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 Separators .......................................................... 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 Champbell ............................................ 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

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 \{\(|i\rangle\)\}, 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 matrix\(^1\). Analogous to the classical computer which is built from an electrical circuit

\(^1\)That is, a matrix \( U \in \mathbb{C}^{2^n \times 2^n} \) such that \( U^\dagger U = I_{2^n} \), where \( .^\dagger \) 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 know as pure states are represented as \( |i\rangle \) where \( i \) is a \( n \)-bit binary number. For example,

\[
|000\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},
|001\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},
|010\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix},
\ldots,
|101\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},
|110\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},
|111\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}
\]

Similarly, the row vector \( \begin{bmatrix} c_1^* \\ c_2^* \\ \ldots \\ c_n^* \end{bmatrix} \), 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 \( z^* = a - ib \).

- \( A^\dagger \) (Hermitian conjugate (adjoint) of matrix \( A \))
  Defined as \( A^\dagger = (A^\dagger)^* \); if \( A = \begin{bmatrix} 1 \\ 3i \\ 2 + 4i \end{bmatrix} \) then \( A^\dagger = \begin{bmatrix} 1 \\ -3i \\ -2 - 4i \end{bmatrix} \).

- \( |||\psi\rangle|| \) (\( \ell_2 \) 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 $A|\psi\rangle$).
Equivalently\(^2\), 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 $|x\rangle^\otimes n |y\rangle^\otimes m$ is represented as follows:

$$U_f(|x\rangle^\otimes n |y\rangle^\otimes m) = |x\rangle^\otimes n |y \oplus f(x)\rangle^\otimes m,$$

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 transpose 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.

---

\(^2\)Recall 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:

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

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:

```python
def function_name(type1 param1, type2 param2, ..., typek paramk): type returnvar {
    # variable declarations
    # 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:

```python
def foo(mat bar): mat blah {
    int bleh; # OK
    int bluh = 0; # Not OK: parsing error
    comp blih, bloh;
    comp blih; comp bloh; # OK
}
```
As examplified 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 recognize then as of type mat):

```plaintext
mat idt;
mat vct;
mat qub;
qub = [11>];  # this is a ket of dirac notation
qub = <01];  # this is a bra of dirac notation
idt = [(1,0)(0,1)];  # this is a matrix
vct = [(1,2,C(3.2 + 1.1))];  # this is a vector (with complex entries)
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.

```plaintext
int 1;  # Will be used as 'for' loop index
int a;
for( i from 0 to 2 by 1 ) # OK
a=a+5;
for( i from 2 to 0 by -1 ) # OK
{
    a=a*10;
    continue;  # going to next iteration: the next instruction will never be executed.
    print(a);
}
for( i from 1 to 10 ) # OK: missing 'by 1' is implicit
```
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:

```c
if ( predicate )
{
    # Do something
}
else
{
    # Do something else
}
```

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 built-in construct elseif, but instead relies on a nested combination of if and else:

```c
if ( z eq 5 ) a = 0;

a = a - 2;
if ( z leq 5 )
{
    a = 0;
}
else
{
    a = 10;
    b = 24;
}
if ( a gt 100 )
{
    print(b); # a > 100
}
else if ( a eq 10 )
{
    print(a);
}
```
Built-in functions and operators. As shown in the previous two examples, QLang provides
built-in 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 ≥ b`, `a ≤ b`, `a = b` and `a ≠ b`;

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

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

The complete list of these functions, operators and built-in 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 ~ float ~ comp ~ 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 → {0,1}`, promised to be either constant or balanced\(^1\), 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:

\[
\begin{align*}
\ket{0} & \xrightarrow{H^\otimes n} \ket{0} \\
\ket{1} & \xrightarrow{H} \ket{1} \\
\ket{0} & \xrightarrow{H^\otimes n} \ket{0} \\
\ket{1} & \xrightarrow{H} \ket{1} \\
\ket{0} & \xrightarrow{U_f} \ket{0} \\
\ket{1} & \xrightarrow{H^\otimes n} \ket{1} \\
\end{align*}
\]

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:

```python
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 $|x\rangle\langle x|$:

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

(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.

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

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.

```python
def deutsch(int n, qubk top, mat U): float outcomeZero{
    mat bottom; mat top; mat input;
    mat hadtop; mat meas;
    bottom = |1 >;
    top = topqubit(n);
    input = top @ bottom;
    hadtop = hadamard(n);
    input = (hadtop @ H)*input;
    input = U * input;
    input = (hadtop @ IDT)*input;
    meas = measure(top);
    input = (meas @ IDT)* input;
    outcomeZero = norm(input);
}
```

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`:
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 \(C(F)\mid C(F+FI)\mid C(FI)\), 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:

- \textbf{e} the base of natural logarithm \(e = \sum_{k=0}^{\infty} \frac{1}{k!}\). Equivalent to \exp(1); has type \texttt{comp}.
- \textbf{pi} the constant \(\pi\). Has type \texttt{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 \texttt{a-zA-Z}. The language is case-sensitive: \texttt{hullabaloo} and \texttt{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.

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

3.1.6 Expression Operators

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

3.1.7 Separators

Commas are used to separate 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 extend 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 → float
- float → comp
- int → comp

The equality and comparison operators (eq, leq, geq, lt, gt) will perform the implicit conversions above, when they make sense. For instance, \(0 \text{ eq } C(0.0 + 0.0I)\) is valid, and the comparison will be between two complex numbers (after the first operand is converted into a comp). Similarly, \(1 \text{ lt } 2.5\) 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.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Expressions</td>
<td>() [] &lt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Unary</td>
<td>re im norm unit trans det adj conj sin cos tan - not</td>
<td>Right</td>
</tr>
<tr>
<td>Binary</td>
<td>^</td>
<td>* /</td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
<td>Right</td>
</tr>
</tbody>
</table>

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.

```
int a = 3;
int b = 12;
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.

```
mat b_qubit = <0100>;
```
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 <expression> = expression>, except that this notation mimics the 'ket' of 'bra-ket' notation and can therefore be though of as a column vector representation of the given qubit. An example "ket-" qubit is shown below.

```c
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 \textit{expression}

The result is the adjoint of the \textit{expression}. The type of the expression must be \textit{mat}. The result has the same type as the \textit{expression}.

\textbf{conj} \textit{expression}

The result is the complex conjugate of the \textit{expression}. The type of the expression must be \textit{comp} or \textit{mat}. The result has the same type as the \textit{expression}.

\textbf{sin} \textit{expression}

The result is the evaluation of the trigonometric function sine on the \textit{expression}. The type of the expression must be \textit{int}, \textit{float}, or \textit{comp}. The result has type \textit{float} if the expression is of type \textit{int} or \textit{float} and type \textit{comp} if the expression is of type \textit{comp}.

\textbf{cos} \textit{expression}

The result is the evaluation of the trigonometric function cosine on the \textit{expression}. The type of the expression must be \textit{int}, \textit{float}, or \textit{comp}. The result has type \textit{float} if the expression is of type \textit{int} or \textit{float} and type \textit{comp} if the expression is of type \textit{comp}. (If an error occurred because of a division by zero, a runtime exception is raised.)

3.4.5 Binary Operators

\textit{expression} \textbf{^} \textit{expression}

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

\textit{expression} \textbf{*} \textit{expression}

The result is the product of the \textit{expressions}. The type considerations are the same as they are for \textit{expression} \textbf{^} \textit{expression} except that it also allows for matrices.

\textit{expression} \textbf{/} \textit{expression}

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

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).

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

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).

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.

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.

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 \textit{expression} is greater than or equal to the second. The type of the expressions must be int or float.

expression \textbf{or} expression

The result is a 1 or 0 indicating the logical \textit{or} of the \textit{expressions}. The type of the expressions must be int or float and must be the same. In the \textit{expressions}, 0 is considered false and all other values are considered true.

expression \textbf{and} expression

The result is a 1 or 0 indicating the logical \textit{and} of the \textit{expressions}. The type considerations are the same as they are for \textit{expression \textbf{or} expression}.

expression \textbf{xor} expression

The result is a 1 or 0 indicating the logical \textit{xor} of the \textit{expressions}. The type considerations are the same as they are for \textit{expression \textbf{or} expression}.

3.4.6 Assignment Operators

Assignment operators have left associativity

\textbf{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

\begin{verbatim}
description:
  type-specifier declarator-list
\end{verbatim}

3.5.1 Type Specifiers

There are five main type specifiers:

\begin{verbatim}
type-specifier:
  int
  float
  comp
  mat
\end{verbatim}
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,

\[ x = a + 3; \]
is a valid expression statement, and so is

```plaintext
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.

```plaintext
if ( condition ){
} else {
}
```

Example:

```plaintext
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:

```plaintext
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:

```
while ( k leq 100 ) {
  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.

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

3.8 Constant expressions

In order to facilitate efficiency in writing expression, the language introduces various mathematical constants such as $\pi$, $e$ 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 illustrate the use of Qlang in solving quantum computing problems.

#### 3.9.1 Solving Quantum Computation Problem

**Problem 1**

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

```python
2 def compute() : mat evaluate (){
    mat a;
3     a = |00 >;
4     evaluate = (H @ X) * a;
5     printq (evaluate);
}
```

**Problem 2**

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

![Quantum Circuit](image)

```python
1 def measure(mat top) : mat outcome{
2     mat ad;
3         ad = adj(top);  
4         outcome = top*ad;
5 }
6
def outcomezero(mat bottom) : float probability{
7     mat top; mat input;
8     mat had; mat cnot; mat ynot;
9 }
```

Figure 3.1: Quantum Circuit
3.9.2 Simulation of Quantum Algorithm

Deutsch Jozsa Algorithm

```python
def measure(mat top): mat outcome{
    # returns the measurement matrix for top qubit
    mat ad;
    ad = adj(top);
    outcome = top * ad;
}
def hadamard(int n): mat gate{
    # returns Hadamard gate for n qubit system
    int i;
    gate = H;
    for (i from 0 to n-1 by 1){
        for (j from 0 to n-1 by 1){
            gate[i][j] = 1 / sqrt(2) * (1 if i == j else 0);
        }
    }
    return gate;
}
```

Output

\[
(0.707107)|10\rangle + (-0.707107)|11\rangle
\]
\begin{verbatim}
 gate = gate @ H;

 def topqubit (int n) : mat input{
 # returns zero qubit for n qubit system
     int i;
     input = |0>;
     for ( i from 0 to n-1 by 1) {
         input = input @ |0>;
     }
 }

 def deutsch (int n, mat U) : float outcomeZero{
 # series of unitary transformation followed by measurement
     mat bottom; mat top; mat input;
     mat hadtop; mat meas;
     bottom = |1>;
     top = topqubit(n);
     input = top @ bottom;
     hadtop = hadamard(n);
     input = (hadtop @ H)*input;
     input = U*input;
     input = (hadtop @ IDT)*input;
     meas = measure(top);
     input = (meas @ IDT)*input;
     outcomeZero = norm(input);
 }

 def compute () : float outcome{
     int n; mat Ub; mat Uc;
     # test for n equals 1
     n = 1;
     # Ub is balanced, Uc is constant
     Ub = [(1,0,0,0) (0,1,0,0) (0,0,0,1) (0,0,1,0)];
     Uc = [(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)];
     outcome = deutsch(n, Ub);
     print(outcome);
     outcome = deutsch(n, Uc);
     print(outcome);
     
     # test for n equals 2
     n = 2;
     Ub = [(1,0,0,0,0,0,0,0)
           (0,1,0,0,0,0,0,0)
           (0,0,1,0,0,0,0,0)
           (0,0,0,1,0,0,0,0)
           (0,0,0,0,1,0,0,0)
           (0,0,0,0,0,1,0,0)
           (0,0,0,0,0,0,1,0)
           (0,0,0,0,0,0,0,1)];
     outcome = deutsch(n, Ub);
 }
\end{verbatim}
Grover’s Search Algorithm

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

```
def measure (mat top) : mat outcome{
    # measurement matrix for top qubit
    mat ad;
    ad = adj(top);
    outcome = top * ad;
}
def ntensor (int n, mat k) : mat gate{
    # return n qubit k
    int i;
    gate = k;
    for (i from 0 to n-1 by 1){
        gate = gate @ k;
    }
}
def prepareU (int n) : mat gate {
    # prepare the Uw or grover oracle
    mat i;
    mat u;
    i = [(1,0)
         (0,0)];
    u = ntensor(n+1, i);
    gate = ntensor(n+1,IDT)-2*u;
}
def prepareG (int n) : mat gate{
    # prepare grover defusive operator
```

Figure 3.2: Grover Algorithm Circuit
mat s; mat sa; mat i; mat h;
s = ntensor(n,|0>);
sa = adj(s);
i = ntensor(n,IDT);
gate = 2*sa - i;
h = ntensor(n, H);
gate = h*gate*h;
gate = gate @ IDT;
}
def grover (int n) : float outcomeZero{
    mat bottom; mat top; mat input;
    mat hadtop; mat u; mat g; mat go; mat meas;
    int i;
    bottom = |1>;
top = ntensor(n, |0>);
    input = top @ bottom;
    hadtop = ntensor(n, H);
    input = (hadtop @ H)*input;
    u = prepareU(n);
g = prepareG(n);
    # grover operator
go = g*u;
    # apply grover operator over iteration
    for (i from 0 to n by 1){
        input = go*input;
    }
    # measure on top qubit
    meas = measure(top);
    input = (meas @ IDT)* input;
    # likelihood to get 0 on top register
    outcomeZero = norm(input);
}
def compute () : float outcome{
    #simulate the grover for f(0)=1
    int n; mat Ub; mat Uc;
n = 1;
    outcome = grover(n);
    print(outcome);
    n = 2;
    outcome = grover(n);
}

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 functions with no params working
2014-12-17: khadka - demo 1 added
2014-12-17: 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
2014-12-13: Winnie Narang - Refined failures folder and added powerpoint
2014-12-13: khadka - row and column
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: 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.
2016-12-12: 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 - 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 -> vectorToBraket
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 tutorial, improved syntax for the code
  highlighting.
2014-12-11: Clement Canonne - Started fixing deutsch.ql, 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 .ql
  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
2014-12-03: 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 unecessary 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: 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: Jonathan Wong - Made initial fixes to grover search
2014-11-23: Jonathan Wong - Added control, updated problems, more examples
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 ananalyzer, 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 and 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 compete 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: 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 - First round of changes: (lexical conventions updated soon).
2014-10-26: khadka - main
2014-10-26: jonathanwong - turned off colour and notes in main.tex
2014-10-26: khadka - troubleshooting
2014-10-26: khadka - grover circuit
2014-10-26: khadka - new examples
2014-10-26: Christopher Campbell - Finished sec-expressions.text
2014-10-26: khadka - main
2014-10-26: jonathanwong - Fixed sec-declarations.tex
2014-10-26: khadka - Merge branch 'master' of https://github.com/thejonathanwong/PLT
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 listing 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
2014-10-24: khadka - added miicrc to resource
2014-10-24: Christopher Campbell - example section
2014-10-20: Christopher Campbell - Made small changes to the scanner and added package and preamble files
2014-10-15: Clement Canonne - Filled lexical conventions, added packages and preamble files.
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 - Started scanner
2014-10-13: Christopher Campbell - Added folder and documents for our compiler
2014-10-13: Christopher Campbell - Adding miicrc to resources
2014-10-13: Sankalpa Khadka - starter ast
2014-10-12: khadka - Project proposal
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 ocamlex - 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 \textquote{g} option, runTests.sh will perform the tasks under \textquote{a}, \textquote{s} and then the operations specific to \textquote{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:

\texttt{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.ql is a QLang program as follows:

42
def func_test(int a) : int ret_name {
    int i;
    for (i from 0 to 2 by 1)
        a=a+5;
    for (i from 2 to 0 by -1)
        { a=a*10; print(a); break; }    
    for (i from 1 to 5)
    {  print(a); continue;  a=a*10;  }
    ret_name = a;
} 

def compute() : int trial {
    trial = func_test(20);
}

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

#include <iostream>
#include <complex>
#include <cmath>
#include <Eigen/Dense>
#include <qlang>
using namespace Eigen;
using namespace std;

int func_test (int a)
{
    int i;
    int ret_name;
    for (int i = 0; i < 2; i = i + 1){
        a = a + 5;
    }
    for (int i = 2; i < 0; i = i + -1)
    {  a = a * 10;  cout << a << endl;  break;  }
}
```cpp
for (int i = 1; i < 5; i = i + 1) {
    cout << a << endl;
    continue;
    a = a * 10;
}
ret_name = a;
return ret_name;
}
int main ()
{
    int trial;
    trial = func_test(20);
    std::cout << trial << endl;
    return 0;
}
```

and the generated output of this is:

```cpp
Another example we consider is mat_qubit.ql
```

```cpp
def func_test(mat a, mat b): mat ret_name {
    ret_name = a*b;
}
def compute(int a):mat trial {
    mat zero;
    mat one;
    zero = |0>;
    one = |1>;
    trial = func_test(H,zero);
    printq(trial);
    trial = func_test(H,one);
    printq(trial);
}
```
It generates mat_qubit.cpp as below:

```cpp
#include <iostream>
#include <complex>
#include <cmath>
#include <Eigen/Dense>
#include <qlang>

using namespace Eigen;
using namespace std;

MatrixXcf func_test (MatrixXcf a, MatrixXcf b)
{
    MatrixXcf ret_name;
    ret_name = a * b;
    return ret_name;
}

int main()
{
    MatrixXcf zero;
    MatrixXcf one;
    MatrixXcf trial;

    zero = genQubit("0", 0);
    one = genQubit("1", 0);
    trial = func_test(H, zero);
    cout << vectorToBraket(trial) << endl;
    trial = func_test(H, one);
    cout << vectorToBraket(trial) << endl;
    std::cout << trial << endl;
    return 0;
}
```

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

```
(0.707107)|0> + (0.707107)|1>
(0.707107)|0> + (-0.707107)|1>
(0.707107,0)
(-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.

```py
def measure(mat top): mat outcome{
    mat ad;
    ad = adj(top);
    outcome = top * ad;
}

def hadamard(int n): mat gate{
    int i;
    gate = H;
    for (i from 0 to n-1 by 1){
```
gate = gate @ H;


def topqubit (int n) : mat input{
    int i;
    input = |0>;
    for (i from 0 to n-1 by 1){
        input = input @ |0>;
    }
}

def deutsch (int n, mat U) : float outcomeZero{
    mat bottom; mat top; mat input;
    mat hadtop; mat meas;
    bottom = |1>;
    top = topqubit(n);
    input = top @ bottom;
    hadtop = hadamard(n);
    input = (hadtop @ H)*input;
    input = U * input;
    input = (hadtop @ IDT)*input;
    meas = measure(top);
    input = (meas @ IDT)* input;
    outcomeZero = norm(input);
}

def compute () : float outcome{
    int n; mat Ub; mat Uc;
    n = 1;
    Ub = [ (1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1) ];
    Uc = [ (1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1) ];
    outcome = deutsch(n, Ub);
    print(outcome);
    outcome = deutsch(n, Uc);
    print(outcome);
    n = 2;
    Ub = [ (1,0,0,0,0,0,0,0,0)
           (0,1,0,0,0,0,0,0,0)
           (0,0,1,0,0,0,0,0,0)
           (0,0,0,1,0,0,0,0,0)
           (0,0,0,0,1,0,0,0,0)
           (0,0,0,0,0,1,0,0,0)
           (0,0,0,0,0,0,1,0,0)
           (0,0,0,0,0,0,0,1,0) ];
    outcome = deutsch(n, Ub);
}

It creates the C++ code as follows:
```cpp
#include <iostream>
#include <complex>
#include <cmath>
#include <Eigen/Dense>
#include <qlang>
using namespace Eigen;
using namespace std;

MatrixXcf measure (MatrixXcf top)
{
    MatrixXcf ad;
    MatrixXcf outcome;
    ad = top.adjoint();
    outcome = top * ad;
    return outcome;
}

MatrixXcf hadamard (int n)
{
    int i;
    MatrixXcf gate;
    gate = H;
    for (int i = 0; i < n - 1; i = i + 1){
        gate = tensor(gate, H);
    }
    return gate;
}

MatrixXcf topqubit (int n)
{
    int i;
    MatrixXcf input;
    input = genQubit("0", 0);
    for (int i = 0; i < n - 1; i = i + 1){
        input = tensor(input, genQubit("0", 0));
    }
    return input;
}

float deutsch (int n, MatrixXcf U)
{
    MatrixXcf bottom;
    MatrixXcf top;
    MatrixXcf input;
    MatrixXcf hadtop;
    MatrixXcf meas;
    float outcomeZero;
    bottom = genQubit("1", 0);
    top = topqubit(n);
    input = tensor(top, bottom);
```
hadtop = hadamard(n);
input = tensor(hadtop, H) * input;
input = U * input;
input = tensor(hadtop, IDT) * input;
meas = measure(top);
input = tensor(meas, IDT) * input;
outcomeZero = input.norm();

return outcomeZero;

} int main ()
{
 int n;
 MatrixXcf Ub;
 MatrixXcf Uc;
 float outcome;

 n = 1;
 Ub = ( Matrix<complex<float>, Dynamic, Dynamic>(4,4) << 1,0,0,0,0,1,0,0,0,0,0,1,0,0,1,0).
 finished();
 Uc = ( Matrix<complex<float>, Dynamic, Dynamic>(4,4) << 1,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,1).
 finished();
 outcome = deutsch(n,Ub);
 cout << outcome << endl << endl;
 outcome = deutsch(n,Uc);
 cout << outcome << endl << endl;
 n = 2;
 Ub = ( Matrix<complex<float>, Dynamic, Dynamic>(8,8)
 << 1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0).
 finished();
 outcome = deutsch(n,Ub);
 std::cout << outcome << endl;
 return 0;
}

The output of this execution is:

0
2
1
4
0

Following programs show the ability of the semantic analyzer to catch incorrect programs. For instance, the program:

def func_test1(int z) : int ret_name {
    int a;
    int b;
    int d;
    a = z;
    ret_name = z;
}
def func_test1(int z) : int ret_name2 {
    ret_name2 = z;

```python
def compute(a):
    trial = func_test1(4);
```
gives the error:

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

whereas the sample program

```python
def func_test(z):
    float a;
    a = 5.8;
    ret_name = z;
```

would give the error:

```
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 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 concensus 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.

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

- $\text{CNOT}|00⟩ = |00⟩$
- $\text{CNOT}|01⟩ = |01⟩$
- $\text{CNOT}|10⟩ = |11⟩$
- $\text{CNOT}|11⟩ = |10⟩$.

A.2 Tensor product and its properties

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

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

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

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

Source Code

B.1 Scanner

scanner.mll

(* Christopher Campbell, Winnie Narang *)
{ open Parser }

let whitespace = [ ' ' 't' 'r' 'n' ]

let name = [ 'a' 'z' 'A' 'Z' ] [ 'a' 'z' 'A' 'Z' '0' '9' '_'] *

let integers = [ '0' '9'] +

let floats = [ '0' '9'] + ' . ' [ '0' '9'] *

rule token = parse
whitespace { token lexbuf }

| '#' { comment lexbuf }

| 'int' { INT } (* Integer type *)

| 'float' { FLOAT } (* Float type *)

| 'comp' { COMP } (* Complex type *)

| 'mat' { MAT } (* Matrix *)

| 'C' { C } (* Start of complex number *)

| 'I' { I } (* Imaginary component *)

| 'def' { DEF } (* Define function *)

| '=' { ASSIGN } (* Assignment *)

| ', ' { COMMA } (* Separate list elements *)

| ':' { COLON } (* Separate matrix rows *)

| ';' { SEMI } (* Separate matrix columns *)

| '(' { LPAR } (* Surround expression *)

| ')' { RPAR } (* )

| '[' { LBRACK } (* Surround vectors/matrices *)

| ']' { RBRACK } (* )

| '{' { LBRACE } (* Surround blocks *)

| '}' { RBACE } (* )

| '<' { LCAR } (* Open bra- *)

| '>' { RCAR } (* Close -ket *)

| '|' { BAR } (* Close bra- and Open -ket *)

| '+' { PLUS } (* Addition *)

| '-' { MINUS } (* Subtraction *)

| '*' { TIMES } (* Multiplication *)

| '/' { DIV } (* Division *)

| '%' { MOD } (* Modulus *)
B.2 Parser

```
\[\text{\texttt{if}} \ \text{\texttt{else}} \ \text{\texttt{for}} \ \text{\texttt{from}} \ \text{\texttt{to}} \ \text{\texttt{by}} \ \text{\texttt{while}} \ \text{\texttt{break}} \ \text{\texttt{continue}} \ \text{\texttt{name}} \ \text{\texttt{integers}} \ \text{\texttt{floats}} \ \text{\texttt{eof}} \ \text{\texttt{and}} \ \text{\texttt{comment}} = \text{\texttt{parse}}\]
```

```
\texttt{parser.mly}
\begin{verbatim}
\%
\texttt{(Christopher Campbell, Sankalpa Khadka*)}
\%
\texttt{open Ast}
\%
\texttt{\texttt{C I INT FLOAT COMP MAT DEF ASSIGN \texttt{COMMA COLON SEMI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE LCAR RCAR BAR}}}
\end{verbatim}
```

55
%token PLUS MINUS TIMES DIV MOD EXPN
%token EQ NEQ LT GT LEQ GEQ
%token NOT AND OR XOR
%token TENS UNIT NORM TRANS DET ADJ CONJ IM RE SIN COS TAN
%token IF ELIF ELSE FOR FROM TO BY WHILE BREAK CONT
%token EOF
%token <string> ID
%token <string> INT_LIT
%token <float> FLOAT_LIT
%token <string> COMP_LIT
%token <string> ID
%token <string> INT { Int }
%token <string> FLOAT { Float }
%token <string> COMP { Comp }
%token <string> MAT { Mat }
%token <string> NOELSE
%token ELSE
%token ASSIGN
%token OR XOR
%token AND
%token NOT
%token EQ NEQ
%token LT GT LEQ GEQ
%token PLUS MINUS
%token TIMES DIV MOD TENS
%token EXPN
%token RE IM NORM TRANS DET ADJ CONJ UNIT SIN COS TAN

%dstart program
%type <Ast.program> program

vtype:
    INT { Int } | FLOAT { Float } | COMP { Comp } | MAT { Mat }
vdecl:
    vtype ID SEMI { { typ = $1; name = $2 } }
vdecl_list:
    /* nothing */ { [] } | vdecl_list vdecl { $2 ::: $1 }
formal_params:
    /* nothing */ { [] } | formal_params_list { List.rev $1 }
formal_params_list:
    vtype ID { [{ typ = $1; name = $2; }] } | formal_params_list COMMA vtype ID { { typ = $3; name = $4; } ::: $1 }
actual_params:
    /* nothing */ { [] } | actual_params_list { List.rev $1 }
actual_params_list:
    expr { [$1] } | actual_params_list COMMA expr { $3 ::: $1 }
fdcl:
    DEF ID LPAREN formal_params RPAREN COLON vtype ID LBRACE vdecl_list stmt_list RBRACE
    { { func_name = $2; formal_params = $4; ret_typ = $7; } }
ret_name = $8;
lodals = List.rev $10;
body = List.rev $11; }

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

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

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

expr:
| ID { Id($1) }
| INT_LIT { Lit_int(int_of_string $1) }
| FLOAT_LIT { Lit_float($1) }
| C LPAREN inner_comp RPAREN { Lit_comp(List.hd $3, List.hd (List.rev $3)) }
| LCAR INT_LIT BAR { Lit_qub($2, 0) }
| BAR INT_LIT RCAR { Lit_qub($2, 1) }
| LBRACK mat_row_list RBRACK { Mat(List.rev($2)) }
| LPAREN expr RPAREN { $2 }
| ID ASSIGN expr { Assign($1, $3) }
| ID LPAREN actual_params RPAREN { Call($1, $3) }
| MINUS expr { Unop(Neg, $2) }
| RE LPAREN expr RPAREN { Unop(Re, $3) }
| IM LPAREN expr RPAREN { Unop(Im, $3) }
| NORM LPAREN expr RPAREN { Unop(Norm, $3) }
| TRANS LPAREN expr RPAREN { Unop(Trans, $3) }
| DET LPAREN expr RPAREN { Unop(Det, $3) }
| ADJ LPAREN expr RPAREN { Unop(Adj, $3) }
| UNIT LPAREN expr RPAREN { Unop(Unit, $3) }
| SIN LPAREN expr RPAREN { Unop(Sin, $3) }
| COS LPAREN expr RPAREN { Unop(Cos, $3) }
| TAN LPAREN expr RPAREN { Unop(Tan, $3) }
| expr PLUS expr { Binop($1, Add, $3) }
| expr MINUS expr { Binop($1, Sub, $3) }
| expr TIMES expr { Binop($1, Mult, $3) }
| expr DIV expr { Binop($1, Div, $3) }
| expr MOD expr { Binop($1, Mod, $3) }
| expr EXPN expr { Binop($1, Expn, $3) }
| expr TENS expr { Binop($1, Tens, $3) }
| expr EQ expr { Binop($1, Eq, $3) }
| expr NEQ expr { Binop($1, Neq, $3) }
| expr LT expr { Binop($1, Lt, $3) }
| expr GT expr { Binop($1, Gt, $3) }
| expr LEQ expr { Binop($1, Leq, $3) }
| expr GEQ expr { Binop($1, Geq, $3) }
| expr OR expr { Binop($1, Or, $3) }
| expr AND expr { Binop($1, And, $3) }
| expr XOR expr { Binop($1, Xor, $3) }

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

stmt:
| expr SEMI { Expr($1) }
| LBRACE stmt_list RBRACE { Block(List.rev $2) }
B.3 AST

ast.ml

(* Christopher Campbell, Winnie Narang *)
(* Elementary Data Types *)
type data_type =
  | Int
  | Float
  | Comp
  | Mat

(* Unary Operators *)
type un_op =
  | Neg
  | Not
  | Re
  | Im
  | Norm
  | Trans
  | Det
  | Adj
  | Conj
  | Unit
  | Sin
  | Cos
  | Tan

(* Binary Operators *)
type bi_op =
  | Add
  | Sub
  | Mult
  | Div
  | Mod
  | Expn
  | Tens
  | Eq
  | Neq
  | Lt
  | Gt
  | Leq
```
(* Expressions *)
type expr =
  Lit_int of int
| Lit_float of float
| Lit_comp of float * float
| Lit_qub of string * int
| Mat of expr list list
| Id of string
| Unop of un_op * expr
| Binop of expr * bi_op * expr
| Assign of string * expr
| Call of string * expr list
| Noexpr

(* Statements *)
type stmt =
  Expr of expr
| Block of stmt list
| If of expr * stmt * stmt
| For of expr * expr * expr * expr * stmt
| While of expr * stmt
| BreakCont of int

(* Statement Lists *)
type stmt_list =
  stmt list

(* Variables Declaration *)
type var_decl =
  {
    typ : data_type;
    name : string;
  }

(* Function Declaration *)
type func_decl =
  {
    ret_typ : data_type;
    ret_name : string;
    func_name : string;
    formal_params : var_decl list;
    locals : var_decl list;
    body : stmt list;
  }

(* Program *)
type program =
  func_decl list

(* Pretty Printer *)
let rec string_of_expr = function
  Lit_int(n) -> string_of_int n
| Lit_float(n) -> string_of_float n
| Lit_comp(f1,f2) -> string_of_float f1 ^ * + * ^ string_of_float f2 ^ 'i'
| Lit_qub(s,t) -> let typ = string_of_int t in (match typ with
  '0' -> 'Qub-bra of '^ s
| _ -> 'Qub-ket of '^ s)
| Mat(l) -> string_of_mat l
| Id(s) -> s
| Unop(un1,expl) ->
```
105  (match un1 with
106      Neg -> '−'
107    | Not -> '!' | Re -> 'Re'
108      Im -> 'Im'
109    | Norm -> 'Norm '
110    | Trans -> 'Trans '
111    | Det -> 'Det'
112    | Adj -> 'Adj'
113    | Conj -> 'Conj'
114    | Unit -> 'Unit'
115    | Sin -> 'Sin'
116    | Cos -> 'Cos'
117    | Tan -> 'Tan') ^ string_of_expr ex1
118
119  | Binop(ex1, binop, ex2) -> string_of_expr ex1 ^
120     (match binop with
121        Add -> ' + ' | Sub -> '−' | Mult -> '∗'
122        Div -> ' / ' | Mod -> '%' | Expn -> ' ^ ' | Tens -> '@'
123        Eq->'=='* | Neq->'!=' | Lt -> '<'
124        Leq -> '<= '* | Gt -> '> '
125        Xor -> ' XOR ' | And -> ' && ' | Or -> ' || '
126     ) ^ string_of_expr ex2
127  | Assign(str, expr) -> s t r ^ ' = ' ^ string_of_expr expr
128  | Call(str, expr_list) -> 'Calling ' ^ str ^ ' on ' ^ string_of_exprs expr_list
129  | Noexpr -> ''
130
131  and string_of_mat l =
132    let row_strs =
133       List.map string_of_row l
134    in
135        '[ ' ^ String.concat '' row_strs ^ ' ]'
136
137  and string_of_row r =
138    let row_str =
139       String.concat ', ' (List.map string_of_expr r)
140    in
141        '( ' ^ row_str ^ ' )'
142
143  and string_of_exprs exprs =
144       String.concat '
 ' (List.map string_of_expr exprs)
145
146  and string_of_stmt = function
147     Expr (exp) -> string_of_expr exp ^ '
'
148    | Block(stmt_list) -> '{
149        string_of_stmts stmt_list ^ '
'
150    | If (e, s, Block([])) -> 'if (' ^ string_of_expr e ^ ')
151        string_of_expr s
152    | If (e, s1, s2) -> 'if (' ^ string_of_expr e ^ ')
153        string_of_stmt s1 ^ 'else'
154        string_of_stmt s2
155    | For(ex1, ex2, ex3, ex4, stmt) -> 'For args :
156        string_of_expr ex1 ^ ' ' ^ string_of_expr ex2
157        string_of_expr ex3 ^ ' ' ^ string_of_expr ex4
158        ' \nstatement :
159    | While(expr, stmt) -> 'While condition :
160        string_of_expr expr ^ ' \nstatement :'
161    | BreakCont(t) -> string_of_breakcont t
162
163  and string_of_breakcont t =
164    if (t = 0) then
165        'break'
166    else
167        'continue'
168
169  and string_of_stmts stmts =
170       String.concat '
 ' (List.map string_of_stmt stmts)
171
172  and string_of_var_decl varDecl =
(match var_decl.typ with
  | Int -> "int", "name": var_decl.name
  | Float -> "float", "name": var_decl.name
  | Comp -> "comp", "name": var_decl.name
  | Mat -> "mat", "name": var_decl.name)

and string_of_fdecl fdecl =
  (match fdecl.ret_typ with
    | Int -> "int"
    | Float -> "float"
    | Comp -> "comp"
    | Mat -> "mat"
    | "\nret_name": fdecl.ret_name
    | "\nfunc_name": fdecl.func_name
    | "\n(* "
    | String.concat " (List.map string_of_var_decl fdecl.formal_params) " /* 
    | String.concat " (List.map string_of_var_decl fdecl.locals) " /* 
    | String.concat " (List.map string_of_stmt fdecl.body) " /*

and string_of_program (funcs) =
  "program:" " String.concat " "n" (List.map string_of_fdecl funcs)

B.4 Analyzer

 analyzer.ml

 let builtin_vars =
  [ { styp = Sast.Float; sname = 'e'; builtinv = true; };
    { styp = Sast.Float; sname = 'pi'; builtinv = true; };
    { styp = Sast.Float; sname = 'X'; builtinv = true; };
    { styp = Sast.Float; sname = 'Y'; builtinv = true; };
    { styp = Sast.Float; sname = 'Z'; builtinv = true; };
    { styp = Sast.Float; sname = 'H'; builtinv = true; };
    { styp = Sast.Float; sname = 'IDT'; builtinv = true; };
  ]

 let builtin_funcs =
  [ { sret_typ = Sast.Void;
      sret_name = "null";
      sfunc_name = 'print';
      sformal_params = [{ styp = Sast.Poly; sname = 'print_val'; builtinv = true; }];
  ];
slocals = [];
sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
builtinf = true;

{sret_typ = Sast.Void;
sret_name = "null";
sfunc_name = "printq";
sformal_params = [{syp = Sast.Mat; sname = "printq_val"; builtinv = true}];
slocals = [];
sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
builtinf = true;
}

{sret_typ = Sast.Int;
sret_name = "null";
sfunc_name = "rows";
sformal_params = [{syp = Sast.Mat; sname = "rows_val"; builtinv = true}];
slocals = [];
sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
builtinf = true;
}

{sret_typ = Sast.Int;
sret_name = "null";
sfunc_name = "cols";
sformal_params = [{syp = Sast.Mat; sname = "rows_val"; builtinv = true}];
slocals = [];
sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
builtinf = true;
}

{sret_typ = Sast.Comp;
sret_name = "null";
sfunc_name = "elem";
sformal_params = [{syp = Sast.Mat; sname = "elem_mat"; builtinv = true}];
{syp = Sast.Int; sname = "elem_row"; builtinv = true}];
{syp = Sast.Int; sname = "elem_col"; builtinv = true}];
slocals = [];
sbody = [Sast.Sexpr(Sast.Expr(Sast.Noexpr, Sast.Void))];
builtinf = true;
]

let root_symbol_table =
{ ret_typ = Sast.Void;
ret_nam = "";
func_nam = "";
formal_param = [];
local = [];
builtin = builtin_vars; }

let root_environment =
{ scope = root_symbol_table;
functions = builtin_funcs; }

(**************
* Exceptions *
**************
)

exception Except of string

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

let qub_error t = match t with
| 0 -> raise (Except("Invalid qubit: incorrect use of |expr>"))
| 1 -> raise (Except("Invalid qubit: incorrect use of <expr|"))
| _ -> raise (Except("Invalid qubit"))
let program_error t = match t with
  | _ -> raise (Except("Invalid function call: ' ^ s ^ ' was not declared"))

let stmt_error t = match t with
  | _ -> raise (Except("Invalid statement"))

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

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

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

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

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

let binop_error t = match t with
  | _ -> raise (Except("Invalid use of binop: ' expr @ expr '")

let unop_error t = match t with
  |301x102(102,301,810,825)

let call_error t = match t with
  | _ -> raise (Except("Invalid function call: function undeclared"))

let stmt_error t = match t with
  | _ -> raise (Except("Invalid statement: ' ^ s ^ '")

let program_error t = match t with
  | _ -> raise (Except("Missing 'compute' function"))
| 1 -> raise (Except('compute function must be of type int'))
| _ -> raise (Except('Invalid program'))

(**********************
* Utility Functions *
**********************)

let var_exists name scope =
  if (List.exists (fun vdecl -> name = vdecl.sname) scope.formal_param) then true
  else if (List.exists (fun vdecl -> name = vdecl.sname) scope.formal_param) then true
  else List.exists (fun vdecl -> name = vdecl.sname) scope.builtin

let func_exists name env =
  List.exists (fun fdecl -> name = fdecl.sfunc_name) env.functions

let lookup_var name scope =
  let vdecl_found =
    try List.find (fun vdecl -> name = vdecl.sname) scope.formal_param
    with Not_found ->
      try List.find (fun vdecl -> name = vdecl.sname) scope.local
      with Not_found ->
        try List.find (fun vdecl -> name = vdecl.sname) scope.builtin
        with Not_found -> var_error name in
    vdecl_found

let lookup_func name env =
  let fdecl_found =
    try
    List.find (fun fdecl -> name = fdecl.sfunc_name) env.functions
    with Not_found -> func_error name
  in
  fdecl_found

(*************
* Checks *
*************)

let rec check_qub_expr i =
  let r = i mod 10 in
  if (r = 0 || r = 1) then
    let i = i / 10 in
    if (i != 0)
      then
        check_qub_expr i
      else 1
    else 0
  and check_qub i =
    let int_expr =
      int_of_string i
    in
    if (check_qub_expr int_expr = 1) then
      (match t with
        0 -> Sast.Expr(Sast.Lit_qub(i, 1), Sast.Mat)
        | 1 -> Sast.Expr(Sast.Lit_qub(i, 0), Sast.Mat)
        | _ -> qub_error 2)
    else
      qub_error t
  and check_mat l env =
    let mat =
      List.map (fun row -> check_mat_rows row env) l
    in
    Sast.Expr(Sast.Mat(mat), Sast.Mat)
and check_mat_rows l env =
  let row =
    List.map (fun e -> check_mat_row e env) l
  in row

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

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

and check_unop op e env =
  let e = check_expr env e in
  match e with
  | Sast.Expr(q, t) ->
    (match op with
      | Ast.Neg ->
        (match t with
          | Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
          | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
          | Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
          | _ -> unop_error op)
      | Ast.Not ->
        (match t with
          | Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
          | _ -> unop_error op)
      | Ast.Re ->
        (match t with
          | Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
          | _ -> unop_error op)
      | Ast.Im ->
        (match t with
          | Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
          | _ -> unop_error op)
      | Ast.Unit ->
        (match t with
          | Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
          | _ -> unop_error op)
      | Ast.Norm ->
        (match t with
          | Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
          | _ -> unop_error op)
      | Ast.Det ->
        (match t with
          | Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
          | _ -> unop_error op)
      | Ast.Trans | Ast.Adj ->
        (match t with
          | Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Mat)
          | _ -> unop_error op)
      | Ast.Conj ->
        (match t with...
Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
| Sast.Mat -> Sast.Expr(Sast.Unop(op, e), Sast.Mat)
| _ -> unop_error op

| Ast.Sin ->
  (match t with
    Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
    | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
    | Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
    | _-> unop_error op)

| Ast.Cos ->
  (match t with
    Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
    | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
    | Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
    | _-> unop_error op)

| Ast.Tan ->
  (match t with
    Sast.Int -> Sast.Expr(Sast.Unop(op, e), Sast.Int)
    | Sast.Float -> Sast.Expr(Sast.Unop(op, e), Sast.Float)
    | Sast.Comp -> Sast.Expr(Sast.Unop(op, e), Sast.Comp)
    | _-> unop_error op)

and check_binop e1 op e2 env =
  let e1 = check_expr env e1 and e2 = check_expr env e2 in
  match t1 with
    Sast.Expr(_, t1) ->
      (match e2 with
        Sast.Expr(_, t2) ->
          (match op with
            Ast.Add | Ast.Sub ->
              (match t2 with
                Sast.Int ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | _-> binop_error op)
                Sast.Float ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | _-> binop_error op)
                Sast.Comp ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | _-> binop_error op)
                Sast.Mat ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
                    | _-> binop_error op)
                    | _-> binop_error op)
            | Ast.Mult | Ast.Div ->
              (match t1 with
                Sast.Int ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
                    | _-> binop_error op)
                Sast.Float ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
                    | _-> binop_error op)
                Sast.Comp ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
                    | _-> binop_error op)
                Sast.Mat ->
                  (match t2 with
                    Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                    | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Float)
                    | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Comp)
                    | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Mat)
                    | _-> binop_error op)
                _-> binop_error op)
Float )
| Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Comp )
| Sast . Mat -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Mat )
| _ -> binop_error op
| Sast . Comp ->
(match t2 with
  Sast . Int | Sast . Float | Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op ,
e2 ) , Sast . Comp )
  Sast . Mat -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Mat )
  | _ -> binop_error op
| Sast . Mat ->
(match t2 with
Binop ( e1 , op , e2 ) , Sast . Mat )
| _ -> binop_error op
| Ast . Mod | Ast . Expn ->
(match t1 with
Sast . Int ->
(match t2 with
  Sast . Int -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
  Sast . Float -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Float )
  Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Comp )
| _ -> binop_error op)
| Sast . Float ->
(match t2 with
Float )
  Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Comp )
| _ -> binop_error op)
| Sast . Comp ->
(match t2 with
  Sast . Int | Sast . Float | Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op ,
e2 ) , Sast . Comp )
| _ -> binop_error op)
| Ast . Tens ->
(match t1 with
Sast . Mat ->
(match t2 with
  Sast . Mat -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Mat )
| _ -> binop_error op)
| _ -> binop_error op)
| Ast . Eq | Ast . Neq ->
(match t1 with
Sast . Int ->
(match t2 with
  Sast . Int -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
  Sast . Float -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
  Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
| _ -> binop_error op)
| Sast . Float ->
(match t2 with
  Sast . Int -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
  Sast . Float -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
  Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
| _ -> binop_error op)
| Sast . Comp ->
(match t2 with
  Sast . Int -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
  Sast . Float -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
  Sast . Comp -> Sast . Expr ( Sast . Binop ( e1 , op , e2 ) , Sast . Int )
| _ -> binop_error op)
| Sast . Mat ->

67
(match t2 with
  | Sast.Mat -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
  | _ -> binop_error op)
| Ast.Lt | Ast.Gt | Ast.Leq | Ast.Geq ->
  (match t1 with
    | Sast.Int ->
      (match t2 with
        | Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
        | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
        | _ -> binop_error op)
        | Sast.Float ->
          (match t2 with
            | Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
            | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
            | _ -> binop_error op)
          | Ast.Or | Ast.And | Ast.Xor ->
            (match t1 with
              | Sast.Int ->
                (match t2 with
                  | Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                  | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                  | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
                  | _ -> binop_error op)
          | Sast.Float ->
            (match t2 with
              | Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
              | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
              | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
              | _ -> binop_error op)
          | Sast.Comp ->
            (match t2 with
              | Sast.Int -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
              | Sast.Float -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
              | Sast.Comp -> Sast.Expr(Sast.Binop(e1, op, e2), Sast.Int)
              | _ -> binop_error op)))

and check_assign name e env =
  let vdecl = lookup_var name env.scope in
  let e = check_expr env e in
  match e with
t    | Sast.Expr(_, t1) ->
      let t2 = vdecl.styp in
      if (t1 = t2) then
        Sast.Expr(Sast.Assign(name, e), t1)
      else
        assignment_error name

and check_call_params formal_params params =
  if ((List.length formal_params) = 0)
    then true
  else
    let fdecl_arg = List.hd formal_params in
    let param = match (List.hd params) with
      | Sast.Expr(_, t) -> t in
      if (fdecl_arg.styp = Sast.Poly || (fdecl_arg.styp = param))
        then check_call_params (List.tl formal_params) (List.tl params)
      else false

and check_call name params env =
  let fdecl =
    try
      lookup_func name env
    with 
      ignore

with Not found -> call_error 0 in
let params = List.map (check_expr env) params in
if (List.length fdecl.sformal_params) != (List.length params)
  then call_error 1
else if (check_call_params fdecl.sformal_params params) = true
  then Sast.Expr(Sast.Call(name, params), fdecl.sret_typ)
else
  call_error 2

and check_expr env = function
  Ast.Lit_int(i) -> Sast.Expr(Sast.Lit_int(i), Sast.Int)
| Ast.Lit_float(f) -> Sast.Expr(Sast.Lit_float(f), Sast.Float)
| Ast.Lit_comp(f1, f2) -> Sast.Expr(Sast.Lit_comp(f1, f2), Sast.Comp)
| Ast.Lit_qub(i, t) -> check_qub i t
| Ast.Mat(l) -> check_mat l env
| Ast.Id(s) -> check_id s env
| Ast.Unop(op, e) -> check_unop op e env
| Ast.Binop(e1, op, e2) -> check_binop e1 op e2 env
| Ast.Assign(s, e) -> check_assign s e env
| Ast.Call(s, l) -> check_call s l env

and check_block stmts env =
let sstmts = List.map (fun stmt -> check_stmt env stmt) stmts in
Sast.Block(sstmts)

and check_if e s1 s2 env =
let se = check_expr env e in
match se with
  Sast.Expr(_, t) ->
    (match t with
      Sast.Int ->
        let ss1 = check_stmt env s1 in
        let ss2 = check_stmt env s2 in
        Sast.If(se, ss1, ss2)
      _ -> stmt_error 0)

and check_for e1 e2 e3 e4 s env =
let sel = check_expr env e1 in
match sel with
  Sast.Expr(Sast.Id(_), Sast.Int) ->
    let se2 = check_expr env e2 in
    (match se2 with
      Sast.Expr(_, Sast.Int) ->
        let se3 = check_expr env e3 in
        (match se3 with
          Sast.Expr(_, Sast.Int) ->
            let se4 = check_expr env e4 in
            (match se4 with
              Sast.Expr(_, t) ->
                (match t with
                  Sast.Int ->
                    let ss = check_stmt env s in
                    Sast.For(sel, se2, se3, se4, ss)
                  Sast.Void ->
                    let ss = check_stmt env s in
                    Sast.For(sel, se2, se3, Sast.Expr(Sast.Lit_int(1), Sast.Int), ss)
                  _ -> stmt_error 1))
              _ -> stmt_error 1)
            _ -> stmt_error 1)
        _ -> stmt_error 1)
      _ -> stmt_error 1)
    _ -> stmt_error 1)
  _ -> stmt_error 1)

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

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

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

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
  else let sdecl = vdecl_to_sdecl formal_param in
  let new_formals = sdecl :: scope.formal_param in
  let new_scope =
    { 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

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

and local_to_slocal scope local =
  let found = var_exists local.name scope in
  if found then var_decl_error local.name
  else let sdecl = vdecl_to_sdecl local in
  let new_locals = sdecl :: scope.local in
  let new_scope =
    { 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

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

and ret_to_sret scope ret_typ =
  let ret_typ =
    match ret_typ with

let new_scope =
  \{ ret_typ = sret_typ;
     ret_nam = scope.ret_nam;
     func_nam = scope.func_nam;
     formal_param = scope.formal_param;
     local = scope.local;
     builtin = scope.builtin; \} in
new_scope

and rname_to_srname scope ret_name =
  let new_scope = \{ ret_typ = scope.ret_typ;
                    ret_nam = ret_name;
                    func_nam = scope.func_nam;
                    formal_param = scope.formal_param;
                    local = scope.local;
                    builtin = scope.builtin; \} in
new_scope

and fname_to_sfname scope func_name =
  let new_scope = \{ ret_typ = scope.ret_typ;
                    ret_nam = scope.ret_nam;
                    func_nam = func_name;
                    formal_param = scope.formal_param;
                    local = scope.local;
                    builtin = scope.builtin; \} in
new_scope

and ret_to_slocal scope name typ =
  let vdecl = { typ = typ; name = name; } in
  let sdecl = vdecl_to_sdecl vdecl in
  let new_locals = sdecl :: scope.local in
  let new_scope = \{ 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

and fdecl_to_sdecl fdecl env =
  let new_scope = ret_to_slocal env.scope fdecl.ret_name fdecl.ret_typ in
  let new_locals = locals_to_sformals new_scope fdecl.formal_params in
  let new_scope = locals_to_slocals new_scope fdecl.locals in
  let new_scope = ret_to_sret new_scope fdecl.ret_typ in
  let new_scope = rname_to_sfname new_scope fdecl.func_name in
  let new_env = \{ scope = new_scope; functions = env.functions; \} in
  let stmts = List.map (fun stmt -> check_stmt new_env stmt) fdecl.body in
  \{ sret_typ = new_scope.ret_typ;
    sret_name = new_scope.ret_name;
    sfunc_name = new_scope.func_name;
    sformal_params = new_scope.formal_params;
    slocals = new_scope.local;
    sbody = stmts;
    builtinf = false; \}

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

new_env

and check_compute_fdecl fdecls =
  let fdecl = List.hd (List.rev fdecls) in
  let name = fdecl.func_name in
  if (name = "compute") then fdecls
    else program_error 0

and check_program fdecls =
  let fdecls = check_compute_fdecl fdecls in
  let env = List.fold_left check_function root_environment fdecls in
  let sfdecls = List.rev env.functions in
  sfdecls

B.5 SAST

sast.ml

(* Sankalpa Khadka *)
open Ast

type sdata_type =
  | Int
  | Float
  | Comp
  | Mat
  | Poly
  | Void

and expr_wrapper =
  Expr of sexpr * sdata_type

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

and sstmt =
  Sexpr of expr_wrapper
  | Block of sstmt list
  | If of expr_wrapper * sstmt * sstmt
  | For of expr_wrapper * expr_wrapper * expr_wrapper * expr_wrapper * sstmt
  | While of expr_wrapper * sstmt
  | BreakCont of int

and svar_decl =
  { styp : sdata_type;
    sname : string;
    builtinv : bool;
  }
and sfunc_decl =
{
  sret_typ : sdata_type;
  sret_name : string;
  sfunc_name : string;
  sformal_params : svar_decl list;
  slocals : svar_decl list;
  sbody : sstmt list;
  builtinf : bool;
}

type sprogram =
  sfunc_decl list
(* Pretty Printer *)
let rec string_of_unop op e =
  (match op with
   | Neg -> ' − '
   | Not -> ' ! '
   | Re -> ' Re '
   | Im -> ' Im '
   | Norm -> ' Norm '
   | Trans -> ' Trans '
   | Det -> ' Det '
   | Adj -> ' Adj '
   | Conjugate -> ' Conjugate '
   | Unit -> ' Unit '
   | Sin -> ' Sin '
   | Cos -> ' Cos '
   | Tan -> ' Tan '
   | string_of_expr_wrapper e
and string_of_binop e1 op e2 =
  string_of_expr_wrapper e1 ^
  (match op with
   | Add -> ' + ' | Sub -> ' − ' | Mult -> ' * '
   | Eq -> ' == ' | Neq -> ' != ' | Lt -> ' < '
   | Leq -> ' <= ' | Gt -> ' > ' | Geq -> ' >= '
   | Xor -> ' XOR ' | And -> ' && ' | Or -> ' || '
   | string_of_expr_wrapper e2
and string_of_mat l =
  let row strs =
    List.map string_of_row l
  in
    "[ " ^ String.concat " " row strs ^ " ] "
and string_of_row r =
  let row_str =
    String.concat " , " (List.map string_of_expr_wrapper r)
  in
    "( " ^ row_str ^ " ) "
and string_of_sexpr = function
  Lit_int (i) -> string_of_int i
  | Lit_float (f) -> string_of_float f
  | Lit_compl (f1, f2) -> string_of_float f1 ^ ' + ' ^ string_of_float f2 ^ ' j ' 
  | Lit_qub (i, t) -> i
  | Mat (l) -> string_of_mat l
  | Id s -> s
  | Unop (op, e) -> string_of_unop op e
  | Binop (e1, op, e2) -> string_of_binop e1 op e2
  | Assign (name, e) -> name ^ " = " ^ string_of_expr_wrapper e
  | Call (name, params) -> " Calling " ^ name ^ " on " ^ string_of_sexprs params
  | Noexpr -> " noexpr"
and string_of_expr_wrapper w =
  let sexpr =
    match w with
    | Expr(Lit_int(i), Int) -> Lit_int(i)
    | Expr(Lit_float(f), Float) -> Lit_float(f)
    | Expr(Lit_comp(f1, f2), Comp) -> Lit_comp(f1, f2)
    | Expr(Mat(l), Mat) -> Mat(l)
    | Expr(Id(name), typ) -> Id(name)
    | Expr(Unop(op, e), _) -> Unop(op, e)
    | Expr(Binop(e1, op, e2), _) -> Binop(e1, op, e2)
    | Expr(Assign(name, e), t1) -> Assign(name, e)
    | Expr(Call(name, params), _) -> Call(name, params)
    | Expr(Lit_qub(i, t), _) -> Lit_qub(i, t)
    | _ -> Noexpr
    in
    string_of_sexpr sexpr

and string_of_svar_decl svar_decl =
  'svdecl : styp:
    (match svar_decl.styp with
     | Int -> 'int,' name: ' ^ svar_decl.sname ^ ' 
     | Float -> 'float,' ^ ' name: ' ^ svar_decl.sname ^ ' 
     | Comp -> 'comp,' ^ ' name: ' ^ svar_decl.sname ^ ' 
     | Mat -> 'mat,' ^ ' name: ' ^ svar_decl.sname ^ ' 
     | _ -> ')

and string_of_sexprs e =
  String.concat '\n' (List.map string_of_expr_wrapper e)

and string_of_sstmts sstmts =
  function
  Sexpr(e) -> string_of_expr_wrapper e ^ '\n'
  | Block(l) -> '*' string_of_sstmts l ^ '\n' ^ string_of_sstmt s2
  | If(e, s, Block([])) -> 'if (' ^ string_of_expr_wrapper e ^ ')\n' ^ string_of_sstmt s1 ^ 'else\n' ^ string_of_sstmt s
  | 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 : ' ^ string_of_sstmt s
  | While(e, s) -> 'While condition : ' ^ string_of_expr_wrapper e ^ '\nstatement : ' ^ string_of_sstmt s
  | BreakCont(t) -> string_of_breakcont t

and string_of_breakcont t =
  if (t = 0) then
    'break'
  else
    'continue'

and string_of_sfdcl sfdecl =
  'ssfdecl:\nsret_typ:
    (match sfdecl.sret_typ with
     | Int -> 'int
     | Float -> 'float
     | Comp -> 'comp
     | Mat -> 'mat
     | _ -> ')
    ^'nsret_name: ' ^ sfdecl.sret_name ^ 'nsfunc_name: ' ^ sfdecl.sfunc_name ^ '\n(' ^ String.concat ' ' (List.map string_of_svar_decl sfdecl.sformal_params) ^ ' )\n{\n'( ^ String.concat ' ' (List.map string_of_svar_decl sfdecl.slocals) ^ ' )\n'}

and string_of_sstmt sstmts =
  String.concat '\n' (List.map string_of_sstmt sstmts)
and string_of_sprogram (l) =
  "program:\n" ^ String.concat "\n" (List.map string_of_sfdecl l)
fprintf "\%s\" outStr

and cpp_funcList func =
  if func.builtin then ""
else
  let cppFName = func.sfunc_name
  and cppRtnType = cpp_from_type func.sret_typ
  and cppFParm = if (func.sformal_params = []) then "" else cppVarDecl func.sformal_params ",",
  and cppFBody = cppStmtList func.sbody
  and cppLocals = cppVarDecl func.slocals ";\n  in
  if cppFName = 'compute' then
    fprintf "\nint main ()\n{\n  %s\n  %s\n  std::cout << %s << endl;\n  return 0;\n}" cppLocals cppFBody cppRtnValue
  else
    if (cppFParm = "") then
      fprintf "\n%s %s ()\n{\n  %s\n  %s\n  return %s;\n}" cppRtnType cppFName cppLocals cppFBody cppRtnValue
    else
      fprintf "\n%s %s (%s)\n{\n  %s\n  %s\n  return %s;\n}" cppRtnType cppFName cppFParm cppLocals cppFBody cppRtnValue

(* generate variable declarations *)
and cppVarDecl vardeclist delim =
  let varDecStr = List.fold_left (fun a b -> a ^ (cppVar b delim)) "" vardeclist
  in
  let varDectrun = String.sub varDecStr 0 ((String.length varDecStr) - 1)
  in
    sprintf %s %s * varDectrun

(* generate variable declaration *)
and cppVar var delim =
  if not var.builtin then
    let vartype =
      cpp_from_type var.styp
    in
      sprintf %s %s %s %s * vartype var.sname delim
    else ""

(* generate list of statements *)
and cppStmtList astmlist =
  let outStr =
    List.fold_left (fun a b -> a ^ (cppStmt b)) "" astmlist
  in
    sprintf %s %s * outStr

(* generate statement *)
and cppStmt stmts = match stmts with
  Sast.Sexpr(expr_wrap) -> \"\n  ^ cppExpr (expr_of expr_wrap) ^ \";\n  \n| Sast.Block(stmtt) -> cppStmtBlock stmtt
| Sast.If(expr_wrap, stmtt1, stmtt2) -> writeIfStmt (expr_of expr_wrap) stmtt1 stmtt2
| Sast.For(var, init, final, increment, stmt) -> writeForStmt var init final increment stmt
| Sast.While(expr_wrap, stmt) -> writeWhileStmt (expr_of expr_wrap) stmt
| Sast.BreakCont(t) -> writeBreakCont t

(* generate break/continue statement *)
and writeBreakCont t =
  if (t =0) then
    fprintf 'break;'
  else
    ...
```csharp
* generate expression *)
and cppExpr expr = match expr with
  Lit_int(lit) -> string_of_int lit
  Lit_float(flit) -> string_of_float flit
  Lit_comp(re,im) -> "complex<float>(" ^ string_of_float re ^ "," ^ string_of_float im ^ ")")
  Unop(op, expr) -> writeUnop op expr
  Binop(expr1, op, expr2) -> writeBinop expr1 op expr2
  Lit_qub(vec, t) -> writeQubit vec t
  Mat(expr_wrap) -> writeMatrix expr_wrap
  Id(str) -> str
  Assign(name, expr) -> name ^ " = " ^ cppExpr (expr_of expr)
  Call(name, l) ->
    if is_builtin_func name then
      writeBuiltinFuncCall name l
    else
      name ^ "(" ^ writeFunCall l ^ ")"*
  Noexpr -> ""

(* generate built-in function call *)
and writeBuiltinFuncCall name l =
  match name with
  "print" -> writePrintStmt l
  "printq" -> writePrintqStmt l
  "rows" -> writeRowStmt l
  "cols" -> writeColStmt l
  "elem" -> writeElemStmt l
  _ -> **

(* generate row statement *)
and writeRowStmt l =
  let expr_wrap = List.hd l in
  let expr = cppExpr (expr_of expr_wrap) in
  sprintf "%s.rows()" expr

(* generate col statement *)
and writeColStmt l =
  let expr_wrap = List.hd l in
  let expr = cppExpr (expr_of expr_wrap) in
  sprintf "%s.cols()" expr

(* generate elem statement *)
and writeElemStmt l =
  let ew1 = List.hd l in
  let el = cppExpr (expr_of ew1)
  and ew2 = List.hd (List.tl l) in
  let e2 = cppExpr (expr_of ew2)
  and ew3 = List.hd (List.tl (List.tl l)) in
  let e3 = cppExpr (expr_of ew3) in
  sprintf "%s(%s,%s)" el e2 e3

(* generate print statement *)
and writePrintStmt l =
  let expr_wrap = List.hd l in
  let expr = cppExpr (expr_of expr_wrap) in
  match expr_wrap with
    Sast.Expr(_, t) ->
      (match t with
        Sast.Sym _ -> sprintf "cout << %s << endl" expr
      )
  _ -> sprintf "cout << %s << endl" expr

(* generate qubit print statement *)
and writePrintqStmt l =
```

let expr_wrap = List.hd 1 in
let expr = cppExpr (expr_of expr_wrap) in
match expr_wrap with
  Sast.Expr(_, t) ->
    (match t with
     Sast.Mat -> sprintf "cout << vectorToBraket(%s) << endl" expr
     _ -> sprintf "cout << %s << endl" expr)

(* generate block *)
and cppStmtBlock stmtl1 =
  let slist = List.fold_left (fun output element ->
    let stmt = cppStmt element in
    output ^ stmt ^ "\n") " " stmtl1 in
  "\n\t{\n" ^ slist ^ "\t}\n"

(* generate if statement *)
and writeIfStmt expr stmt1 stmt2 =
  let cond = cppExpr expr in
  let body = cppStmt stmt1 in
  let ebody = writeElseStmt stmt2 in
  sprintf " if(%s)\n%s%s" cond body ebody

(* generate else statements *)
and writeElseStmt stmt =
  let body =
    if ((String.compare body "\t;\n") = 0) then
      sprintf "\n"
    else
      sprintf " \else%s" body

(* generate while statement *)
and writeWhileStmt expr stmt =
  let condString = cppExpr expr
  and stmtString = cppStmt stmt in
  sprintf " while (%s)\n%s\n" condString stmtString

(* generate for statements *)
and writeForStmt var init final increment stmt =
  let varname = cppExpr (expr_of var) in
  let initvalue = cppExpr (expr_of init) in
  let finalvalue = cppExpr (expr_of final) in
  let incrementval = cppExpr (expr_of increment) in
  let stmtbody = cppStmt stmt in
  sprintf " for (int %s = %s ; %s < %s ; %s = %s + %s) {
    %s
  } varname initvalue varname finalvalue varname varname incrementval stmtbody

(* generate unary operators *)
and writeUnop op expr =
  let unopFunc op exp = match op with
    Ast.Neg -> sprintf "-%s" exp
    Ast.Not -> sprintf "!(%s)" exp
    Ast.Re -> sprintf "real(%s)" exp (* assumes exp is matrix*)
    Ast.Im -> sprintf "imag(%s)" exp
    Ast.Norm -> sprintf "%s.norm()" exp
    Ast.Trans -> sprintf "%s.transpose()" exp
    Ast.Det -> sprintf "%s.determinant()" exp
    Ast.Adj -> sprintf "%s.adjoint()" exp
    Ast.Conj -> sprintf "%s.conjugate()" exp
    Ast.Unit -> sprintf "(%s.conjugate() + %s).isIdentity()" exp exp (* till here
(* generate binary operations *)
and writeBinop expr1 op expr2 =
  let el1 = cppExpr (expr_of expr1)
  and tl1 = type_of expr1
  and el2 = cppExpr (expr_of expr2) in
  let binopFunc el1 tl1 op el2 = match op with
    | Ast.Add -> sprintf "%s + %s" el1 el2
    | Ast.Sub -> sprintf "%s - %s" el1 el2
    | Ast.Mult -> sprintf "%s * %s" el1 el2
    | Ast.Div -> sprintf "%s / %s" el1 el2
    | Ast.Mod -> sprintf "%s % % %s" el1 el2
    | Ast.Expn -> sprintf "pow(%s,%s)" el1 el2
    | Ast.Tens -> sprintf "tensor(%s,%s)" el1 el2
    | Ast.Eq -> equalCaseWise el1 tl1 el2
    | Ast.Neq -> sprintf "%s != %s" el1 el2
    | Ast.Lt -> sprintf "%s < %s" el1 el2
    | Ast.Gt -> sprintf "%s > %s" el1 el2
    | Ast.Leq -> sprintf "%s <= %s" el1 el2
    | Ast.Geq -> sprintf "%s >= %s" el1 el2
    | Ast.Or -> sprintf "%s || %s" el1 el2
    | Ast.And -> sprintf "%s && %s" el1 el2
    | Ast.Xor -> sprintf "%s ^ %s" el1 el2
  in binopFunc el1 tl1 op el2

(* generate equality expressions (structural equality is used) *)
and equalCaseWise el1 tl1 el2 = match tl1 with
  | Sast.Mat -> sprintf "%s . isApprox(%s)" el1 el2
  | _ -> sprintf "%s == %s" el1 el2

(* generate matrix *)
and writeMatrix expr_wrap =
  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
  sprintf "( Matrix<complex<float>, Dynamic , Dynamic>(%d,%d)< <%s ) . finished ()" (rowMatrix expr_wrap) (colMatrix expr_wrap) submatrix

(* generate matrix row *)
and writeRow row_expr =
  let rowStr = List.fold_left (fun a b -> a ^ (cppExpr (expr_of b)) ^ " , ") " row_expr in
  sprintf "%s" rowStr

(* generate column matrix *)
and colMatrix expr_wrap =
  List.length (List.hd expr_wrap)

(* generate row matrix *)
and rowMatrix expr_wrap =
  List.length expr_wrap

(* generate function call *)
and writeFunCall expr_wrap =
  if expr_wrap = [] then
    sprintf ""
  else
    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
    sprintf "%s" argvStrCom
B.7 Scripts

B.7.1 Makefile

Makefile

```bash
# Christopher Campbell, Jonathan Wong
# stuff for compiling cpp files
CXX = g++
CPPDIR = ./cpp
INC = $(CPPDIR)/includes/headers
INCLUDES = $(INC:%=-I%)
CXXFLAGS = -g -Wall $(INCLUDES)
OBJJS = ast.cmo sast.cmo parser.cmo scanner.cmo analyzer.cmo generator.cmo qlc.cmo
.PHONY: default

default: qlc cpp/qlang.o

qlc : $(OBJJS)
     ocamlc -g -o qlc $(OBJJS)

scanner.ml : scanner.mll
     ocamllex scanner.mll

parser.ml parser.mli : parser.mly
     ocamlyacc parser.mly

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

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

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

.PHONY : clean

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

# Generated by ocamldep *.ml *.mli
analyzer.cmo: sast.cmo ast.cmo
analyzer.cmx: sast.cmx ast.cmx
generator.cmo: sast.cmo
generator.cmx: sast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
qlc.cmo: scanner.cmo sast.cmo parser.cmi ast.cmo analyzer.cmo
qlc.cmx: scanner.cmx sast.cmo parser.cmx ast.cmx analyzer.cmx
sast.cmo: ast.cmo
```

```
B.7.2 Compilation script

qlc.ml

```ocaml
let __ =
  let action =
    List.assoc Sys.argv.(1) [('-a', Ast); ('-s', Sast); ('-g', Gen); ('-d', Debug);]
in
  let lexbuf = Lex.ing.from_channel (open_in Sys.argv.(2)) (*stdin *) and
  output_file = String.sub Sys.argv.(2) 0 (String.length(Sys.argv.(2)) - 3) in
  match action with
    Ast -> print_string (Ast.string_of_program program)
    | Sast ->
      let sprogram =
        Analyzer.check_program program
      in
        print_string (Sast.string_of_sprogram sprogram)
    | Gen -> Generator.gen_program output_file (Analyzer.check_program program)
    | Debug -> print_string "debug"
```

B.7.3 Testing script

runTests.sh

```bash
#!/bin/bash

AST=0
SAST=0
GEN=0
COMP=0
EXEC=0

if [ $1 == "clean" ]
then
  rm -f ast_error_log sast_error_log gen_error_log comp_error_log ast_log sast_log ast_output
  sast_output exec_output
  rm -f SemanticSuccess/*.cpp SemanticSuccess/*.o
  rm -f SemanticFailure/*.cpp SemanticFailure/*.o
  rm -f Analyzer/*.cpp Analyzer/*.o
else

if [ $1 == "a" ]
then
  AST=1
fi
```
if [$1 == "s"]
then
SAST=1
fi
if [$1 == "g"] || [$1 == "c"] || [$1 == "e"]
then
GEN=1
fi
if [$1 == "c"]
then
COMP=1
fi
if [$1 == "e"]
then
EXEC=1
fi
if [$2 == "ss"]
then
files="SemanticSuccess/*.*.ql"
cfiles="SemanticSuccess/*.*.cpp"
elif [$2 == "sf"]
then
files="SemanticFailures/*.*.ql"
cfiles="SemanticFailures/*.*.cpp"
elif [$2 == "al"]
then
files="Algorithms/*.*.ql"
cfiles="Algorithms/*.*.cpp"
f

ASTCheck()
{
eval "../qlc-a $1" 1>> ast_output 2>> ast_error_log
wc ast_error_log | awk '{print $1}'
}

SASTCheck()
{
eval "../qlc-s $1" 1>> sast_output 2>> sast_error_log
wc sast_error_log | awk '{print $1}'
}

GenerationCheck()
{
eval "../qlc-g $1" 2>> gen_error_log
wc gen_error_log | awk '{print $1}'
}

CompilationCheck()
{
eval "g++ -w $1 -I../includes=headers -L../includes/libs -lqlang" 2>> comp_error_log
wc comp_error_log | awk '{print $1}'
}

ExecutionCheck()
{
output=$(eval ".a.out")
echo "" >> exec_output
echo "Output: " >> exec_output
echo "$output" >> exec_output
echo "$output"
}

#Check AST
if [ $AST == 1 ]
then
  echo "* AST Generation *"
  rm -f ast_error_log ast_output
  errors=0
  prev_errors=0
  for file in $files
do
    errors=0
    errors=$(ASTCheck $file)
    if [ "$errors" -le "$prev_errors" ]
    then
      count=1
    else
      echo 'Pass' $file
      echo 'Fail' $file
    fi
    prev_errors=$errors
done
else
  echo **
fi

#Check SAST
if [ $SAST == 1 ]
then
  echo "* SAST Generation *"
  rm -f sast_error_log sast_output
  errors=0
  prev_errors=0
  for file in $files
do
    errors=$(SASTCheck $file)
    if [ "$errors" -le "$prev_errors" ]
    then
      echo 'Pass: ' $file
    else
      echo 'Fail: ' $file
    fi
    prev_errors=$errors
done
else
  echo **
fi

#Check Generation
if [ $GEN == 1 ]
then
cd ..;/cpp
make
cd ..;/test
  echo "* Code Generation *"
  rm -f gen_error_log
  errors=0
  prev_errors=0
  for file in $files
do
    errors=$(GenerationCheck $file)
    if [ "$errors" -le "$prev_errors" ]
    then
      echo 'Pass: ' $file
    else
      echo 'Fail: ' $file
    fi
    prev_errors=$errors
done
else
  echo **
fi
B.8 Programs

B.8.1 Demo

demo1.ql
# Sankalpa Khadka

def compute():
    mat a;
    mat b;
    mat c;
    mat k;
    a = |11>;
    b = |0>;
    k = <0|;
    c = a @ b;
    printq(c);
    c = H*b;
    printq(c);
    output = b*k;
}

demo2.ql

# Sankalpa Khadka

def measure(mat top): mat outcome{
    mat ad;
    ad = adj(top);
    outcome = top*ad;
}

def outcomezero(mat bottom): float probability{
    mat top;
    mat input;
    mat had;
    mat cnot;
    mat ynot;
    mat output;
    mat meas;
    top = |0>;
    input = top @ bottom;
    had = H @ IDT;
    cnot = [(1,0,0,0),
            (0,1,0,0),
            (0,0,0,1),
            (0,0,1,0)];
    ynot = [(1,0,0,0),
            (0,0,0,C(1.01)),
            (0,0,1,0),
            (0,C(-1.1),0,0)];
    output = (ynot*(cnot*(had*input)));
    printq(output);
    probability = norm(output);
```python
def compute() : float outcome{
    mat bottom;
    bottom = |1>;
    outcome = outcomezero(bottom);
    print(outcome);
    bottom = |0>;
    outcome = outcomezero(bottom);
}

demo3.ql
# Sankalpa Khadka
# simulation of Deutsch's Algorithm
def measure(mat top) : mat outcome{
    # returns the measurement matrix for top qubit
    mat ad;
    ad = adj(top);
    outcome = top * ad;
}
def hadamard(int n) : mat gate{
    # returns Hadamard gate for n qubit system
    int i;
    gate = H;
    for (i from 0 to n-1 by 1) {
        gate = gate @ H;
    }
}
def topqubit(int n) : mat input{
    # return zero qubit for n qubit system
    int i;
    input = |0>;
    for (i from 0 to n-1 by 1) {
        input = input @ |0>;
    }
}
def deutsch(int n, mat U) : float outcomeZero{
    mat bottom; mat top; mat input;
    mat hadtop; mat meas;
    bottom = |1>;
    top = topqubit(n);
    input = top @ bottom;  # input qubit, tensor of top and bottom
    hadtop = hadamard(n);
    input = (hadtop @ H)*input;  # application of Hadamard gate
    input = U * input;  # application of the Oracle U
    input = (hadtop @ IDT)*input;  # application of Hadamard on top only, IDT-Identity
    meas = measure(top);
    input = (meas @ IDT)* input;  # measure zero on top register
    outcomeZero = norm(input);  # likelihood of getting zero on top register
}```
```python
def compute():
    float outcome = 0;
    int n; mat Ub; mat Uc;

    # test for n = 1
    n = 1;
    # Ub - balanced, Uc - Constant Oracles
    Ub = [[(1,0,0,0) (0,1,0,0) (0,0,0,1) (0,0,1,0)]];
    Uc = [[(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)]];
    outcome = deutsch(n, Ub);
    print(outcome);
    outcome = deutsch(n, Uc);
    print(outcome);

    # test for n = 2
    n = 2;
    Ub = [[(1,0,0,0,0,0,0,0) (0,1,0,0,0,0,0,0) (0,0,1,0,0,0,0,0) (0,0,0,1,0,0,0,0) (0,0,0,0,1,0,0,0) (0,0,0,0,0,1,0,0) (0,0,0,0,0,0,1,0) (0,0,0,0,0,0,0,1)]];
    outcome = deutsch(n, Ub);
```
\[ u = n_{\text{tensor}}(n+1, i) \]
\[ \text{gate} = n_{\text{tensor}}(n+1, \text{IDT}) - 2u \]

```python
def prepareG(int n):
    mat gate =
    # Prepare grover definitive operator
    mat s; mat sa; mat i; mat h;
    s = n_{\text{tensor}}(n, |0\rangle);
    sa = adj(s);
    i = n_{\text{tensor}}(n, \text{IDT});
    gate = 2s*sa - i;
    h = n_{\text{tensor}}(n, H);
    gate = h*gate*h;
    gate = gate @ \text{IDT};

def grover(int n):
    float outcomeZero =
    mat bottom; mat top; mat input;
    mat hadtop; mat u; mat g; mat go; mat meas;
    int i;
    bottom = |1\rangle;
    top = n_{\text{tensor}}(n, |0\rangle);
    input = top @ bottom; # input is tensor of top and bottom registers
    hadtop = n_{\text{tensor}}(n, H);
    input = (hadtop @ H)*input; # apply Hadamard gate
    u = prepareU(n);
    g = prepareG(n);
    go = g*u; # Grover Operator
    for (i from 0 to n by 1) {
        input = go*input;
    }
    meas = measure(top);
    input = (meas @ \text{IDT})*input; # measure on top register
    outcomeZero = norm(input); # likelihood to find 0 on top register
```

def compute():
    float outcome =
    # simulate the grover for f(0)=1
    int n; mat Ub; mat Uc;
    n = 1;
    outcome = grover(n);
    print(outcome);
    n = 2;
    outcome = grover(n);
```

B.8.2 Successful Test cases

binop_comp_matrix.ql
#Winnie Narang

def test_func(comp a, comp b, comp c, comp d) : mat ret_val {
    mat x;
    x = [(a,b)(c,d)];
    ret_val = [(a,c)(d,b)];
    ret_val = ret_val * x;
    ret_val = ret_val + x;
    ret_val = ret_val - x;
    ret_val = ret_val / 2;
}

def compute() : mat ret_val {
    comp a;
    comp b;
    comp c;
    comp d;
    mat k;
    a = C(4.+5.I);
    b = C(6.+6.I);
    c = C(7.+8.I);
    d = C(9.+10.I);
    ret_val = test_func(a, b, c, d);
}

binop_float_matrix.ql

#Winnie Narang

def test_func(float a, float b, float c, float d) : mat ret_val {
    mat x;
    x = [(a,b)(c,d)];
    ret_val = [(a,c)(d,b)];
    ret_val = ret_val * x;
    ret_val = ret_val + x;
    ret_val = ret_val - x;
    ret_val = ret_val / 2;
}

def compute() : mat ret_val {
    float a;
    float b;
    float c;
    float d;
    a = 3.4;
    b = 6.6;
    c = 5.6;
    d = 100.0;
def func_test ( int z ) : int ret_name {
    int a ;
    int b ;
    int d ;
    a = z ;
    b = 10 ;
    d = a + b * a + b / a - b ;
    ret_name = d ;
}

def compute ( int a ) : int trial {
    trial = func_test ( 34 ) ;
}

def compute ( ) : mat out {
    mat a ;
    mat b ;
    mat c ;
    a = [ (1) (0) ] ;
    b = [ (0) (1) ] ;
    c = a @ b ;
    print ( c ) ;
}

def func_test ( int a ) : int ret_name {
    int i ;
    for ( i from 0 to 2 by 1 )
        a = a + 5 ;
    for ( i from 2 to 0 by -1 )
        { a = a * 10 ;
          print ( a ) ;
          break ;
        }
    for ( i from 1 to 5 )
        { print ( a ) ;
          continue ;
          a = a * 10 ;
        }
def compute():
    trial = func_test(20);

    ret_name = a;

def compute():
    trial = func_test(20);

#Sankalpa Khadka

def compute():
    int num_rows;
    int num_cols;
    comp val;
    mat m;

    m = [(1, 2, 3) (4, 5, 6) (7, 8, 9)];
    num_rows = rows(m);
    num_cols = cols(m);
    val = elem(m, 1, 2);

    print(num_rows);
    print(num_cols);
    trial = val;

#Sankalpa Khadka

def compute():
    int num_rows;
    int num_cols;
    comp val;
    mat m;

    m = [(1, 2, 3) (4, 5, 6) (7, 8, 9)];
    num_rows = rows(m);
    num_cols = cols(m);
    val = elem(m, 1, 2);

    print(num_rows);
    print(num_cols);
    trial = val;

#Jonathan Wong

def test_func(int a):
    mat ret_val {
        mat x;
    }
mat z;
mat y;
mat w;
x = X;
z = H;
y = Y;
w = IDT;
print(x);
print(y);
print(w);
ret_val = x * z * y * w;
def compute(): mat ret_val {
    ret_val = test_func(0);
}
def test_func(): mat ret_val {
    mat x;
    x = [(1,2) (3,4)];
    ret_val = x;
}
def compute(): mat ret_val {
    ret_val = test_func();
}
def func_test(float b): float ret_name {
    float a;
    float c;
    a = 5.0;
c = a * b;
    ret_name = c;
}
def compute(): float trial {
    trial = func_test(3.7);
}
for_stmt.ql

```python
#Jonathan Wong
def func_test(int z) : int ret_name {
    int i;
    int a;
    for(i from 0 to 2 by 1)
        a=a+5;
    for(i from 2 to 0 by -1)
        {a=a*10;
         print(a);
        }
    for(i from 1 to 10 by 1)
        {a=a-3;
        }
    for(i from 1 to 100)
        {print(a*100);
        }
    ret_name = 5;
}
def compute(int a): int trial {
    trial = func_test(20);
}
```

if_stmt.ql

```python
#Winnie Narang
def func_test(int z) : int ret_name {
    int a;
    int b;
    a = 10;
    if(z eq 5) a = 0;
    a = a - 2;
    if( z leq 5 )
        {a = 0;
        }
    else
        {a = 10;
         b = 24;
        }
    if( a gt 100 )
        {print(b); # a > 100
        }
```
```python
else
    {
        print(a);
    }

    ret_name = 8;

mat_add.ql

#Sankalpa Khadka

def test_func (comp a , comp b , comp c , comp d) : mat ret_val {
    mat x;
    x = [ (a, b)(c, d) ];
    ret_val = x;
}

def compute () : mat trial {
    comp a;
    comp b;
    comp c;
    comp d;
    mat k;
    a = C(2.);
    b = C(2.);
    c = C(2.);
    d = C(2.);
    trial = test_func (a, b, c, d) + test_func (a, b, c, d);
}

mat_mult.ql

#Winnie Narang

def test_func (comp a , comp b , comp c , comp d) : mat ret_val {
    mat x;
    x = [ (a, b)(c, d) ];
    ret_val = x;
}

def compute () : mat trial {
    comp a;
    comp b;
    comp c;
    comp d;
    mat k;
    a = C(2.);
    b = C(2.);
    c = C(2.);
    d = C(2.);
    trial = test_func (a, b, c, d) ∗ test_func (a, b, c, d);
}
```
### mat_qubit.ql

```python
#Winnie Narang
def func_test(mat a, mat b): mat ret_name {
    ret_name = a * b;
}
def compute(int a): mat trial {
    mat zero;
    mat one;
    zero = |0>;
    one = |1>;
    trial = func_test(H, zero);
    printq(trial);
    trial = func_test(H, one);
    printq(trial);
}
```

### un_op_det.ql

```python
#Winnie Narang
def func_test(mat z): mat ret_name {
    mat a;
    comp b;
    a = [(1, 9), (4, 5)];
    b = det(a);
    ret_name = a;
}
def compute(int a): mat trial {
    mat x;
    x = [(1, 2), (3, 4)];
    trial = func_test(x);
}
```

### un_op_trans.ql

```python
#Winnie Narang
def func_test(mat z): mat ret_name {
    mat a;
    mat b;
    a = [(1, 9, 9), (4, 5, 5)];
    b = trans(a);
}
def compute(int a): int trial {
    trial = 8;
}
```
while_stmt.ql

```python
# Winnie Narang

def func_test(int z):
    int ret_name {
        int a;
        a = 5;

        # now checking while with comment
        while (a leq 10)
            a=a+1;

        while (a neq 1)
            {  # Comment, inside
                a = (a+1) % 42;
            }

        ret_name = a;
    }

    def compute():
        int trial {
            trial = func_test(5);
        }
```

B.8.3 Execution output of successful cases

exec_output

```
Output:
(-12.76) (-11.5,98)
(-10,87.5) (-6,114)

Output:
(21.46,0) (290.2,0)
(186.8,0) (600,0)

Output:
364

Output:
2

Output:
(0,0)
(1,0)
(0,0)
(0,0)

Output:
30
30
30
30
30
30

Output:
3
3
3
```
Output: 
(3.52, 8.6)
<table>
<thead>
<tr>
<th></th>
<th>3275000</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>3275000</td>
</tr>
<tr>
<td>98</td>
<td>3275000</td>
</tr>
<tr>
<td>100</td>
<td>3275000</td>
</tr>
<tr>
<td>102</td>
<td>3275000</td>
</tr>
<tr>
<td>104</td>
<td>3275000</td>
</tr>
<tr>
<td>106</td>
<td>3275000</td>
</tr>
<tr>
<td>108</td>
<td>3275000</td>
</tr>
<tr>
<td>110</td>
<td>3275000</td>
</tr>
<tr>
<td>112</td>
<td>3275000</td>
</tr>
<tr>
<td>114</td>
<td>3275000</td>
</tr>
<tr>
<td>116</td>
<td>3275000</td>
</tr>
<tr>
<td>118</td>
<td>3275000</td>
</tr>
<tr>
<td>120</td>
<td>3275000</td>
</tr>
<tr>
<td>122</td>
<td>3275000</td>
</tr>
<tr>
<td>124</td>
<td>3275000</td>
</tr>
<tr>
<td>126</td>
<td>3275000</td>
</tr>
<tr>
<td>128</td>
<td>3275000</td>
</tr>
<tr>
<td>130</td>
<td>3275000</td>
</tr>
<tr>
<td>132</td>
<td>3275000</td>
</tr>
<tr>
<td>134</td>
<td>3275000</td>
</tr>
<tr>
<td>136</td>
<td>3275000</td>
</tr>
<tr>
<td>138</td>
<td>3275000</td>
</tr>
<tr>
<td>140</td>
<td>3275000</td>
</tr>
<tr>
<td>142</td>
<td>3275000</td>
</tr>
<tr>
<td>144</td>
<td>3275000</td>
</tr>
<tr>
<td>146</td>
<td>3275000</td>
</tr>
<tr>
<td>148</td>
<td>3275000</td>
</tr>
<tr>
<td>150</td>
<td>3275000</td>
</tr>
<tr>
<td>152</td>
<td>3275000</td>
</tr>
<tr>
<td>154</td>
<td>3275000</td>
</tr>
<tr>
<td>156</td>
<td>3275000</td>
</tr>
<tr>
<td>158</td>
<td>3275000</td>
</tr>
<tr>
<td>160</td>
<td>3275000</td>
</tr>
</tbody>
</table>
Output: 4
Output: 10
Output: 8
Output: 20
Output: (4.0) (4.0)
Output: (4.0) (4.0)
Output: (8.0) (8.0)
Output: (8.0) (8.0)
Output: (0.707107) |0> + (0.707107) |1>
Output: (−0.707107,0)
Output: (0.0) (0.0)
Output: (0.0) (0.0)
Output: (4.5) (6.6)
Output: (7.8) (9.10)
Output: (−0,−4.5)
Output: 5
Output: (0.0) (1.0) (1.0) (0.0)
Output: <01| + <10|
Output: (0.0) (1.0) (1.0) (0.0)
Output: (1.−0) (4.−0)
Output: (9.−0) (5.−0)
Output: (9.−0) (5.−0)
Output: (1,0) (9,0)
Output: (4,0) (5,0)
Output: (4.5,0)
Output: (0,4.5)
B.8.4 Failed cases

comp_wrong_decl.ql

```python
# Winnie Narang

def func_test(comp val1, comp val2) : comp ret_name {
    comp val3;
    val3 = 1;
    ret_name = val1 + val2 * val3;
}
def compute() : comp ret_name {
    comp comp1;
    comp comp2;
    if (1) {1; 2+3;} else {3+6;}
    comp1 = C(7.51);
    comp2 = C(3.2 + 1.1);
    ret_name = func_test(comp1, comp2);
}
```

func_decl_twice.ql

```python
# Winnie Narang

def func_test1(int z) : int ret_name {
    int a;
    int b;
    int d;
    a = z;
    ret_name = z;
}
def func_test1(int z) : int ret_name2 {
```
if_stmt.ql

```python
# Winnie Narang
def func_test(int z): int ret_name {
    int a;
    int b;
    a = 10;

    else {
        a = 10;
        b = 24;
    }
}

def compute(int a): int trial {
}
```

invalid_use_binop.ql

```python
# Winnie Narang
def compute(): int ret_name_test {
    int test_int;
    ret_name_test = test_int + test_int;
}
```

mat_type.ql

```python
# Winnie Narang
def test_func(comp a, comp b, comp c, comp d): mat ret_val {
    mat x;
    mat f;
    x = [(a, b)(c, f)];
    ret_val = x;
}

def compute(): mat ret_val {
    comp a;
    comp b;
    comp c;
    comp d;
```
mixed_datatypes.ql

```python
# Winnie Narang

def func_test(int z):
    int ret_name {
        int a;
        comp b;
        int d;
        a = z;
        b = C(7.5*I);
        d = a+b*a+b/a-b;
        ret_name = d;
    }
def compute(int a):
    int trial {
        trial = func_test(35);
    }
```

no_compute.ql

```python
# Winnie Narang

def func_test(float z):
    float ret_name {
        float a;
        a = 5.8;
        ret_name = z;
    }
```

print_stmt.ql

```python
# Winnie Narang

def func_test(int z):
    int ret_name {
        int a;
        a = 5;
        a = z;
        ret_name = a;
    }
def compute(int a):
    int trial {
        print(a);
    }
```

un_op_adj.ql
def func_test(mat z) : mat ret_name {
    mat a;
    comp b;
    a =[(1,9,9)(4,5,5)];
    z = adj(b);
}
def compute(int a):int trial {
}

un_op_conj.ql

un_op_conj.ql

un_op_conj.ql

un_op_conj.ql

un_op_conj.ql

un_op_conj.ql

undec_func_call.ql
```python
ret_name = z;
}
def compute( int a): int trial {
    trial = func_test(4);
}
```

```python
unmatched_args.ql

# Winnie Narang
def func_test1(int z, int c): int ret_name {
    int a;
    int b;
    int d;
    a = z;
    ret_name = z;
}
def compute( int a): int trial {
    trial = func_test1(4);
}
```

```python
var_undeclared.ql

# Winnie Narang
def compute() : int ret_name_test {
    int test_int;
    ret_name_test = test_float;
}
```

**B.8.5 Output for failed cases**

test.out

```plaintext
#generated for test cases under SemanticFailures
Fatal error: exception Analyzer.Except("Invalid assignment to variable: val3")
Fatal error: exception Analyzer.Except("Invalid function declaration: func_test1 was already declared")
Fatal error: exception Parsing.Parse_error
Fatal error: exception Parsing.Parse_error
Fatal error: exception Analyzer.Except("Invalid matrix: incorrect type")
Fatal error: exception Analyzer.Except("Invalid assignment to variable: d")
Fatal error: exception Analyzer.Except("Missing 'compute' function")
Fatal error: exception Analyzer.Except("Invalid function call: incorrect type for parameter ")
Fatal error: exception Analyzer.Except("Invalid use of unop: 'Adj(expr)'")
Fatal error: exception Analyzer.Except("Invalid assignment to variable: b")
Fatal error: exception Parsing.Parse_error
Fatal error: exception Analyzer.Except("Invalid function call: func_test was not declared")
Fatal error: exception Analyzer.Except("Invalid function call: incorrect number of parameters")
```
B.9 C++ Helper files

B.9.1 qlang.cpp

```cpp
// Jonathan Wong
#include <Eigen/Dense>
#include <iostream>
#include <complex>
#include <string>
#include <cmath>
#include "qlang.hpp"

using namespace Eigen;
using namespace std;

MatrixXcf tensor(MatrixXcf mat1, MatrixXcf mat2) {
    int mat1rows = mat1.rows();
    int mat1cols = mat1.cols();
    int mat2rows = mat2.rows();
    int mat2cols = mat2.cols();
    MatrixXcf output(mat1rows * mat2rows, mat1cols * mat2cols);
    // iterates through one matrix, multiplying each element with the whole
    // 2nd matrix
    for (int m = 0; m < mat1rows; m++) {
        for (int n = 0; n < mat1cols; n++) {
            output.block(m * mat2rows, n * mat2cols, mat2rows, mat2cols) =
                mat1(m, n) * mat2;
        }
    }
    return output;
}

Matrix4cf control(Matrix2cf mat) {
    Matrix4cf output;
    output.topLeftCorner(2, 2) = IDT;
    output.topRightCorner(2, 2) = Matrix<complex<float>, 2, 2>::Zero();
    output.bottomLeftCorner(2, 2) = Matrix<complex<float>, 2, 2>::Zero();
    output.bottomRightCorner(2, 2) = mat;
    return output;
}

MatrixXcf genQubit(string s, int bra) {
    int slen = s.length();
    int qlen = pow(2, slen); // length of vector
    int base10num = 0;
    // iterates through qstr. Whenever digit is a 1, it adds the associated
```
//power of 2 for that position to base10num
const char * cq = s.c_str();
char * c = new char();
for(int i = 0; i < slen; i++) {
    strncpy(c,cq+i,1);
bases10num += strtol(c, NULL, 10) * pow(2,(slen-1-i));
}
delete c;

//creates the vector and sets correct bit to 1
MatrixXcf qub;
if(bra) {
    qub = MatrixXcf::Zero(1,qlen);
    qub(0,qlen-1-base10num) = 1;
} else if(!bra){
    qub = MatrixXcf::Zero(qlen,1);
    qub(base10num,0) = 1;
}
return qub;

string vectorToBraket(MatrixXcf qub) {
    int bra;
    int qlen;

    //determines whether bra or ket
    if(qub.rows() == 1) { qlen = qub.cols(); bra = 1; }
    else if(qub.cols() == 1) { qlen = qub.rows(); bra = 0;}
    else { //prints reg matrix if not row or column vector
        //cerr << "Incorrect matrix size for vectorToBraket* << endl;
        //exit(1);
        stringstream test;
        test << qub << endl;
        return test.str();
    }

    //gets position of 1 in the qubit
    complex<float> zero(0,0);
    int xi = 0;
    int yi = 0;
    int number;
    int index;
    string result;
    int count = 0;
    for(index = 0; index < qlen; index++) {
        if(bra) { xi = index; }
        else { yi = index; }

        if(qub(yi,xi) != zero) {
            //if(bra) { number = qlen-1-index; }
            //else { number = index; }
            number = index;

            //converts position to binary number reversed
            string bin = "";
do {
                if ( (number & 1) == 0 )
                    bin += "0";
                else
                    bin += "1";
                number >>= 1;
            } while ( number );
        }
    }

    return bin;
}
\[
\text{int outQubLen} = \text{sqrt}(\text{qlen});
\]

//adds necessary 0s
for (int i = bin.length(); i < outQubLen; i++) {
    bin += "0";
}
reverse(bin.begin(), bin.end()); //reverses

ostringstream convert;
float re = qub(yi, xi).real();
float im = qub(yi, xi).imag();
string oper = "*";
string rstr = ";
string istr = ";

//adds constant expression
convert << "("
if (re != 0) { convert << re; }
if (re != 0 && im != 0) { convert << "+"; }
if (im != 0) { convert << im << ";i"; }
convert << ");";

//cleans up (1) and (1i) cases
string constant = convert.str();
if (constant.compare("(1)") == 0) { constant = ";"; }
else if (constant.compare("(1i)") == 0) { constant = ";i"; }

//generates appropriate bra or ket representation
string qubstr;
if (bra) { qubstr = constant + "<" + bin + ";"; }
else { qubstr = constant + ";" + bin + ";>"; }
if (count > 0) {
    result += " + " + qubstr;
} else { result = qubstr; }
count++;
}
return result;

B.9.2 qlang.hpp

//Jonathan Wong
#ifndef QLANG_HPP_
#define QLANG_HPP_

using namespace Eigen;
using namespace std;

//CONSTANTS
const Matrix2cf H = (Matrix2cf() << 1/sqrt(2), 1/sqrt(2),
    1/sqrt(2), -1/sqrt(2)).finished();
const Matrix2cf IDT = Matrix2cf::Identity();
const Matrix2cf X = (Matrix2cf() << 0, 1, 1, 0).finished();
const Matrix2cf Y = (Matrix2cf() << 0, std::complex<float>(0,1),
    std::complex<float>(0,1), 0).finished();
const Matrix2cf Z = (Matrix2cf() << 1, 0, 0, -1).finished();

`
//METHODS
MatrixXcf tensor (MatrixXcf mat1, MatrixXcf mat2);
Matrix4cf control (Matrix2cf mat);
MatrixXcf genQubit (string s, int bra);
MatrixXcf genQubits (string s);
string vectorToBraket (MatrixXcf qub);

#endif