QuantC: A C-Inspired Language for Quantum Computing

Tyler Burkett

Demo

Quantum Computing: Qubits

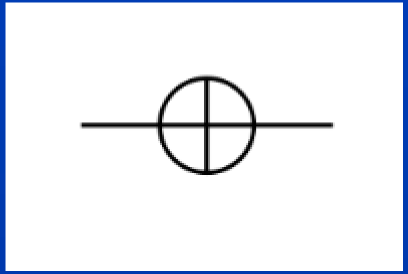
- *Qubits* are like bits, but are based on some quantum phenomenon with a binary set of states when measured.
- Can store 0 and 1 like a regular bit, but can also be in a *superposition* of 0 and 1.
- Measuring a qubit causes it to *collapse* into either a 0 or 1 if it’s in a superposition.

Quantum Computing: Gates and Measuring

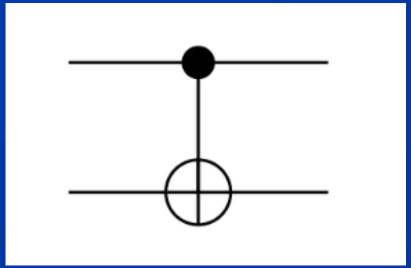
- A common representation of computations on qubits is as a circuit, with operations represented as *quantum logic gates.*
- Quantum circuits are just as computationally powerful as classical circuits.
- However, quantum circuits do have some constraints classical versions don't have.
    - Gates/circuits must be reversible; at minimum, the # outputs = # inputs.
    - Gates can't be used to copy or delete arbitrary quantum states.

Examples of Quantum Gates

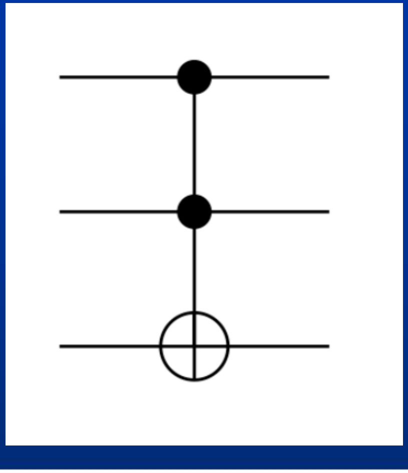
Classically Equivalent Gates



NOT (a.k.a X-gate)

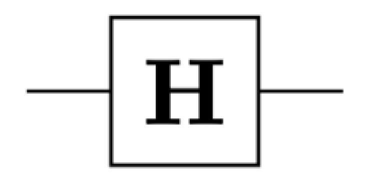


XOR
(a.k.a Controlled NOT, CNOT)

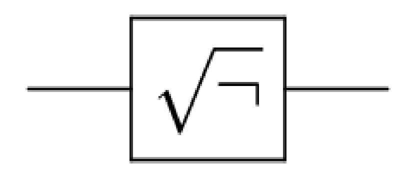


AND (a.k.a Toffoli gate, CCNOT)

Purely Quantum Gates



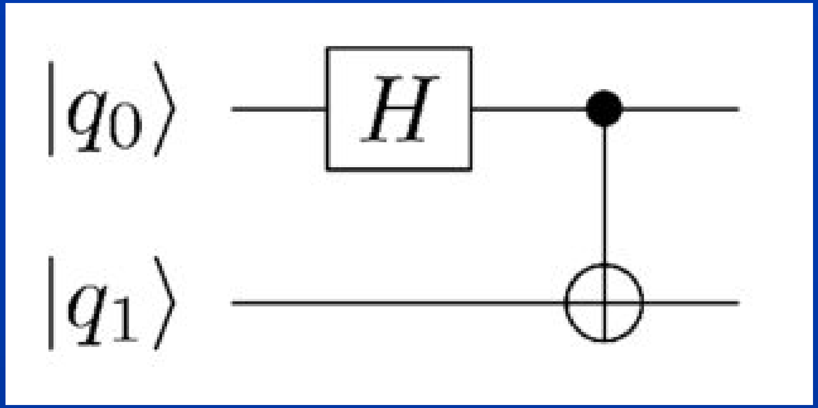
Hadamard gate



$\sqrt{NOT}$ gate

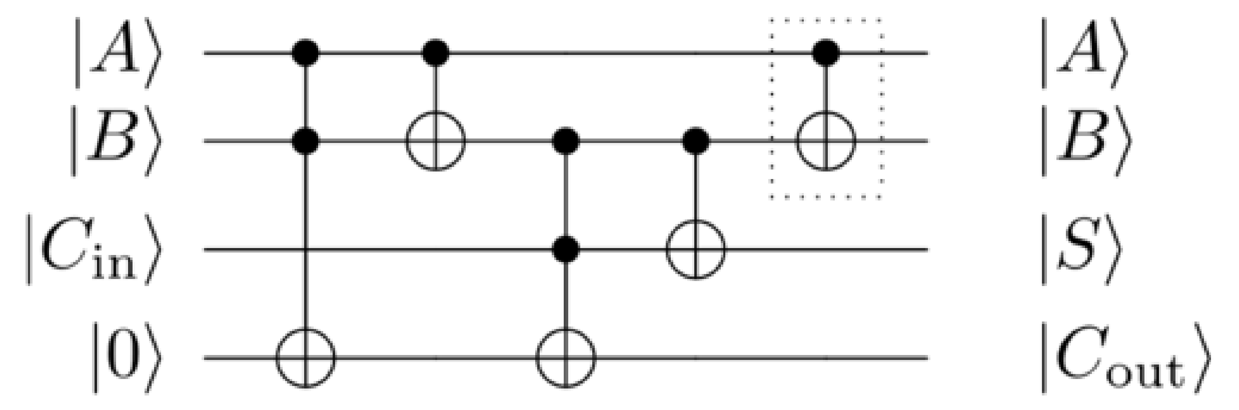
Simple Circuit Examples

Entanglement Circuit



$$|00\rangle \rightarrow \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Full Adder

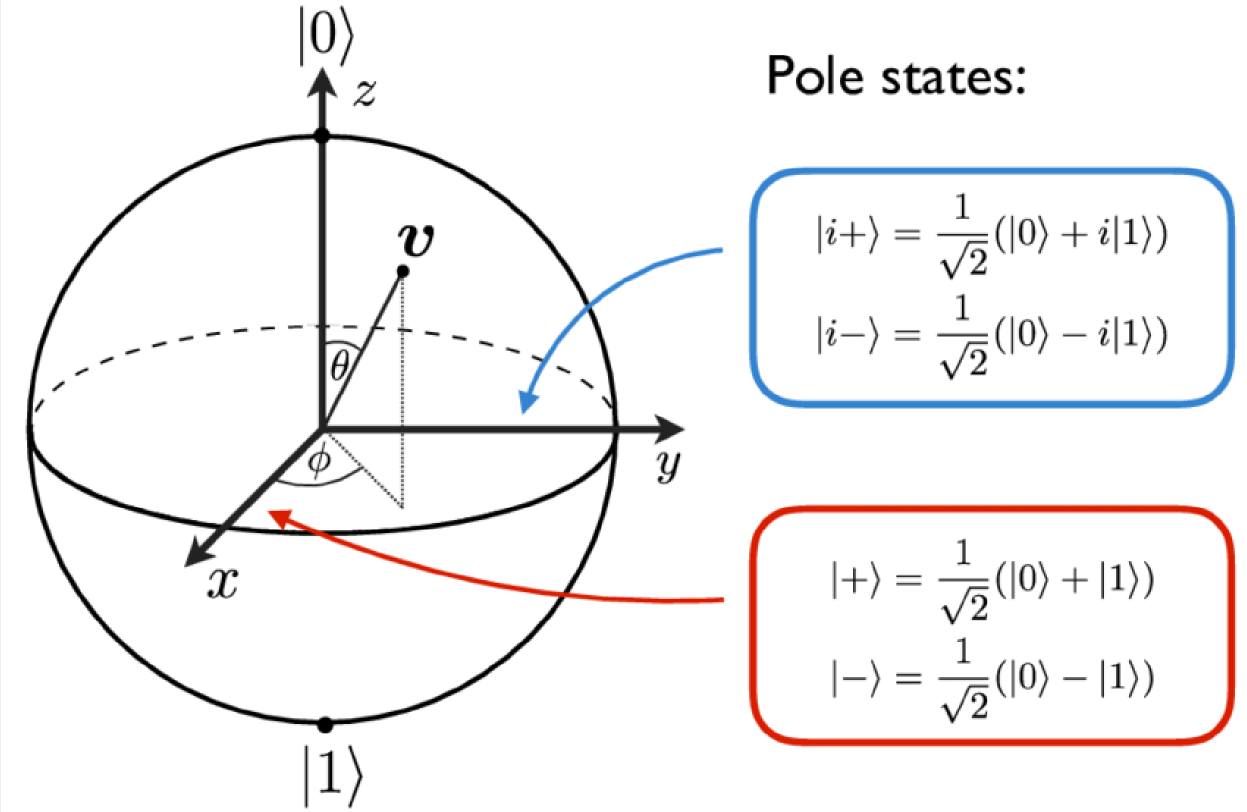


Crash Course in Quantum Mechanics

- Value of a qubit is represented by a normalized vector in $\mathbb{C}^2$
  - First coefficient is always taken to be real
- Gates are unitary matrices in $\mathbb{C}^{2^n \times 2^n}$
  - Unitary: $UU^\dagger = U^\dagger U = I$
- Measurement corresponds to randomly picking a base state, where the probability of picking it is based on the coefficient in the qubit vector

$$|\psi\rangle = \sin\left(\frac{\theta}{2}\right)|0\rangle + \cos\left(\frac{\theta}{2}\right)e^{i\phi}|1\rangle$$

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



Pole states:

$$|i+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$$

$$|i-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$$

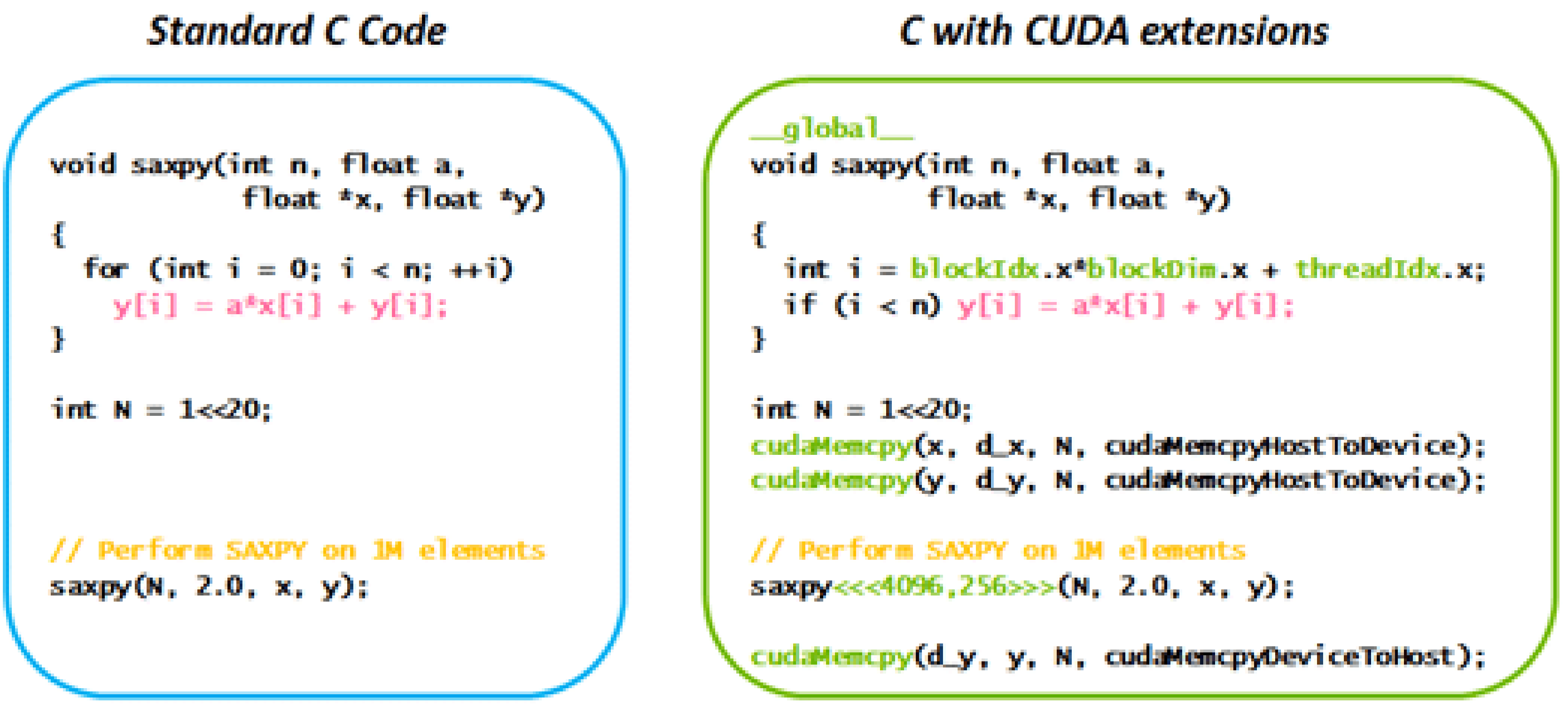
$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Design Goals of QuantC

- Create a language with syntax to support creating programs with both classical and quantum components.
- Minimize the overhead for learning the language.
    - Extend an existing language, rather than start from scratch.
    - Make distinguishing the two parts of code clear.
    - Avoid overloading the meaning of existing symbols for quantum code.
- Abstract communication between the classical and quantum computers.

CUDA as an Inspiration for Extending C



Standard C Code

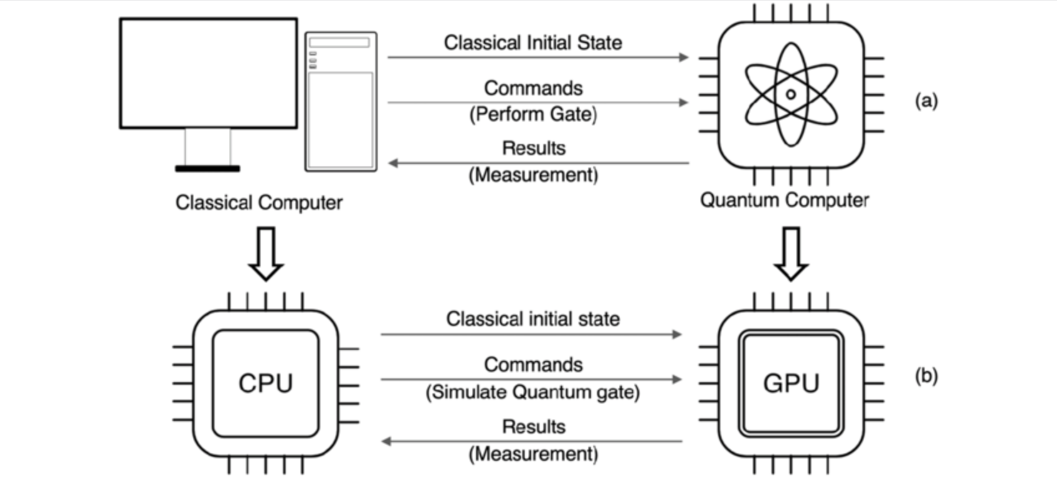
```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

C with CUDA extensions

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

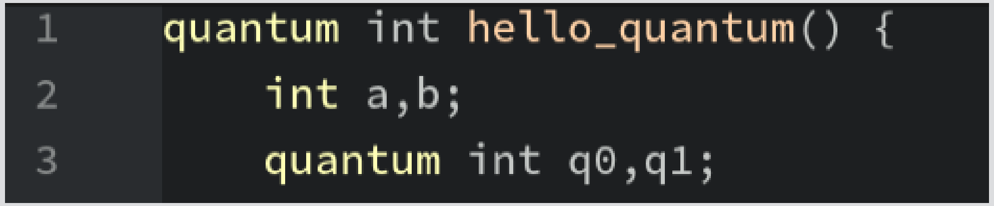
Assumptions about Architecture

- The quantum computer operates as a coprocessor to the classical computer.
- There is no shared memory between the two computers.
- Quantum computer contains some classical memory and can perform some classical instructions.
- The two computers communicate via a synchronous communication channel.



Storage Classifier: `quantum`

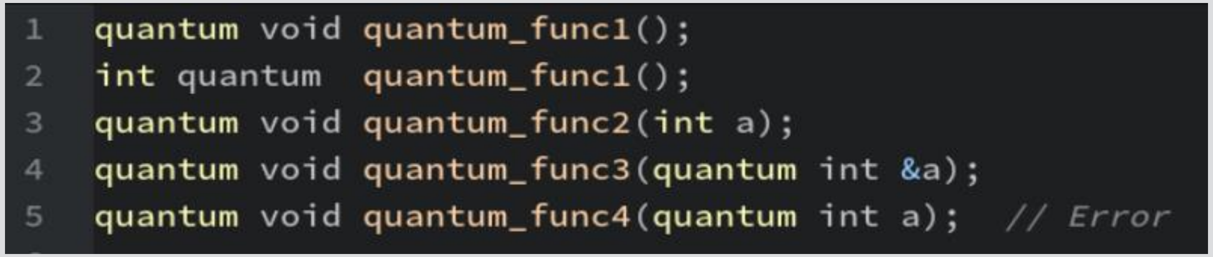
- The keyword `quantum` applied to functions distinguishes classical code and quantum code. This keyword is a *storage class specifier* in the C grammar.
- *Quantum functions* and *quantum variables* refer to those constructs with this keyword applied to them.
- Quantum variables can only appear inside quantum functions.



```
quantum int hello_quantum() {
    int a,b;
    quantum int q0,q1;
```

Semantics of Quantum Function

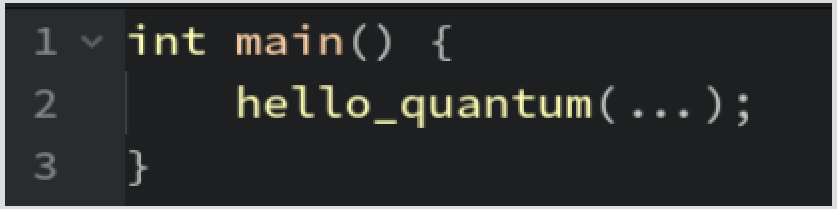
- A quantum function can take both classical and quantum values as arguments
- Quantum arguments necessarily use pass-by-reference semantics, denoted with an & like in C++. QuantC reserves this syntax for pass by reference for quantum arguments.
- Quantum functions cannot contain recursive calls and must have a definition.
- Quantum functions also only permit built-in types.
- Otherwise, they follow the same grammatical and semantical rules as classical functions.



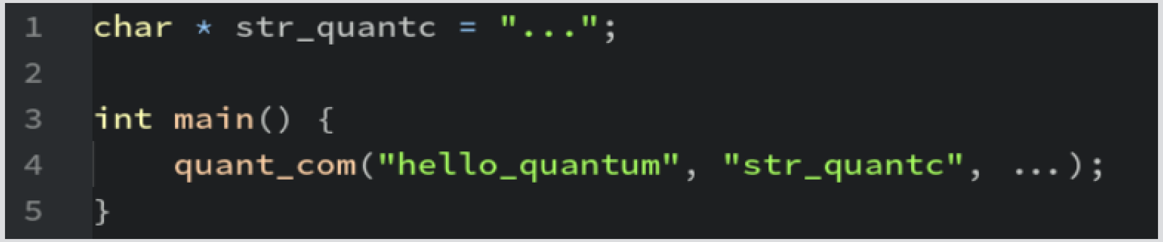
```
quantum void quantum_func1();
int quantum  quantum_func1();
quantum void quantum_func2(int a);
quantum void quantum_func3(quantum int &a);
quantum void quantum_func4(quantum int a);   // Error
```

Calling Quantum Functions

- When calling a quantum function in classical code, the compiler substitutes it with a call to a function to transmit the compiled quantum code to the quantum computer.
    - The quantum *intermediate representation* (IR) is stored inside the classical code and transmitted to the quantum computer when needed.
    - It's assumed that any additional translation from the quantum IR to quantum machine instructions is handled by the quantum computer itself.
- Only functions with exclusively classical arguments with pass-by-value semantics can be called from classical code.



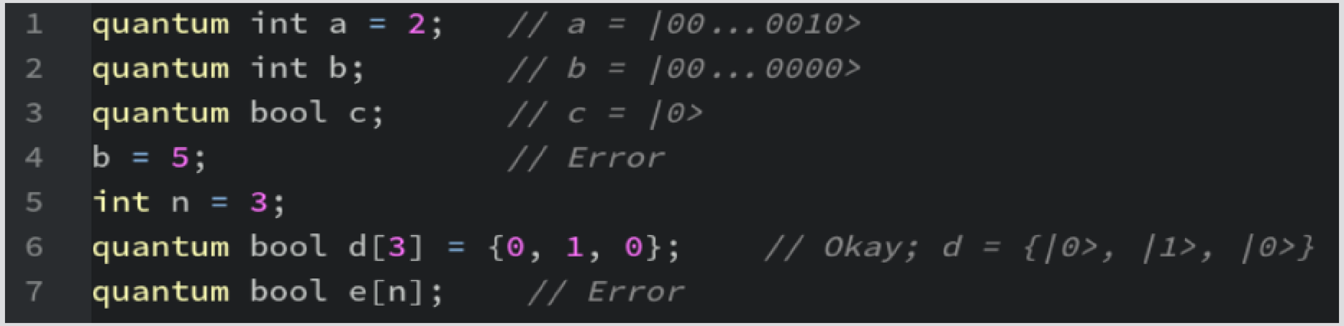
```
int main() {
    hello_quantum(...);
}
```



```
char * str_quantc = "...";

int main() {
    quant_com("hello_quantum", "str_quantc", ...);
}
```

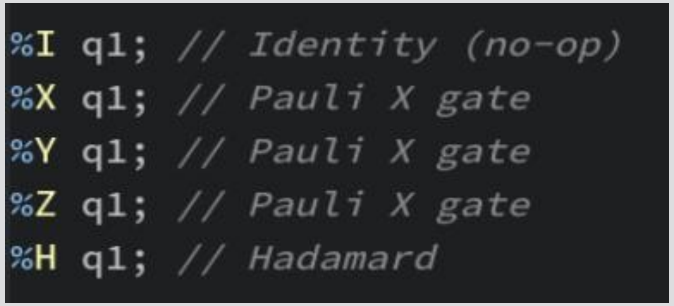
Semantics of Quantum Variables

- Quantum variables can only be assigned a value at declaration.
    - Classical expression initialize the corresponding qubits to one of the base state. The i_{th} bit of the expressions initializes the i_{th} qubit.
    - Uninitialized variables are 0 initialized.
- Quantum arrays must have a compile-time constant size.
    - Initialization of quantum arrays with initializer lists follow the same rule as other classical expressions for each entry.

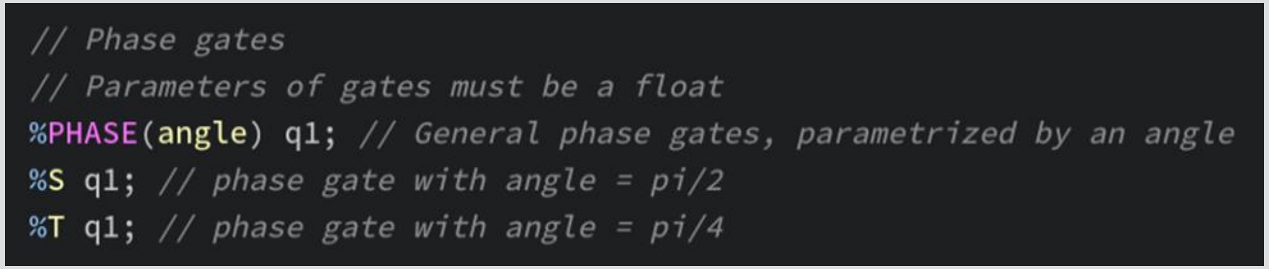


```
quantum int a = 2;    // a = |00...0010>
quantum int b;        // b = |00...0000>
quantum bool c;       // c = |0>
b = 5;                // Error
int n = 3;
quantum bool d[3] = {0, 1, 0};    // Okay; d = {|0>, |1>, |0>}
quantum bool e[n];     // Error
```

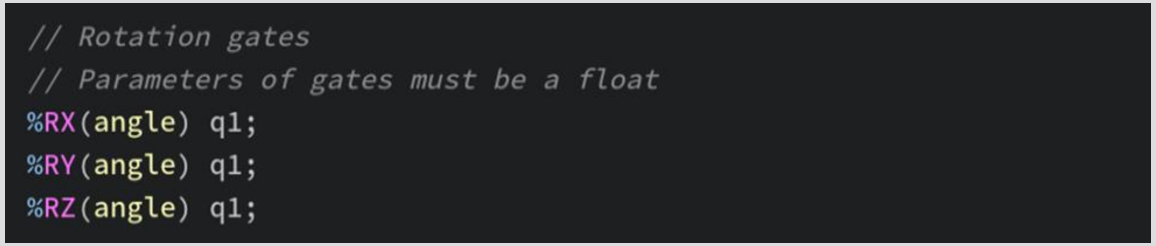
Quantum Operations (Unary)



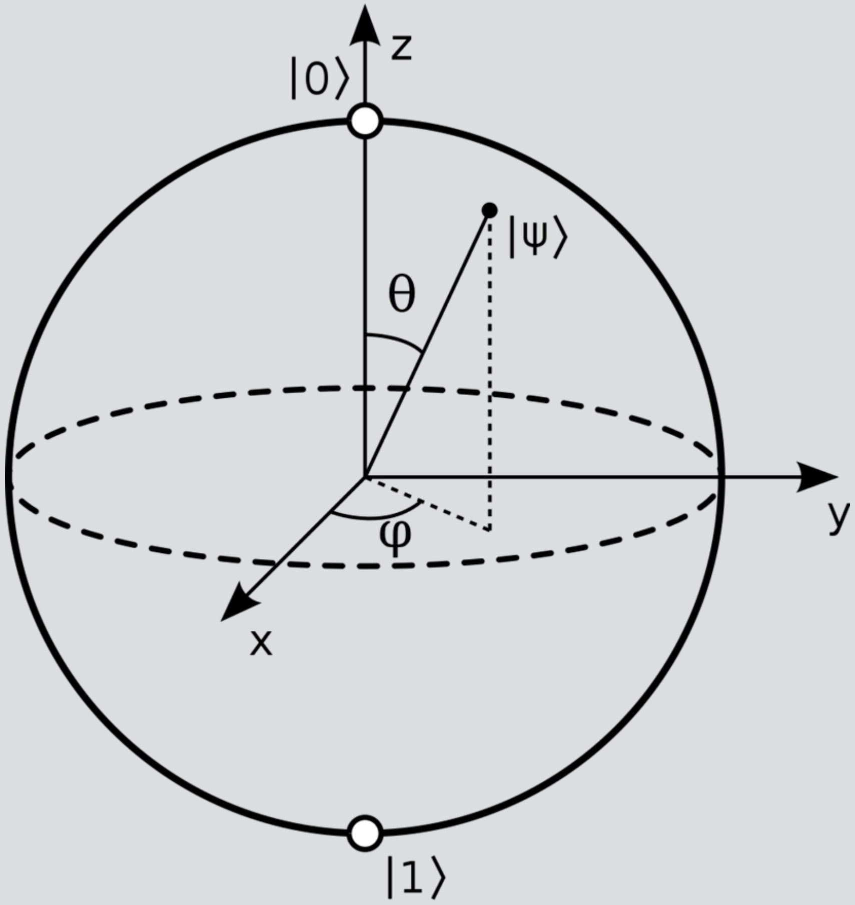
```
%I q1; // Identity (no-op)
%X q1; // Pauli X gate
%Y q1; // Pauli X gate
%Z q1; // Pauli X gate
%H q1; // Hadamard
```



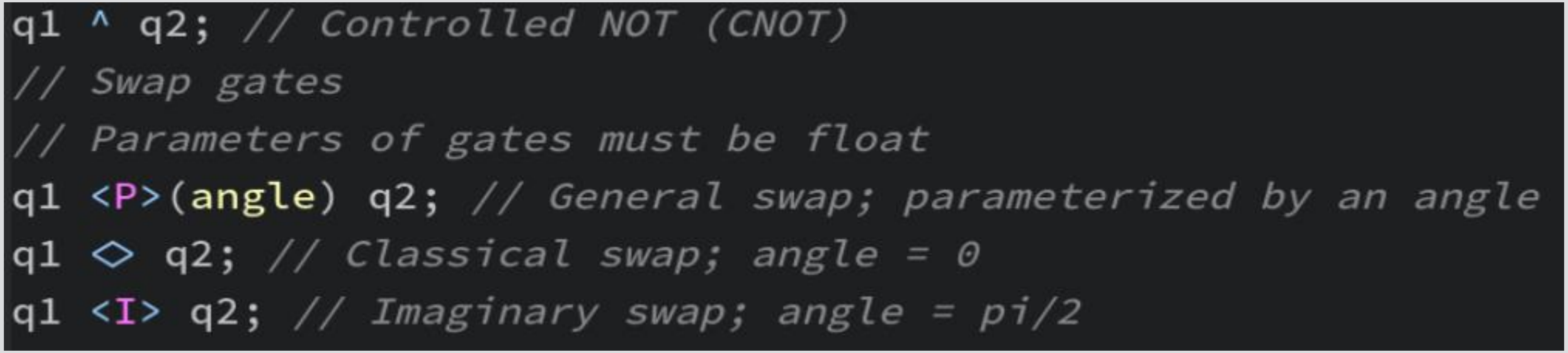
```
// Phase gates
// Parameters of gates must be a float
%PHASE(angle) q1; // General phase gates, parametrized by an angle
%S q1; // phase gate with angle = pi/2
%T q1; // phase gate with angle = pi/4
```



```
// Rotation gates
// Parameters of gates must be a float
%RX(angle) q1;
%RY(angle) q1;
%RZ(angle) q1;
```



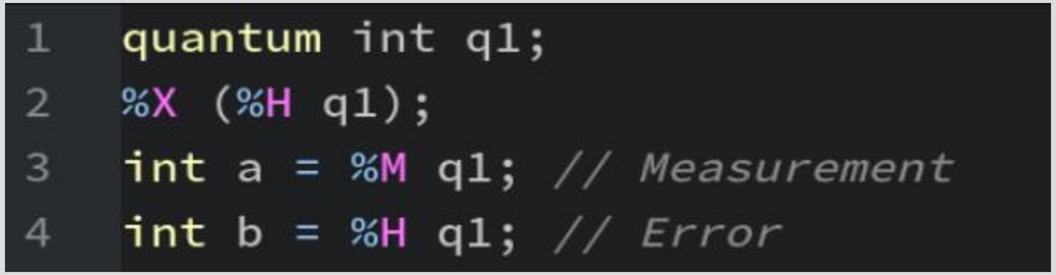
Quantum Operations (Binary)



```
q1 ^ q2; // Controlled NOT (CNOT)
// Swap gates
// Parameters of gates must be float
q1 <P>(angle) q2; // General swap; parameterized by an angle
q1 <> q2; // Classical swap; angle = 0
q1 <I> q2; // Imaginary swap; angle = pi/2
```

Quantum Expressions

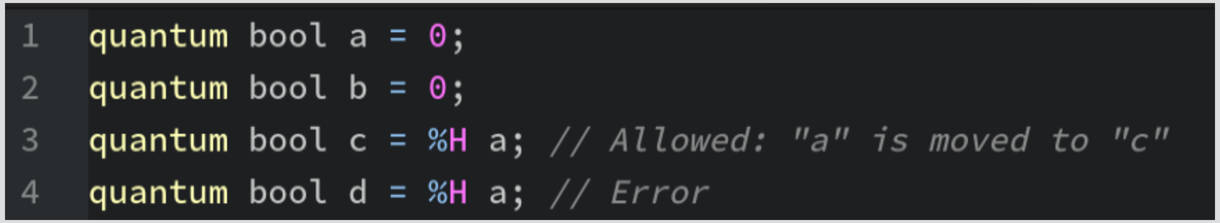
- A quantum expression is a combination of quantum variable identifiers and operations.
    - These correspond to gates applied to qubits.
    - These gate applications occur as side effects of the expressions; there is no need to assign the expression to another variable.
- Measurement operations and calls to quantum functions are the only expressions which can take quantum expressions and result in a classical value
    - All others quantum operations take a quantum expression and result in another.



```
quantum int q1;
%X (%H q1);
int a = %M q1; // Measurement
int b = %H q1; // Error
```

Declarations using Quantum Expressions

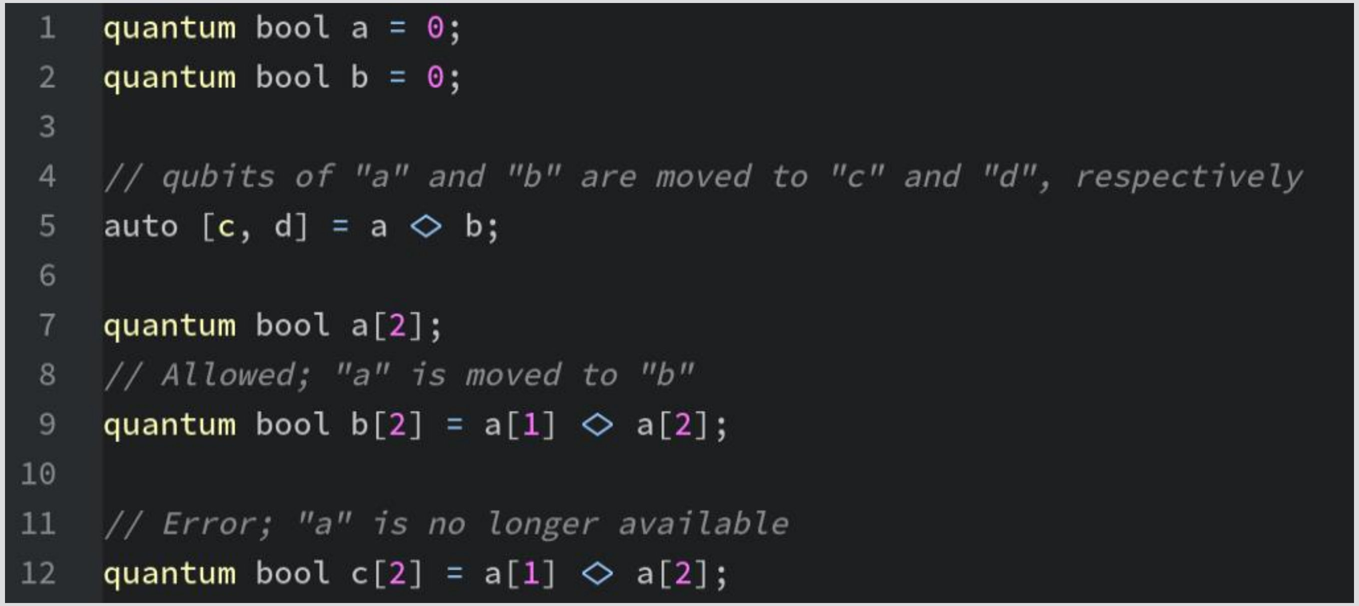
- Quantum expression can also be used as initializers, but they follow a move semantics instead.
  - Future statements cannot reference quantum variable inside of such an initializer. The same is true for other expressions inside the same initializer.
  - The initialized declaration takes ownership of the qubits of the quantum variable inside the expression.
  - The declared declaration must have the same type as the quantum variable in the initializer.



```
quantum bool a = 0;
quantum bool b = 0;
quantum bool c = %H a; // Allowed: "a" is moved to "c"
quantum bool d = %H a; // Error
```

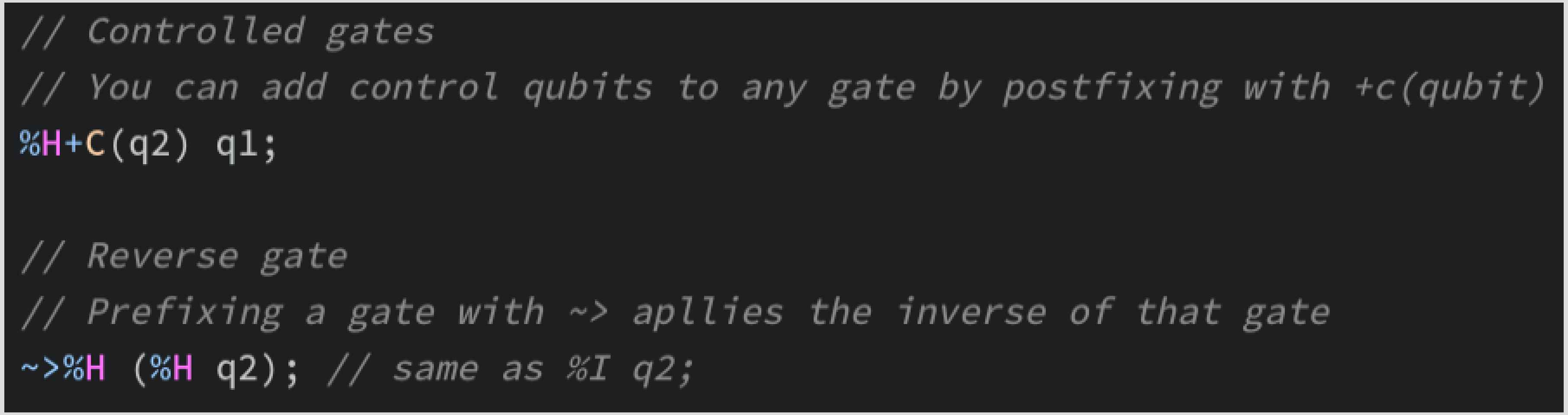
Declarations using Quantum Expressions (Cont.)

- QuantC adds a structured binding assignment to declare new variables for quantum expressions with multiple quantum variables.
- Each instance of indexing a quantum arrays counts as a unique identifier, but the corresponding declaration takes ownership of the entire array.



```
quantum bool a = 0;
quantum bool b = 0;

// qubits of "a" and "b" are moved to "c" and "d", respectively
auto [c, d] = a ◇ b;

quantum bool a[2];
// Allowed; "a" is moved to "b"
quantum bool b[2] = a[1] ◇ a[2];

// Error; "a" is no longer available
quantum bool c[2] = a[1] ◇ a[2];
```

Quantum Operations (Modifiers)

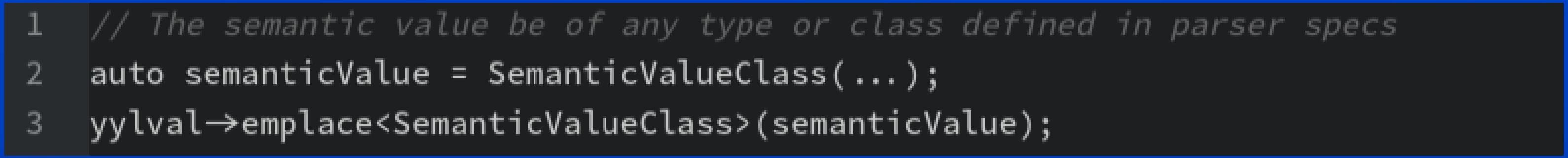


```
// Controlled gates
// You can add control qubits to any gate by postfixing with +c(qubit)
%H+C(q2) q1;

// Reverse gate
// Prefixing a gate with ~> apllies the inverse of that gate
~>%H (%H q2); // same as %I q2;
```

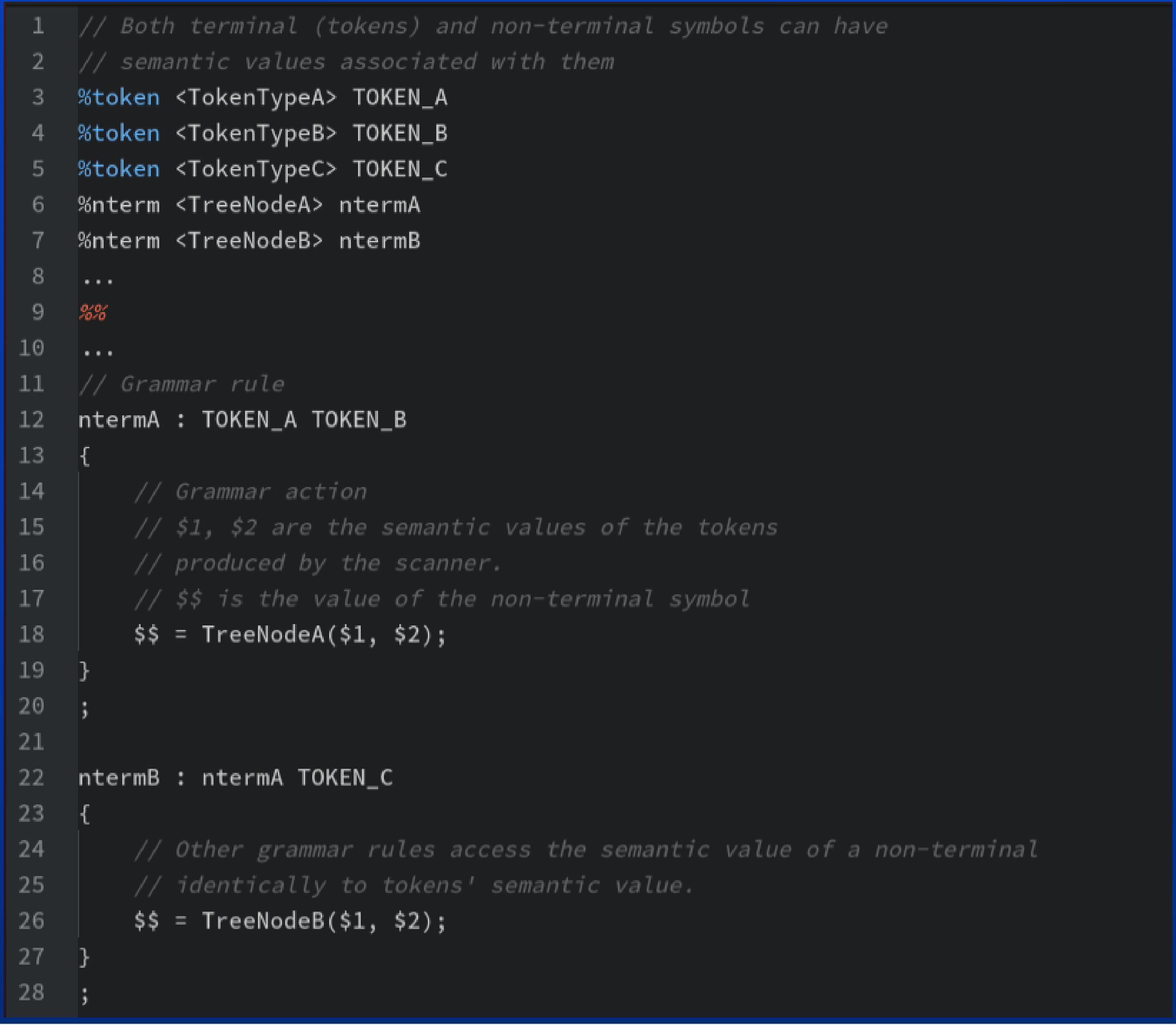
Implementation

Preprocessing, Lexical Analyzer, and Parser

- The compiler calls `gcc` to handle preprocessing.
  - QuantC does not modify preprocessor directives.
- I generate the scanner and parser with Flex and Bison, respectively
  - I define the token regular expressions and actions in a Flex scanner description file.
  - I define the grammar rules and actions in a Bison grammar file.
  - These files are modified versions of Lex and Yacc files from Jutta Degener.
- Both the scanner and parser include options to generate them as C++ code rather than the default C
  - This choice of language enables the scanner and parser to take advantage of Bison’s version of variants, which allow for near arbitrary semantic values to pass between the two through a common interface.



```
// The semantic value be of any type or class defined in parser specs
auto semanticValue = SemanticValueClass(...);
yylval->emplace<SemanticValueClass>(semanticValue);
```



```
// Both terminal (tokens) and non-terminal symbols can have
// semantic values associated with them
%token <TokenTypeA> TOKEN_A
%token <TokenTypeB> TOKEN_B
%token <TokenTypeC> TOKEN_C
%nterm <TreeNodeA> ntermA
%nterm <TreeNodeB> ntermB
...
%%
...
// Grammar rule
ntermA : TOKEN_A TOKEN_B
{
    // Grammar action
    // $1, $2 are the semantic values of the tokens
    // produced by the scanner.
    // $$ is the value of the non-terminal symbol
    $$ = TreeNodeA($1, $2);
}
;

ntermB : ntermA TOKEN_C
{
    // Other grammar rules access the semantic value of a non-terminal
    // identically to tokens' semantic value.
    $$ = TreeNodeB($1, $2);
}
;
```

Abstract Syntax Tree (AST)

- I implement the AST as a set of C++ classes
  - I take advantage of class inheritance to create categories of nodes that mirror QuantC’s grammar
- The constructors of these nodes take smart pointers to other nodes to form the AST
  - Smart pointers ease memory management compared to regular pointers and manually creating and deleting nodes.
  - These pointers alongside the class inheritance allow nodes to access the methods of a range of different subnodes through a generic pointer.

Abstract Syntax Tree (AST) (Cont.)

- The base AST node defines static methods and member used by other nodes in later steps of compilation
    - This node also defines the interface nodes interact with to perform later steps.
    - These static items include a symbol table and helper methods for semantics check, along with LLVM data structures for code generation.
- The base AST nodes also implements other components to take advantage of a class introspection feature provided by the LLVM library.
    - All of the other AST nodes use this feature to perform specific actions based on subnodes' type information during semantics checking and code generation.

Semantics Checking

- AST nodes define `checkSemantics` methods to handle semantics checking.
  - Each method involves recursively calling the semantic checking method on its subnodes, then using the information generated from that to perform additional checks if needed.
- There are two main mechanisms for information about semantics to flow between nodes.
  - For the common case of information related to identifiers, the static symbol table stores their type and kind information.
  - For other cases, such as blocks checking return statements, nodes implement additional public members to handle passing information between themselves.

Code Generation (Classical Code)

- AST nodes define a `codegen` method to handle code generation.
  - These methods use the LLVM library to handle most of the work.
  - The static LLVM structures in the base node maintain contextual information, collect instructions to create a complete module, and provide an interface to create and insert new instructions into the module.
- The output of code generation is LLVM IR
  - This IR is architecture independent.
  - This IR is formatted into static single-assignment. Though not currently used, this format enables applying further optimizations to the final output.
  - Once generated, the compiler uses a provided IR compiler `llc` to convert this into an object file.

Code Generation (Quantum Code)

- AST nodes also define a `quantum_codegen` method to handle code generation for quantum functions.
  - When generating code for quantum functions, the `codegen` method for a function instead creates a separate builder for the quantum IR.
  - The function passes its own name and type information to the builder.
  - When ready, the function node calls the `quantum_codegen` method of its body node and passes the builder to the method by reference.
  - Once the body node returns, the function node calls a method on the builder to generate a string containing the quantum IR.
- After all the quantum functions are generated, the collection of quantum IR strings are stored as a single string literal inside the LLVM module.
  - For this compiler, the output IR is Quil.
  - Quil is another architecture independent IR which the compiler assumes can be interpreted by the quantum computer

Handling Classical Calls to Quantum Functions

- When classical code generation reaches a function call, the function call node check to see if the function is a quantum function
- If so, the call expression generates a call to a helper function `quant_com.`
    - This function takes the following
        - The name of the quantum function
        - The quantum IR string
        - The number of qubits allocated by the quantum functions
        - Any arguments passed to the quantum function
- This function handles packaging the IR and a call to the necessary function in a message to transmit to the quantum computer
    - If the IR is successfully executed, the helper function parses the resulting message to create the necessary return value.

Linker

- The compiler also calls `gcc` to handle linking the generated object file into an executable.
- Since preprocessing and linking are handled by `gcc`, the compiler can generate programs that include code from the standard C library.

Testing

- Testing was done ad hoc.
  - I used a combination of my own programs and regular C programs from an online repository to form the set of test programs.
  - Testing consisted of manually compiling and running these programs with my compiler to verify the output match expectations.
- For the quantum computer, I installed the Riggetti Computing ForestSDK to run the provided quantum virtual machine (QVM) on my local machine.
  - The QVM runs as a local server with an HTTP API on a localhost port.
  - The `quant_com` function currently handles communicate with this setup only, using cURL and JSON libraries to handle communications.

Final Demo

State of Project

- The QuantC compiler is partially complete
    - The demo programs depict the primary structures the compiler can handle.
    - The scanner and parser have complete definitions for their tokens and grammar rules, I have not implemented all of the necessary actions to completely parse any arbitrary program.
    - Semantics checking is also minimal, primarily consisting of determining type and control flow information needed in code generation.

Future Work

- Next Steps
    - Finish implementing the compiler for QuantC as defined.
    - Integrate asynchronous channels into QuantC.
    - Determine a method for linking quantum functions.
    - Consider adding syntax to combine or split quantum variables when declaring new variables.
    - Consider adding a keyword to mark quantum functions without classical code that can work with operation modifiers.