varname varname incrementval stmtbody
230
231 (* generate unary operators *)
232 and writeUnop op expr =
233   let exp = cppExpr (expr_of expr) in
234   let unopFunc op exp = match op with
235   | Ast.Neg -> sprintf "-%s" exp
236   | Ast.Not -> sprintf "!(%s)" exp
237   | Ast.Re -> sprintf "real(%s)" exp (* assumes exp is matrix*)
238   | Ast.Im -> sprintf "imag(%s)" exp
239   | Ast.Norm -> sprintf "%s.norm()" exp
240   | Ast.Trans -> sprintf "%s.transpose()" exp
241   | Ast.Det -> sprintf "%s.determinant()" exp
242   | Ast.Adj -> sprintf "%s.adjoint()" exp
243   | Ast.Conj -> sprintf "%s.conjugate()" exp
244   | Ast.Unit -> sprintf "(%s.conjugate()*%s).isIdentity()" exp exp (* till here

```

```

*)
244 | Ast.Sin    -> sprintf "  sin((double)%s)" exp
      | Ast.Cos    -> sprintf "  cos((double)%s)" exp
246 | Ast.Tan    -> sprintf "  tan((double)%s)" exp
      in unopFunc op exp
248
(* generate binary operations *)
250 and writeBinop expr1 op expr2 =
      let e1 = cppExpr (expr_of expr1)
252   and t1 = type_of expr1
      and e2 = cppExpr (expr_of expr2) in
254   let binopFunc e1 t1 op e2 = match op with
      Ast.Add  -> sprintf "%s + %s" e1 e2
256 | Ast.Sub   -> sprintf "%s - %s" e1 e2
      Ast.Mult -> sprintf "%s * %s" e1 e2
258 | Ast.Div   -> sprintf "%s / %s" e1 e2
      Ast.Mod  -> sprintf "%s %% %s" e1 e2
260 | Ast.Expn  -> sprintf "pow(%s,%s)" e1 e2
      Ast.Tens -> sprintf "tensor(%s, %s)" e1 e2
262 | Ast.Eq    -> equalCaseWise e1 t1 e2
      Ast.Neq  -> sprintf "%s != %s" e1 e2
264 | Ast.Lt    -> sprintf "%s < %s" e1 e2
      Ast.Gt   -> sprintf "%s > %s" e1 e2
266 | Ast.Leq   -> sprintf "%s <= %s" e1 e2
      Ast.Geq  -> sprintf "%s >= %s" e1 e2
268 | Ast.Or    -> sprintf "%s || %s" e1 e2
      Ast.And  -> sprintf "%s && %s" e1 e2
270 | Ast.Xor   -> sprintf "%s ^ %s" e1 e2
      in binopFunc e1 t1 op e2
272
(* generate equality expressions (structural equality is used) *)
274 and equalCaseWise e1 t1 e2 = match t1 with
      Sast.Mat -> sprintf "%s.isApprox(%s)" e1 e2
276 | _ -> sprintf "%s == %s" e1 e2

(* generate matrix *)
and writeMatrix expr_wrap =
280   let matrixStr = List.fold_left (fun a b -> a ^ (writeRow b)) "" expr_wrap in
      let submatrix = String.sub matrixStr 0 ((String.length matrixStr)-1) in
282   sprintf "(Matrix<complex<float>, Dynamic, Dynamic>(%d,%d)<<%s).finished()" (rowMatrix
      expr_wrap) (colMatrix expr_wrap) submatrix

(* generate matrix row *)
and writeRow row_expr =
286   let rowStr = List.fold_left (fun a b -> a ^ (cppExpr (expr_of b)) ^ ",") "" row_expr in
      sprintf "%s" rowStr
288
(* generate column matrix *)
290 and colMatrix expr_wrap =
      List.length (List.hd expr_wrap)
292
(* generate row matrix *)
294 and rowMatrix expr_wrap =
      List.length expr_wrap
296
(* generate function call *)
298 and writeFunCall expr_wrap =
      if expr_wrap == [] then
300   sprintf ""
      else
302   let argvStr = List.fold_left (fun a b -> a ^ (cppExpr (expr_of b)) ^ ",") "" expr_wrap
      in
      let argvStrCom = String.sub argvStr 0 ((String.length argvStr)-1) in
304   sprintf "%s" argvStrCom

```



```

306 (* generate qubits *)
and writeQubit expr bra=
308   (* let exp = string_of_int expr in *)
   sprintf "genQubit(\"%s\",%d)" expr bra

```

B.7 Scripts

B.7.1 Makefile

Makefile

```

1 #Christopher Campbell, Jonathan Wong
  #stuff for compiling cpp files
3 CXX = g++
  CPPDIR = ./cpp
5 INC = $(CPPDIR) ./includes/headers
  INCLUDES =$(INC:%=-I%)
7 CXXFLAGS = -g -Wall $(INCLUDES)

9 OBJS = ast.cmo sast.cmo parser.cmo scanner.cmo analyzer.cmo generator.cmo qlc.cmo

11 .PHONY: default

13 default: qlc cpp/qlang.o

15
  qlc : $(OBJS)
17   ocamlc -g -o qlc $(OBJS)

19 scanner.ml : scanner.mll
   ocamllex scanner.mll

21
  parser.ml parser.mli : parser.mly
23   ocamlyacc parser.mly

25 %.cmo : %.ml
   ocamlc -g -c $<

27
  %.cmi : %.mli
29   ocamlc -g -c $<

31 cpp/qlang.o:
   $(MAKE) -C $(CPPDIR)
33

35 .PHONY : clean
  clean :
37   rm -f qlc parser.ml parser.mli scanner.ml *.cmo *.cmi
   $(MAKE) -C $(CPPDIR) clean
39

  # Generated by ocamldep *.ml *.mli
41 analyzer.cmo: sast.cmo ast.cmo
  analyzer.cmx: sast.cmx ast.cmx
43 generator.cmo: sast.cmo
  generator.cmx: sast.cmx
45 parser.cmo: ast.cmo parser.cmi
  parser.cmx: ast.cm parser.cmi
47 qlc.cmo: scanner.cmo sast.cmo parser.cmi ast.cmo analyzer.cmo
  qlc.cmx: scanner.cmx sast.cmo parser.cmx ast.cmx analyzer.cmx
49 sast.cmo: ast.cmo

```

```

sast.cmx: ast.cmx
51 scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
53 parser.cmi: ast.cmo

```

B.7.2 Compilation script

qlc.ml

```

1 (* Christopher Campbell, Winnie Narang *)
type action = Ast | Sast | Gen | Debug
3
4 let _ =
5   let action =
6     List.assoc Sys.argv.(1) [("-a", Ast); ("-s", Sast); ("-g", Gen); ("-d", Debug);]
7   in
8     let lexbuf = Lex
9       ing.from_channel (open_in Sys.argv.(2)) (*stdin *) and
10       output_file = String.sub Sys.argv.(2) 0 (String.length(Sys.argv.(2))-3) in
11       let program = Parser.program Scanner.token lexbuf in
12       match action with
13       | Ast -> print_string (Ast.string_of_program program)
14       | Sast ->
15         let sprogram =
16           Analyzer.check_program program
17         in
18         print_string (Sast.string_of_sprogram sprogram)
19       | Gen -> Generator.gen_program output_file (Analyzer.check_program program)
20       | Debug -> print_string "debug"

```

B.7.3 Testing script

runTests.sh

```

1 #Christopher Campbell, Winnie Narang
2 #!/bin/bash
3
4 AST=0
5 SAST=0
6 GEN=0
7 COMP=0
8 EXEC=0
9
10
11 if [ $1 == "clean" ]
12 then
13 rm -f ast_error_log sast_error_log gen_error_log comp_error_log ast_log sast_log ast_output
14   sast_output exec_output
15 rm -f SemanticSuccess/*.cpp SemanticSuccess/*.o
16 rm -f SemanticFailure/*.cpp SemanticFailure/*.o
17 rm -f Analyzer/*.cpp Analyzer/*.o
18 else
19
20 if [ $1 == "a" ]
21 then
22 AST=1
23 fi

```

```

23 if [ $1 == "s" ]
24 then
25 SAST=1
26 fi
27 if [ $1 == "g" ] || [ $1 == "c" ] || [ $1 == "e" ]
28 then
29 GEN=1
30 fi
31 if [ $1 == "c" ]
32 then
33 COMP=1
34 fi
35 if [ $1 == "e" ]
36 then
37 EXEC=1
38 fi
39
40 if [ $2 == "ss" ]
41 then
42 files="SemanticSuccess/*.ql"
43 cfiles="SemanticSuccess/*.cpp"
44 elif [ $2 == "sf" ]
45 then
46 files="SemanticFailures/*.ql"
47 cfiles="SemanticFailures/*.cpp"
48 elif [ $2 == "al" ]
49 then
50 files="Algorithms/*.ql"
51 cfiles="Algorithms/*.cpp"
52 fi
53
54 ASTCheck()
55 {
56     eval "../qlc -a $1" 1>> ast_output 2>> ast_error_log
57     wc ast_error_log | awk '{print $1}'
58 }
59
60 SASTCheck()
61 {
62     eval "../qlc -s $1" 1>> sast_output 2>> sast_error_log
63     wc sast_error_log | awk '{print $1}'
64 }
65
66 GenerationCheck()
67 {
68     eval "../qlc -g $1" 2>> gen_error_log
69     wc gen_error_log | awk '{print $1}'
70 }
71
72 CompilationCheck()
73 {
74     eval "g++ -w $1 -I../includes/headers -L../includes/libs -lqlang" 2>> comp_error_log
75     wc comp_error_log | awk '{print $1}'
76 }
77
78 ExecutionCheck()
79 {
80     output=$(eval "../a.out")
81     echo " " >> exec_output
82     echo "Output: " >> exec_output
83     echo "$output" >> exec_output
84     echo " $output"
85 }
86
87 #Check AST

```

```

89 if [ $AST == 1 ]
then
echo "* AST Generation *"
91 rm -f ast_error_log ast_output
errors=0
93 prev_errors=0
for file in $files
95 do
errors=0
97 errors=$(ASTCheck $file)
if [ "$errors" -le "$prev_errors" ]
99 then
count=1
echo "Pass " $file
101 else
103 echo "Fail " $file
fi
105 prev_errors=$errors
done
107 echo ""
fi
109
#Check SAST
111 if [ $SAST == 1 ]
then
echo "* SAST Generation *"
rm -f sast_error_log sast_output
115 errors=0
prev_errors=0
117 for file in $files
do
119 errors=$(SASTCheck $file)
if [ "$errors" -le "$prev_errors" ]
121 then
echo "Pass: " $file
123 else
echo "Fail: " $file
125 fi
prev_errors=$errors
127 done
echo ""
129 fi

131 #Check Generation
if [ $GEN == 1 ]
133 then
cd ../cpp
135 make
cd ../test
137 echo "* Code Generation *"
rm -f gen_error_log
139 errors=0
prev_errors=0
141 for file in $files
do
143 errors=$(GenerationCheck $file)
if [ "$errors" -le "$prev_errors" ]
145 then
echo "Pass: " $file
147 else
echo "Fail: " $file
149 fi
prev_errors=$errors
151 done
echo ""

```

```

153 fi
155 #Check Compilation
156 if [ $COMP == 1 ]
157 then
158     echo "* Compilation *"
159     rm -f comp_error_log
160     errors=0
161     prev_errors=0
162     for file in $cfiles
163     do
164         errors=$(CompilationCheck $file)
165         if [ "$errors" -le "$prev_errors" ]
166         then
167             echo "Pass: " $file
168         else
169             echo "Fail: " $file
170         fi
171         prev_errors=$errors
172     done
173     echo ""
174 fi
175
176 # Execution check
177 if [ $EXEC == 1 ]
178 then
179     echo "* Compilation and Execution *"
180     rm -f comp_error_log exec_output
181     errors=0
182     prev_errors=0
183     exec_output=0
184     for file in $cfiles
185     do
186         errors=$(CompilationCheck $file)
187         if [ "$errors" -le "$prev_errors" ]
188         then
189             echo "Pass (compilation): " $file
190             exec_output=$(ExecutionCheck)
191             if [ "$exec_output" != "0" ]
192             then
193                 echo "Pass (execution): " $file
194                 echo $exec_output
195             else
196                 echo "Fail (execution): " $file
197             fi
198         else
199             echo "Fail (compilation): " $file
200         fi
201         prev_errors=$errors
202     done
203 fi
204
205 fi

```

B.8 Programs

B.8.1 Demo

demo1.ql

```

1 # Sankalpa Khadka
2 def compute() : mat output{
3
4     mat a;
5     mat b;
6     mat c;
7     mat k;
8
9     a = |11>;
10    b = |0>;
11    k = <0|;
12
13    c = a @ b;
14    printq(c);
15
16    c = H*b;
17    printq(c);
18
19    output = b*k;
20 }

```

demo2.ql

```

1 # Sankalpa Khadka
2 def measure(mat top): mat outcome{
3     mat ad;
4
5     ad = adj(top);
6     outcome = top*ad;
7 }
8
9 def outcomezero(mat bottom) : float probability{
10
11     mat top;
12     mat input;
13     mat had;
14     mat cnot;
15     mat ynot;
16     mat output;
17     mat meas;
18
19     top = |0>;
20     input = top @ bottom;
21
22     had = H @ IDT;
23     cnot = [(1,0,0,0)
24             (0,1,0,0)
25             (0,0,0,1)
26             (0,0,1,0)];
27
28
29     ynot = [(1,0,0,0)
30             (0,0,0,C(1.0 I))
31             (0,0,1,0)
32             (0,C(-1. I),0,0)];
33
34     output = (ynot*(cnot*(had*input)));
35
36     printq(output);
37
38     probability = norm(output);

```

```

40 }
42 def compute() : float outcome{
44     mat bottom;
46     bottom = |1>;
47     outcome = outcomezero(bottom);
48     print(outcome);
50     bottom = |0>;
51     outcome = outcomezero(bottom);
52 }

```

demo3.ql

```

1  # Sankalpa Khadka
2  def measure (mat top) : mat outcome{
3
4      mat ad;
5
6      ad = adj(top);
7      outcome = top * ad;
8  }
9
10 def hadamard (int n) : mat gate{
11
12     int i;
13     gate = H;
14
15     for (i from 0 to n-1 by 1){
16         gate = gate @ H;
17     }
18 }
19
20 def topqubit (int n) : mat input{
21
22     int i;
23     input = |0>;
24
25     for (i from 0 to n-1 by 1){
26         input = input @ |0>;
27     }
28 }
29
30 def deutsch (int n, mat U) : float outcomeZero{
31
32     mat bottom; mat top; mat input;
33     mat hadtop; mat meas;
34
35     bottom = |1>;
36     top = topqubit(n);
37     input = top @ bottom;
38
39     hadtop = hadamard(n);
40     input = (hadtop @ H)*input;
41     input = U * input;
42     input = (hadtop @ IDT)*input;
43     meas = measure(top);
44
45     input = (meas @ IDT)* input;
46     outcomeZero = norm(input);
47 }

```

```

49 def compute () : float outcome{
51     int n; mat Ub; mat Uc;
53
54     n = 1;
55     Ub = [(1,0,0,0)(0,1,0,0)(0,0,0,1)(0,0,1,0)];
56     Uc = [(1,0,0,0)(0,1,0,0)(0,0,1,0)(0,0,0,1)];
57
58     outcome = deutsch(n, Ub);
59     print(outcome);
60
61     outcome = deutsch(n, Uc);
62     print(outcome);
63
64     n = 2;
65     Ub = [(1,0,0,0,0,0,0,0)
66            (0,1,0,0,0,0,0,0)
67            (0,0,1,0,0,0,0,0)
68            (0,0,0,1,0,0,0,0)
69            (0,0,0,0,0,1,0,0)
70            (0,0,0,0,1,0,0,0)
71            (0,0,0,0,0,0,0,1)
72            (0,0,0,0,0,0,1,0)];
73
74     outcome = deutsch(n, Ub);
75 }

```

demo4.ql

```

# Sankalpa Khadka
2 def measure (mat top) : mat outcome{
4     mat ad;
6     ad = adj(top);
7     outcome = top * ad;
8 }
10 def ntensor (int n, mat k) : mat gate{
12     int i;
13     gate = k;
14
15     for (i from 0 to n-1 by 1){
16         gate = gate @ k;
17     }
18 }
20 def prepareU (int n) : mat gate {
21     mat i;
22     mat u;
23
24     i = [(1,0)
25          (0,0)];
26
27     u = ntensor(n+1, i);
28     gate = ntensor(n+1,IDT)-2*u;
29 }
30
31 def prepareG (int n) : mat gate{
32     mat s; mat sa; mat i; mat h;

```



```

34     s = ntensor(n,|0>);
35     sa = adj(s);
36     i = ntensor(n,IDT);
37     gate = 2*s*sa - i;
38     h = ntensor(n, H);
39     gate = h*gate*h;
40     gate = gate @ IDT;
41 }
42
43 def grover (int n) : float outcomeZero{
44
45     mat bottom; mat top; mat input;
46     mat hadtop; mat u; mat g; mat go; mat meas;
47     int i;
48
49     bottom = |1>;
50     top = ntensor(n, |0>);
51     input = top @ bottom;
52
53     hadtop = ntensor(n, H);
54     input = (hadtop @ H)*input;
55     u = prepareU(n);
56     g = prepareG(n);
57
58     go = g*u;
59
60     for (i from 0 to n by 1){
61         input = go*input;
62     }
63
64     meas = measure(top);
65     input = (meas @ IDT)* input;
66     outcomeZero = norm(input);
67 }
68
69 def compute () : float outcome{
70     #simulate the grover for f(0)=1
71
72     int n; mat Ub; mat Uc;
73     n = 1;
74
75     outcome = grover(n);
76     print(outcome);
77
78     n = 2;
79     outcome = grover(n);
80 }

```

B.8.2 Successful Test cases

binop_comp_matrix.ql

```

#Winnie Narang
2 def test_func(comp a, comp b, comp c, comp d) : mat ret_val {
3
4     mat x;
5
6     x = [(a,b)(c,d)];

```

```

8   ret_val = [(a,c)(d,b)];
10  ret_val = ret_val * x;
   ret_val = ret_val + x;
12  ret_val = ret_val - x;
   ret_val = ret_val / 2;
14 }

16 def compute() : mat ret_val {
18
   comp a;
20   comp b;
   comp c;
22   comp d;
   mat k;
24
   a = C(4.+5.I);
26   b = C(6.+6.I);
   c = C(7.+8.I);
28   d = C(9.+10.I);

30   ret_val = test_func(a, b, c, d);
   }

```

binop_float_matrix.ql

```

1  #Winnie Narang
   def test_func(float a, float b, float c, float d) : mat ret_val {
3
   mat x;
5
   x = [(a,b)(c,d)];
7
   ret_val = [(a,c)(d,b)];
9
   ret_val = ret_val * x;
11  ret_val = ret_val + x;
   ret_val = ret_val - x;
13  ret_val = ret_val / 2;

15 }

17 def compute() : mat ret_val {
19
   float a;
21   float b;
   float c;
23   float d;

25   a = 3.4;
   b = 6.;
27   c = 5.6;
   d = 100.0;

29   ret_val = test_func(a, b, c, d);
31 }

```

binop_int_arith.ql

```

1 #Winnie Narang
2 def func_test(int z) : int ret_name {
3     int a;
4     int b;
5     int d;
6     a = z;
7     b = 10;
8     d = a+b*a+b/a-b;
9     ret_name=d;
10 }
11 def compute( int a ): int trial {
12     trial = func_test(34);
13 }

```

binop_tensor.ql

```

1 #Jonathan Wong
2 def compute():mat out {
3
4     mat a;
5     mat b;
6     mat c;
7
8     a = [(1)(0)];
9     b = [(0)(1)];
10    c= a@b;
11    print(c);
12 }

```

break_continue.ql

```

1 #Winnie Narang
2 def func_test(int a) : int ret_name {
3
4     int i;
5
6     for(i from 0 to 2 by 1)
7         a=a+5;
8
9     for(i from 2 to 0 by -1)
10    {
11        a=a*10;
12        print(a);
13        break;
14    }
15
16    for(i from 1 to 5)
17    {
18        print(a);
19        continue;
20        a=a*10;
21    }
22
23    ret_name = a;
24 }
25
26 def compute(): int trial {
27
28     trial = func_test(20);
29 }

```

```
30 }
```

builtin_matrix_ops.ql

```
#Sankalpa Khadka
2 def compute(): comp trial {
4     int num_rows;
4     int num_cols;
6     comp val;
6     mat m;
8
10    m = [(1,2,3) (4,5,6) (7,8,9)];
10    num_rows = rows(m);
10    num_cols = cols(m);
12    val = elem(m, 1,2);
14    print(num_rows);
14    print(num_cols);
16
18    trial = val;
}
```

comp_type.ql

```
#Sankalpa Khadka
2 def compute(): comp trial {
4     int num_rows;
4     int num_cols;
6     comp val;
6     mat m;
8
10    m = [(1,2,3) (4,5,6) (7,8,9)];
10    num_rows = rows(m);
10    num_cols = cols(m);
12    val = elem(m, 1,2);
14    print(num_rows);
14    print(num_cols);
16
18    trial = val;
}
```

constants.ql

```
#Jonathan Wong
2 def test_func(int a) : mat ret_val {
4     mat x;
4     mat z;
6     mat y;
6     mat w;
8
10    x = X;
10    z = H;
10    y = Y;
12    w = IDT;
}
```

```

14  print(x);
15  print(z);
16  print(y);
17  print(w);
18
19  ret_val = x * z * y * w;
20 }
21
22 def compute() : mat ret_val {
23
24  ret_val = test_func(0);
25 }

```

empty.ql

```

1  #Christopher Campbell
2  def test_func() : mat ret_val {
3
4      mat x;
5
6      x = [(1,2) (3,4)];
7
8      ret_val = x;
9  }
10
11 def compute() : mat ret_val {
12
13  ret_val = test_func();
14 }

```

float_type.ql

```

1  #Christopher Campbell
2  def func_test(float b) : float ret_name {
3
4      float a;
5      float c;
6
7      a = 5.0;
8      c = a * b;
9
10     ret_name = c;
11 }
12
13 def compute() : float trial {
14
15  trial = func_test(3.7);
16
17 }

```

for_stmt.ql

```

1  #Jonathan Wong
2  def func_test(int z) : int ret_name {
3
4      int i;
5      int a;
6
7      for (i = 0; i < z; i++) {
8          a = i * i;
9      }
10
11  ret_name = a;
12 }

```

```

7      for(i from 0 to 2 by 1)
      a=a+5;
9
11     for(i from 2 to 0 by -1)
12     {
13         a=a*10;
14         print(a);
15     }
16
17     for(i from 1 to 10 by 1)
18     {
19         a=a-3;
20     }
21
22     for(i from 1 to 100){
23         print (a*100);
24     }
25
26     ret_name = 5;
27 }
28
29 def compute(int a): int trial {
30
31     trial = func_test(20);
32 }

```

if_stmt.q1

```

1 #Winnie Narang
2 def func_test(int z) : int ret_name {
3
4     int a;
5
6     # comment before b; just checking for end of comment being correct
7
8     int b;
9     a = 10;
10
11     if(z eq 5) a = 0;
12
13     a = a - 2;
14     if( z leq 5 )
15     {
16         a = 0;
17     }
18     else
19     {
20         a = 10;
21         b = 24;
22     }
23
24     if( a gt 100 )
25     {
26         print(b); # a > 100
27     }
28     else
29     {
30         print(a);
31     }
32
33     ret_name = 8;
34 }

```

mat_add.ql

```
#Sankalpa Khadka
2 def test_func(comp a, comp b, comp c, comp d) : mat ret_val {
4     mat x;
6     x = [(a,b)(c,d)];
8     ret_val = x;
9 }
10
11 def compute():mat trial {
12     comp a;
13     comp b;
14     comp c;
15     comp d;
16     mat k;
17
18     a = C(2.);
19     b = C(2.);
20     c = C(2.);
21     d = C(2.);
22
23     trial = test_func(a, b, c, d)+test_func(a,b,c,d);
24 }
25 }
```

mat_mult.ql

```
1 #Winnie Narang
2 def test_func(comp a, comp b, comp c, comp d) : mat ret_val {
3
4     mat x;
5
6     x = [(a,b)(c,d)];
7
8     ret_val = x;
9 }
10
11 def compute():mat trial {
12     comp a;
13     comp b;
14     comp c;
15     comp d;
16     mat k;
17
18     a = C(2.);
19     b = C(2.);
20     c = C(2.);
21     d = C(2.);
22
23     trial = test_func(a, b, c, d)*test_func(a,b,c,d);
24 }
25 }
```

mat_qubit.ql

```

1 #Winnie Narang
2 def func_test(mat a, mat b) : mat ret_name {
3
4     ret_name = a*b;
5
6 }
7
8
9 def compute(int a):mat trial {
10
11     mat zero;
12     mat one;
13
14     zero = |0>;
15     one = |1>;
16
17     trial = func_test(H,zero);
18     printq(trial);
19
20     trial = func_test(H,one);
21     printq(trial);
22
23 }

```

un_op_det.ql

```

1 #Winnie Narang
2 def func_test(mat z) : mat ret_name {
3     mat a;
4     comp b;
5     a = [(1,9)(4,5)];
6     b = det(a);
7     ret_name = a;
8 }
9 def compute(int a):mat trial {
10     mat x;
11     x = [(1,2)(3,4)];
12     trial = func_test(x);
13 }

```

un_op_trans.ql

```

1 def func_test(mat z) : mat ret_name {
2     mat a;
3     mat b;
4     a=[(1,9,9)(4,5,5)];
5     b = trans(a);
6 }
7 def compute(int a):int trial {
8     trial = 8;
9 }

```

while_stmt.ql

```

1 #Winnie Narang
2 def func_test(int z) : int ret_name {
3     int a;
4     a = 5;

```



```

5
7     #now checking while with comment
8     while(a leq 10)
9         a=a+1;
11
12     while(a neq 1)
13     {
14         # Comment, inside
15         a = (a+1) % 42;
16     }
17
18     ret_name = a;
19
20 }
21 def compute():int trial{
22
23 trial = func_test(5);
24 }

```

B.8.3 Execution output of successful cases

exec_output

```

Output:
2 (-12,76) (-11.5,98)
  (-10,87.5) (-6,114)
4
Output:
6 (21.46,0) (290.2,0)
  (186.8,0) (600,0)
8
Output:
10 364
12
Output:
14 2
16
Output:
18 (0,0)
  (1,0)
  (0,0)
  (0,0)
20
Output:
22 30
  30
24 30
  30
26 30
28
Output:
30 3
  (6,0)
32
Output:
34 (3.52,8.6)
36
Output:

```

```

38 (0,0) (1,0)
   (1,0) (0,0)
   (0.707107,0) (0.707107,0)
40 (0.707107,0) (-0.707107,0)
   (0,0) (-0,-1)
42 (0,1) (0,0)
   (1,0) (0,0)
44 (0,0) (1,0)
   (0,-0.707107) (0,-0.707107)
46 (0,0.707107) (0,-0.707107)

48 Output :
   0
50 1
   0
52
   Output :
54 (1,0) (2,0)
   (3,0) (4,0)
56
   Output :
58 18.5

60 Output :
   3275000
62 3275000
   3275000
64 3275000
   3275000
66 3275000
   3275000
68 3275000
   3275000
70 3275000
   3275000
72 3275000
   3275000
74 3275000
   3275000
76 3275000
   3275000
78 3275000
   3275000
80 3275000
   3275000
82 3275000
   3275000
84 3275000
   3275000
86 3275000
   3275000
88 3275000
   3275000
90 3275000
   3275000
92 3275000
   3275000
94 3275000
   3275000
96 3275000
   3275000
98 3275000
   3275000
100 3275000
    3275000

```

102	3275000
	3275000
104	3275000
	3275000
106	3275000
	3275000
108	3275000
	3275000
110	3275000
	3275000
112	3275000
	3275000
114	3275000
	3275000
116	3275000
	3275000
118	3275000
	3275000
120	3275000
	3275000
122	3275000
	3275000
124	3275000
	3275000
126	3275000
	3275000
128	3275000
	3275000
130	3275000
	3275000
132	3275000
	3275000
134	3275000
	3275000
136	3275000
	3275000
138	3275000
	3275000
140	3275000
	3275000
142	3275000
	3275000
144	3275000
	3275000
146	3275000
	3275000
148	3275000
	3275000
150	3275000
	3275000
152	3275000
	3275000
154	3275000
	3275000
156	3275000
	3275000
158	3275000
	3275000
160	5
162	Output :
	4
164	Output :
166	10

```

168 | 8
170 | Output :
172 | 20
174 | Output :
176 | (4,0) (4,0)
178 | (4,0) (4,0)
180 | Output :
182 | (8,0) (8,0)
184 | (8,0) (8,0)
186 | Output :
188 | (0.707107)|0> + (0.707107)|1>
190 | (0.707107)|0> + (-0.707107)|1>
192 | (0.707107,0)
194 | (-0.707107,0)
196 | Output :
198 | (0,0) (0,0)
200 | (0,0) (0,0)
202 | Output :
204 | (4,5) (6,6)
206 | (7,8) (9,10)
208 | Output :
210 | (-0,-4.5)
212 | Output :
214 | 5
216 | 8
218 | Output :
220 | (0,0) (1,0) (1,0) (0,0)
222 | Output :
224 | <01| + <10|
226 | (0,0) (1,0) (1,0) (0,0)
228 | Output :
230 | (1,-0) (4,-0)
232 | (9,-0) (5,-0)
234 | (9,-0) (5,-0)
236 | Output :
238 | 1
240 | Output :
242 | (1,0) (9,0)
244 | (4,0) (5,0)
246 | Output :
248 | (4.5,0)
250 | (0,4.5)
252 | Output :
254 | -5
256 | Output :
258 | 8

```

```

232 Output:
234 1
236 Output:
238 8
240 Output:
242 8
244 Output:
246 1

```

B.8.4 Failed cases

comp_wrong_decl.ql

```

1 # Winnie Narang
2 def func_test(comp val1, comp val2) : comp ret_name {
3
4     comp val3;
5     val3 = 1;
6
7     ret_name = val1 + val2 * val3;
8 }
9 def compute() : comp ret_name {
10
11     comp comp1;
12     comp comp2;
13
14     if (1) {1; 2+3;} else {3+6;}
15
16     comp1 = C(7.5 I);
17     comp2 = C(3.2 + 1.I);
18
19     ret_name = func_test(comp1, comp2);
20 }

```

func_decl_twice.ql

```

1 # Winnie Narang
2 def func_test1(int z) : int ret_name {
3     int a;
4     int b;
5     int d;
6     a = z;
7     ret_name = z;
8 }
9
10 def func_test1(int z) : int ret_name2 {
11
12     ret_name2 = z;
13 }
14 def compute(int a):int trial {
15
16

```

```

18     trial = func_test1(4);
}

```

if_stmt.ql

```

1  # Winnie Narang
2  def func_test(int z) : int ret_name {
3
4      int a;
5      int b;
6      a = 10;
7
8      else
9      {
10         a = 10;
11         b = 24;
12     }
13 }
14
15 def compute(int a):int trial {
16
17 }

```

invalid_use_binop.ql

```

1  # Winnie Narang
2  def compute() : int ret_name_test
3  {
4      int test_int;
5      ret_name_test = test_int - + test_int;
6
7  }

```

mat_type.ql

```

1  # Winnie Narang
2  def test_func(comp a, comp b, comp c, comp d) : mat ret_val {
3
4      mat x;
5      mat f;
6      x = [(a,b)(c,f)];
7      ret_val = x;
8  }
9
10 def compute() : mat ret_val {
11
12
13     comp a;
14     comp b;
15     comp c;
16     comp d;
17
18     a = C(4.+5.I);
19     b = C(6.+6.I);
20     c = C(7.+8.I);
21     d = C(9.+10.I);

```

```

23   ret_val = test_func(a, b, c, d);
    }

```

mixed_datatypes.ql

```

# Winnie Narang
2 def func_test(int z) : int ret_name {
    int a;
    comp b;
    int d;
    a = z;
    b = C(7.5I);

    d = a+b*a+b/a-b;

    ret_name=d;
12 }
13 def compute( int a ): int trial {
14     trial = func_test(35);
16 }

```

no_compute.ql

```

# Winnie Narang
2 def func_test(float z) : float ret_name {
    float a;
    a = 5.8;

    ret_name = z;
8 }

```

print_stmt.ql

```

# Winnie Narang
2 def func_test(int z) : int ret_name {
    int a;
    a = 5;
    a = z;
    ret_name = a;
6 }
8 def compute(int a):int trial {
10     printq(a);
    }

```

un_op_adj.ql

```

1 # Winnie Narang
2 def func_test(mat z) : mat ret_name {
3     mat a;
4     comp b;
5
6     a = [(1,9,9) (4,5,5)];
7     z = adj(b);

```

```

9 }
11
12 def compute(int a):int trial {
13 }

```

un_op_conj.ql

```

# Winnie Narang
2 def func_test(mat z) : mat ret_name {
    mat a;
    float b;
4
    a = [(1,9,9) (4,5,5)];
6
    b = conj(a);
8
}
10 def compute(int a):int trial {
12 }

```

un_op_cos.ql

```

# Winnie Narang
2 def func_test(int z) : int ret_name {
    int a;
    int b;
4
    a = 90;
    b = cos(a);
6
    comp d;
    d = C(7.5I);
8
    z = cos(d);
    ret_name=b;
10
}
12 def compute(int a):int trial {
14
16 }

```

undec_func_call.ql

```

# Winnie Narang
2 def func_test1(int z) : int ret_name {
    int a;
    int b;
4
    int d;
    a = z;
6
    ret_name = z;
8
}
10 def compute(int a):int trial {
12
    trial = func_test(4);
14 }

```


unmatched_args.ql

```
1 # Winnie Narang
2 def func_test1(int z, int c) : int ret_name {
3     int a;
4     int b;
5     int d;
6     a = z;
7
8     ret_name = z;
9 }
10
11
12 def compute( int a):int trial {
13
14     trial = func_test1(4);
15 }
```

var_undeclared.ql

```
1 # Winnie Narang
2 def compute() : int ret_name_test
3 {
4     int test_int;
5     ret_name_test = test_float;
6
7 }
```

B.8.5 Output for failed cases

test.out

```
1 #generated for test cases under SemanticFailures
2 Fatal error: exception Analyzer.Except("Invalid assignment to variable: val3")
3 Fatal error: exception Analyzer.Except("Invalid function declaration: func_test1 was already
  declared")
4 Fatal error: exception Parsing.Parse_error
5 Fatal error: exception Parsing.Parse_error
6 Fatal error: exception Analyzer.Except("Invalid matrix: incorrect type")
7 Fatal error: exception Analyzer.Except("Invalid assignment to variable: d")
8 Fatal error: exception Analyzer.Except("Missing 'compute' function")
9 Fatal error: exception Analyzer.Except("Invalid function call: incorrect type for parameter
  ")
10 Fatal error: exception Analyzer.Except("Invalid use of unop: 'Adj(expr) '")
11 Fatal error: exception Analyzer.Except("Invalid assignment to variable: b")
12 Fatal error: exception Parsing.Parse_error
13 Fatal error: exception Analyzer.Except("Invalid function call: func_test was not declared")
14 Fatal error: exception Analyzer.Except("Invalid function call: incorrect number of
  parameters")
15 Fatal error: exception Analyzer.Except("Invalid use of a variable: test_float was not
  declared")
```

B.9 C++ Helper files

B.9.1 qlang.cpp

```
1 //Jonathan Wong
2 #include <Eigen/Dense>
3 #include <iostream>
4 #include <complex>
5 #include <string>
6 #include <cmath>
7 #include "qlang.hpp"
8
9 using namespace Eigen;
10 using namespace std;
11
12
13 MatrixXcf tensor(MatrixXcf mat1, MatrixXcf mat2) {
14
15     int mat1rows = mat1.rows();
16     int mat1cols = mat1.cols();
17     int mat2rows = mat2.rows();
18     int mat2cols = mat2.cols();
19
20     MatrixXcf output(mat1rows * mat2rows, mat1cols * mat2cols);
21
22     //iterates through one matrix, multiplying each element with the whole
23     //2nd matrix
24     for(int m = 0; m < mat1rows; m++) {
25         for(int n = 0; n < mat1cols; n++) {
26             output.block(m*mat2rows, n*mat2cols, mat2rows, mat2cols) =
27                 mat1(m,n) * mat2;
28         }
29     }
30
31     return output;
32 }
33
34
35 Matrix4cf control(Matrix2cf mat) {
36     Matrix4cf output;
37     output.topLeftCorner(2,2) = IDT;
38     output.topRightCorner(2,2) = Matrix<complex<float>, 2, 2>::Zero();
39     output.bottomLeftCorner(2,2) = Matrix<complex<float>, 2, 2>::Zero();
40     output.bottomRightCorner(2,2) = mat;
41
42     return output;
43 }
44
45 MatrixXcf genQubit(string s, int bra) {
46
47     int slen = s.length();
48     int qlen = pow(2, slen); //length of vector
49
50     int base10num = 0;
51
52     //iterates through qstr. Whenever digit is a 1, it adds the associated
53     //power of 2 for that position to base10num
54     const char * cq = s.c_str();
55     char * c = new char();
56     for(int i = 0; i < slen; i++) {
57         strncpy(c, cq+i, 1);
58         base10num += strtol(c, NULL, 10) * pow(2, (slen-1-i));
59     }
60     delete c;
61 }
```

```

61 //creates the vector and sets correct bit to 1
62 MatrixXcf qub;
63 if(bra) {
64     qub = MatrixXcf::Zero(1,qlen);
65     qub(0,qlen-1-base10num) = 1;
66 } else if (!bra){
67     qub = MatrixXcf::Zero(qlen,1);
68     qub(base10num,0) = 1;
69 }
70
71     return qub;
72 }
73
74
75 string vectorToBraket(MatrixXcf qub) {
76     int bra;
77     int qlen;
78
79     //determines whether bra or ket
80     if(qub.rows() == 1) { qlen = qub.cols(); bra = 1; }
81     else if (qub.cols() == 1) { qlen = qub.rows(); bra = 0;}
82     else { //prints reg matrix if not row or column vector
83         //cerr << "Incorrect matrix size for vectorToBraket" << endl;
84         //exit(1);
85         ostringstream test;
86         test << qub << endl;
87         return test.str();
88     }
89
90     //gets position of 1 in the qubit
91     complex<float> zero(0,0);
92     int xi = 0;
93     int yi = 0;
94     int number;
95     int index;
96     string result;
97     int count = 0;
98     for(index = 0; index < qlen; index++) {
99         if(bra) { xi = index; }
100        else { yi = index; }
101
102        if(qub(yi,xi) != zero) {
103            // if(bra) { number = qlen-1-index; }
104            // else { number = index; }
105            number = index;
106
107            //converts position to binary number reversed
108            string bin = "";
109            do {
110                if ( (number & 1) == 0 )
111                    bin += "0";
112                else
113                    bin += "1";
114
115                number >>= 1;
116            } while ( number );
117
118            int outQubLen = sqrt(qlen);
119
120            //adds necessary 0s
121            for(int i = bin.length(); i < outQubLen; i++) {
122                bin += "0";
123            }
124
125            reverse(bin.begin(), bin.end()); //reverses

```

```

127     ostream convert;
128     float re = qub(yi, xi).real();
129     float im = qub(yi, xi).imag();
130     string oper = "";
131     string rstr = "";
132     string istr = "";
133
134     //adds constant expression
135     convert << "(";
136     if(re != 0) { convert << re; }
137     if(re != 0 && im != 0) { convert << "+"; }
138     if(im != 0) { convert << im << "i"; }
139     convert << ")";
140
141     //cleans up (1) and (1i) cases
142     string constant = convert.str();
143     if(constant.compare("(1)") == 0) { constant = ""; }
144     else if(constant.compare("(1i)") == 0) { constant = "i"; }
145
146     //generates appropriate bra or ket representation
147     string qubstr;
148     if(bra) { qubstr = constant + "<" + bin + "|"; }
149     else { qubstr = constant + "|" + bin + ">"; }
150
151     if(count > 0) {
152         result += " + " + qubstr;
153     } else { result = qubstr; }
154     count++;
155 }
156 }
157 return result;
158 }

```

B.9.2 qlang.hpp

```

//Jonathan Wong
2 #ifndef QLANG_HPP_
3 #define QLANG_HPP_
4
5 using namespace Eigen;
6 using namespace std;
7
8 //CONSTANTS
9 const Matrix2cf H = (Matrix2cf() << 1/sqrt(2), 1/sqrt(2),
10     1/sqrt(2), -1/sqrt(2)).finished();
11 const Matrix2cf IDT = Matrix2cf::Identity();
12 const Matrix2cf X = (Matrix2cf() << 0, 1, 1, 0).finished();
13 const Matrix2cf Y = (Matrix2cf() << 0, -std::complex<float>(0,1),
14     std::complex<float>(0,1), 0).finished();
15 const Matrix2cf Z = (Matrix2cf() << 1, 0, 0, -1).finished();
16
17 //METHODS
18 MatrixXcf tensor(MatrixXcf mat1, MatrixXcf mat2);
19 Matrix4cf control(Matrix2cf mat);
20 MatrixXcf genQubit(string s, int bra);
21 MatrixXcf genQubits(string s);
22 string vectorToBraket(MatrixXcf qub);
23
24

```

```
#endif
```
