Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Implementetion of a Compiler from QASM to QScript

Relatore:
Chiar.mo Prof.
UGO DAL LAGO

Presentata da:
OLGA BECCI

III Sessione
a.a. 2016/2017

Introduction

Today there are computational problems that are still considered unsolvable in a practical time frame. Some of these problems are more theoretical, such as integer factorization and discrete logarithm, some are more practical, like problems linked to material science science and chemistry, but they all have one thing in common: the bigger the input, the harder it is to work out for classical machines. It does not seems possible that we will ever be able to resolve these problems efficiently with the current computational systems.

Quantum computing is the key to solve these problems in a manageable time frame. Quantum computation means to compute using quantum mechanical properties. The idea is to have a new kind of bit, called *qubit*, that can assume the values 1, 0, and both *at the same time*. This is possible thanks to a quantum mechanical property known as *superposition*, that allows to follow every possible paths of a computation at once. This is what brings about the speed-up needed to be able to handle the aforementioned problems [9].

If we have qubits but we still use classical operators, we would not be able to benefit from the advantages offered by quantum computing. That is why we will need quantum operators, or quantum gates. With a combination of both qubits and quantum gates it is possible, in theory, to build entirely a quantum machine.

It has already been suggested that the speed-up offered by the quantum paradigm could open new doors in the fields of of chemistry [8], optimization [13] and machine learning [16].

It is important to notice that quantum computing brings solutions only to specific problems that were considered computationally infeasible, not completely unsolvable. This complies with the Church-Turing conjecture, since the problems that were unsolvable under classic computation stay so under quantum computation.

The day that everyone will have a quantum computer in their house is still far away, and may never come, but it is possible for anyone to start experimenting with quantum computing through some platforms that offer access to simple simulations or small quantum devices.

One of such platforms is IBM's *quantum experience* [19]. Through this platform, IBM gives access to different devices: a 5 qubits machine, a 16 qubits machine and a 20 qubits machine. The latter is the newest and is at this day being tested, but for the other two devices IBM offers free access through the cloud, using QISKit [20] a software development kit or, for the 5 qubits device, using directly the online platform. The access to the device is controlled by a credit system, so when running a program, the result is not given right away. The program is placed in a queue and will be executed when the machine is free. To give the users the possibility of experimenting with quantum computing even if the machine is busy, they offer a simulator where one can run any number of programs without limitations or waiting for the result. To communicate with both the simulator and the devices IBM developed QASM [3], a language that allows to implement quantum gates and to perform simple classic operations.

Another platform that can be used to experiment with quantum computing is Google's *quantum playground* [18]. It is a simulator that offers an environment to implement quantum programs with QScript, a dedicated high level language that allows to implement complex quantum algorithms. The platform provides also a graphical environment that gives a visual representation of each step of the computation of an algorithm.

Even though the two platforms have the same purpose, they are quite different. With the *quantum experience* it is possible to run a program and have back the results while in the *quantum playground* it is possible to see how the computation evolves step by step.

As said before, each of these platforms uses a dedicated language (QASM and QScript). This might be a setback for who is familiar with just one of the two languages. A compiler between the two language could bring different advantages.

- The graphical environment offered by *quantum playground* could help in reaching a better understanding of a QASM program, before running it on IBM's real machine.
- The compiler opens the possibility to compare the two simulators and see which approach is more efficient or more accurate.
- The *quantum playground* offers a simulator with 22 qubits while IBM's simulator offers up to 16 qubits. It would be possible to run QASM program that take advantage of the increased numbers of qubits.
- It could allow some interoperability between the two platforms. In particular, it could be possible to define new gates and circuits in QASM and then integrate them in a more complex QScript program.

In this optic we present the following work of a compiler that allows to translate QASM programs in QScript programs. In particular:

- in the first chapter we will give an introduction to quantum computing and a brief explanation of some important quantum theory concepts;
- in the second chapter we will talk about the architecture of a quantum machine and quantum languages;
- in the third chapter we will focus on the QASM language;
- in the fourth chapter we will focus on the QScript language;
- in the fifth chapter we will outline the implementation and functionality of the QASM-QScript compiler.

Contents

List of Figures

Chapter 1

Quantum Computing

Can we simulate physics using computers? This is the question posed by Richard Feynman [4], which can be seen as the start of quantum computing.

Feynman says that, while it is easy to simulate classical physics giving a discrete representation of time and space, the same cannot be done with quantum mechanical phenomena.

The issue lies in the underlying complexity. Feynman describes the problem by first working with probabilities since quantum phenomena are governed by probabilities.

> "If we had many particles, we have R particles, for example, in a system, then we would have to describe the probability of a circumstance by giving the probability to find these particles at points $x_1, x_2, \dots, x_R$ at the time t. That would be a description of the probability of the system. And therefore, you'd need a k-digit number for every configuration of the system, for every arrangement of the R values of x. And therefore if there are N points in space, we'd need N^R configurations."

This shows that the complexity of simulating a probabilistic behavior grows exponentially. With a quantum mechanical event the situation is similar, but, instead of probability, we talk about amplitudes:

> "But the full description of quantum mechanics for a large system with R particles is given by a function $\psi(x_1, x_2, \ldots, x_R, t)$ which we call the amplitude to find the particles $x_1, \ldots, x_R$, and therefore, because it has too many variables, it cannot be simulated with a normal computer with a number of elements proportional to R or proportional to N."

Feynman then goes on proposing two different approaches to simulate quantum systems: using a classical computer or building the computer itself with quantum elements which obey quantum laws.

The first option is unfeasible since, even though it is possible to efficiently simulate a probabilistic machine in a classical machine through some adjustments, the probabilistic approach still leaves out some important quantum mechanic properties, such as interference which is given, in a quantum system, by the possibility to have negative probabilities.

The solution seems to be the second approach: building the whole computer with quantum mechanical elements. This conclusion leads to the idea that we should not think about the physics of representing information and the information itself as separate things. In this optic, researchers started talking about *quantum computing*. It is in fact nothing more than computing using quantum-mechanical phenomena, in particular exploiting quantum mechanic properties such as entanglement and superposition.

Aside from the possibility of simulating quantum phenomena accurately, quantum computing can be used to provide a considerable speed up in computations. The proof of the possible speedup is given by Shor's algorithm for computing discrete logarithms and factoring [9]. Shor was able to create *quantum* algorithms able to resolve the above problems in polynomial time, which is a level of efficiency not known to be possible to reach on a classical machine, so much so that most of common cryptography is based on the assumption that the resolution of these problems is not computationally feasible.

1.1 The Basics of Quantum Theory

We assume some familiarity with the algebra and geometry of complex numbers and with complex vector spaces.

Given that ϕ is a quantum particle, and $x_0, \ldots, x_{n-1}$ its possible positions in space, we can associate an n-dimensional column complex vector to the particle to represent its state.

By using the Dirac ket notation, we use $|x_i\rangle$ to mean that the particle can be found in position i:

$$\begin{aligned} &|x_0\rangle \longmapsto [1, 0, \ldots, 0]^T \\ &|x_1\rangle \longmapsto [0, 1, \ldots, 0]^T \\ &\vdots \\ &|x_{n-1}\rangle \longmapsto [0, 0, \ldots, 1]^T \end{aligned} \tag{1.1}$$

These are the basic states of a complex vector space $\mathbb{C}^n$, but every vector in $\mathbb{C}^n$ can represent a legitimate state of the particle. The generic state $|\phi\rangle$ is given by a linear combination of the basic states in 1.1.

$$|\phi\rangle = c_0 |x_0\rangle + \ldots + c_{n-1} |x_{n-1}\rangle \qquad c_0, \ldots, c_{n-1} \in \mathbb{C} \tag{1.2}$$

Where $c_0, \ldots, c_{n-1}$ are called *complex amplitudes*, and can be used to compute the probability $p(x_i)$ to find the particle in position x_i after a measurement[1].

Until it is indeed measured, the particle can possibly be in all the positions $|x_0\rangle \ldots |x_{n-1}\rangle$ at once. We say that ϕ is in a *superposition* of states and the amplitudes tell us exactly which superposition the particle is in.

We have established what a state is, now we have to see how the vector changes through time. At the heart of computation is the evolution of a

[1] In a vector $k|\phi\rangle$, k is a complex factor that changes its length, but the vector $|\phi\rangle$ and all its complex scalar multiples $c|\phi\rangle$ describe the same physical space. This is why we can work with *normalized* $|\phi\rangle$, i.e. $\frac{|\phi\rangle}{||\phi\rangle|}$, that has length 1. Given a normalized state, the probability to find a particle in the position x_i is $p(x_i) = |c_i|^2$

system from state to state, namely its *dynamics*. In a quantum system the dynamics are given by *unitary operators*.

If U is an unitary matrix that represents a unitary operator and $|\phi(t)\rangle$ is the state of a system at time t , we have that:

$$|\phi(t+1)\rangle = U\,|\phi(t)\rangle \tag{1.3}$$

If we have a series of unitary operators $U_0, U_1 \ldots U_{n-1}$ we'll have that:

$$|\phi(t+1)\rangle = U_{n-1} \ldots U_1 U_0\,|\phi(t)\rangle \tag{1.4}$$

From a quantum system we can observe a set of physical quantities, such as position, momentum and spin. These physical quantities are the *observables* of the system. Each observable can be thought of as a question: given that the system is in a state $|\phi\rangle$, which values can be observed?

Mathematically, we see that each observable corresponds to an *hermitian operator*. The *eigenvalues* associated with an observable are the only values that can be observed, which must be *real* numbers since eigenvalues of a hermitian operator are always real.

To *measure* a state $|\phi\rangle$ means to apply an observable to the state and take as a result one of the corresponding eigenvalues. Let λ be the eigenvalue derived from a measurement, the state that was measured will be modified into an *eigenvector* corresponding to λ.

By putting everything together we can see how a quantum computation will work: we start with a superposition, we apply a series of unitary operators to do the computation, and we measure at the end to have a result. It is important to notice that measuring in quantum mechanics plays an important role. After the measurement, we can no longer exploit the advantages given by superposition since the measurement itself collapses the system causing it to be in a single, deterministic, state.

Until now we talked about quantum systems with just one particle. Of course, it is possible to have quantum systems with more than one particle. The operation used to assemble quantum systems is the *tensor product*. Let us compose a simple system of two particles x and y. The x will have the $\mathbb{V}$ vector space associated with it, with basic states:

$$\begin{aligned}
&|x_0\rangle \longmapsto [1, 0, \ldots, 0]^T \\
&|x_1\rangle \longmapsto [0, 1, \ldots, 0]^T \\
&\vdots \\
&|x_{n-1}\rangle \longmapsto [0, 0, \ldots, 1]^T
\end{aligned} \tag{1.5}$$

While for y we have the $\mathbb{V}'$ vector space and the basic states:

$$\begin{aligned}
&|y_0\rangle \longmapsto [1, 0, \ldots, 0]^T \\
&|y_1\rangle \longmapsto [0, 1, \ldots, 0]^T \\
&\vdots \\
&|y_{m-1}\rangle \longmapsto [0, 0, \ldots, 1]^T
\end{aligned} \tag{1.6}$$

The assembled system will exist in the vector space obtained by tensoring the two vector spaces: $\mathbb{V} \otimes \mathbb{V}'$. The basic states for this vector space will be:

$$\begin{aligned}
&|x_0\rangle \otimes |y_0\rangle \longmapsto [11, 00, \ldots, 00]^T \\
&|x_0\rangle \otimes |y_1\rangle \longmapsto [10, 01, \ldots, 00]^T \\
&\vdots \\
&|x_0\rangle \otimes |y_{m-1}\rangle \longmapsto [10, 00, \ldots, 01]^T \\
&|x_1\rangle \otimes |y_0\rangle \longmapsto [01, 10, \ldots, 00]^T \\
&\vdots \\
&|x_{n-1}\rangle \otimes |y_{m-1}\rangle \longmapsto [00, 00, \ldots, 11]^T
\end{aligned} \tag{1.7}$$

A general state in this system will be the linear combination:

$$|\phi\rangle = c_{0,0}\,|x_0\rangle \otimes |y_0\rangle + \ldots + c_{i,j}\,|x_i\rangle \otimes |y_j\rangle + \ldots + c_{n-1,m-1}\,|x_{n-1}\rangle \otimes |y_{m-1}\rangle \tag{1.8}$$

When we work with assembled systems, we can observe another important property of quantum mechanics: *entanglement*. Let us use a simple system with two particles, and let us consider the following state:

$$\begin{aligned}|\phi\rangle &= 1\,|x_0\rangle \otimes |y_0\rangle + 0\,|x_0\rangle \otimes |y_1\rangle + 0\,|x_1\rangle \otimes |y_0\rangle + 1\,|x_1\rangle \otimes |y_1\rangle \\ &= |x_0\rangle \otimes |y_0\rangle + |x_1\rangle \otimes |y_1\rangle \end{aligned} \tag{1.9}$$

The state is from an assembled system but it cannot be written as the tensor product of two states coming from the two subsystems. In fact, if we try, we will end up with:

$$\begin{aligned}(c_0\,|x_0\rangle + c_1\,|x_1\rangle) \otimes (d_0\,|y_0\rangle + d_1\,|y_1\rangle) = c_0 d_0\,|x_0\rangle \otimes |y_0\rangle + c_0 d_1\,|x_0\rangle \otimes |y_1\rangle \\ + c_1 d_0\,|x_1\rangle \otimes |y_0\rangle + c_1 d_1\,|x_1\rangle \otimes |y_1\rangle\end{aligned} \tag{1.10}$$

For (1.10) to be the same as (1.9), we should have that $c_0 d_0 = c_1 d_1 = 1$ and $c_0 d_1 = c_1 d_0 = 0$ but these two equations cannot be both true at the same time. From this we can conclude that $|\phi\rangle$ is impossible to write as a tensor product of its components.

States that behave like this are called *entangled states*. Let us now consider the following state:

$$|\phi\rangle = 1\,|x_0\rangle \otimes |y_0\rangle + 1\,|x_0\rangle \otimes |y_1\rangle + 1\,|x_1\rangle \otimes |y_0\rangle + 1\,|x_1\rangle \otimes |y_1\rangle$$

If we try to write this state as the tensor product of its components, we found ourselves in a situation similar to the equation (1.10), but this time we need to have $c_0 d_0 = c_1 d_1 = c_0 d_1 = c_1 d_0 = 1$, which is solvable. This is what we call a *separable state*.

We have now enough information to see an example of quantum computation. A situation very similar to a normal coin toss is perfect to see quantum computation in action and how it differs from a probabilistic computation. We start with two possible outcomes (head or tail) and represent them with vectors. Then, we use a time evolution that gives us a fifty-fifty chance to yield each result.

Given the two vectors:

$$|h\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |t\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1.11}$$

And the transformation given by the unitary operator:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{1.12}$$

After one unit of time or (to go on with the analogy) a toss, the result is what you would expect: a 50% probability that either heads or tails is the outcome.

$$H|h\rangle \longmapsto \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \longmapsto \frac{1}{\sqrt{2}}|h\rangle + \frac{1}{\sqrt{2}}|t\rangle \tag{1.13}$$

$$H|t\rangle \longmapsto \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \longmapsto \frac{1}{\sqrt{2}}|h\rangle - \frac{1}{\sqrt{2}}|t\rangle \tag{1.14}$$

But if we assume that in the first toss the result is heads, in the next toss we have a 100% chance that the outcome will be heads.

$$HH|h\rangle \longmapsto H\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \longmapsto |h\rangle \tag{1.15}$$

1.2 Implementing Quantum Computation

The real challenge of quantum computing lies in its implementation.

It is complex to build a machine with enough computing power to be useful, because the more complicated the system is, the more it is prone to errors. This happens because quantum objects are highly interactive, and could become entangled with the environment, collapsing the state of the machine. This is known as *decoherence*. It is necessary to find the right balance between isolating the system to reduce decoherence and let it communicate with the environment. A way to handle the problem is given by Shor in [10].

There still is skepticism on the possibility of building physical quantum computer. For example Kalai and Kindler [11] consider that the efforts needed for noise correction would balance out the advantages given by a quantum computer, making it no more efficient than a classical machine.

The possible speed up introduced by quantum computing brings people to think of quantum machines as the next step in Moore's law, but this is not the case. Quantum computing can be used to speed up the process only of specific problems with specific algorithms and would still rely on classical computation for certain aspects. It is more useful to see quantum computing as something parallel to classical computing.

It is possible to build physical quantum computers and a proof of that is given by IBM that successfully built a 50 qubits machine. However, we are still far from building a computer that could bring about *quantum supremacy* (i.e. the ability of solving problems that classical computer cannot practically solve using quantum computers).

Chapter 2

Quantum Languages

To be able to program a quantum computer, all the abstraction layers needed for computation must be rebuilt. We start from the logic gate level, where we need to introduce new types of bits and gates that works following the rules of quantum physics. We will then see how to interact with the architecture through assembly and high level languages.

2.1 Architecture

In a quantum computer, information is represented by *quantum bits* also called *qubits*. The main difference between a qubit and a classical bit is that one bit can be used to represent either 1 or 0, while a qubit is capable of representing both 1 and 0 at the same time, each with an amplitude.

We can see the two deterministic states of a classical bit as the vectors:

$$|\mathbf{0}\rangle = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |\mathbf{1}\rangle = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.1}$$

A qubit can instead be in a *superposition*, therefore a general state of a single qubit can be seen as a vector in the vector space $\mathbb{C}^2$:

$$\begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \qquad \textit{where} \quad (c_0 + c_1)^2 = 1, \quad c_0, c_1 \in \mathbb{C} \tag{2.2}$$

As a bit is used to indicate the passage of electrical current (0 there is no current, 1 there is current), a qubit represents the properties of a quantum object (such as the spin of a subatomic particle), and to do so, a qubit needs to represent a quantum state (in particular a two dimensional quantum state).

We can see better the difference of the two systems when we start looking at the difference between a byte and a *qubyte*. A byte is a group of 8 independent bits. Similarly a qubyte is a quantum system composed by 8 qubits. To assemble a quantum system we have to tensor its components. The vector that represents a qubyte would be an element of the vector space

$$\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2 = \mathbb{C}^{256} \tag{2.3}$$

Every state of a qubyte can be written as:

$$\begin{array}{c} \mathbf{00000000} \\ \mathbf{00000001} \\ \vdots \\ \mathbf{01100010} \\ \mathbf{01100011} \\ \mathbf{01100100} \\ \vdots \\ \mathbf{11111111} \end{array} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{98} \\ c_{99} \\ c_{100} \\ \vdots \\ c_{255} \end{bmatrix}$$

Which means that if one were to simulate a single qubyte in a classic machine, they would need to store 256 complex numbers.

We talked about how to represent data, now we have to implement the operations needed to process it.

In classic systems, we have logic gates to perform operations between bits. We can represent a logic gate by a $2^m \times 2^n$ matrix, where m and n are respectively the number of states in input multiplied by their dimension and the number of states in output multiplied by their dimension. Some of the most common logic gates are:

$$\mathbf{NOT} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \mathbf{AND} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{OR} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

To apply a gate to an input means doing the operation:

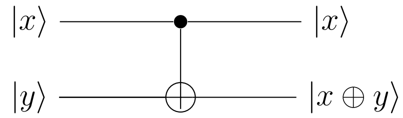
$$\mathbf{AND}\,|11\rangle = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

We need a quantum version of the logic gates to perform operations between qubits. As we have seen before, in quantum mechanics the transition from a state to the other is made possible through unitary matrices, this is why a quantum gate will be represented as a unitary matrix. The most important propriety that quantum gates must satisfy is *reversibility*. All the transitions in a quantum system can be reversed, therefore quantum gate too must be reversible. Given the gate and the output, one should be able to find the input.

Some classic (logic) gates are already reversible, an example is the NOT gate: if the output of a **NOT** gate is $|1\rangle$, the input was $|0\rangle$, if the output is $|0\rangle$ the input was $|1\rangle$. The **AND** gate isn't reversible: $|00\rangle$ can be the result of any of the inputs $|00\rangle$ $|01\rangle$ or $|10\rangle$.

Some important reversible, and therefore quantum, gates are:

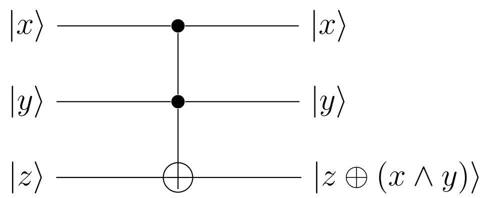
- **CNOT** or controlled not. It works with two qubits as input, one is the target qubit, the other is the control qubit. If the control bit is $|1\rangle$ then the gate flips the target qubit, otherwise nothing happens.



The unitary matrix representing this gate is:

$$\mathbf{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.4}$$

- **Toffoli gate**. It is similar to the CNOT gate but it acts on three qubits, where two bits are used as the control. The target qubit is flipped only if both the control qubits are in state $|1\rangle$.



The unitary matrix that represents the **Toffoli gate** is:

$$\textbf{Toffoli} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.5}$$

The **Toffoli gate** is particularly interesting because it is reversible, hence can be used as a quantum gate, and with various combination of Toffoli gates one can build circuits that act as any logic gate making the Toffoli gate an universal logic gate.

- **Phase shift gates**. If we think about the qubit in its representation as a vector on a **Bloch sphere**, the phase shift gates can be used to rotate the vector around one of the three axis of the sphere. If we want to do a 180° degree rotation around the x, y, or z axis, we use the **Pauli gates**, represented by the unitary matrices:

$$\mathbf{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \mathbf{Z} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.6}$$

While to do a rotation around the sphere of an arbitrary degree θ around the three axis, the matrices to use are the following:

$$\mathbf{R_x}(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\,\sin\frac{\theta}{2} \\ -i\,\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{2.7}$$

$$\mathbf{R_y}(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{2.8}$$

$$\mathbf{R_z}(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \tag{2.9}$$

Indeed $\mathbf{X} = \mathbf{R_x}(\pi)$, $\mathbf{Y} = \mathbf{R_y}(\pi)$ and $\mathbf{Z} = \mathbf{R_z}(\pi)$.

We can combine together different gates to form new unitary transformations. This can be achieved by either concatenating gates or tensoring gates. For example, we can concatenate the $\mathbf{X}$ and $\mathbf{Y}$ gates:

$$\mathbf{XY} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = \begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix}$$

which corresponds to apply the two gate one after the other. Or we can tensor the two gates:

$$\mathbf{X} \otimes \mathbf{Y} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{bmatrix}$$

which yields a new unitary operator with a different dimension. This corresponds to a parallel application of the two gates.

2.2 Assembly Language

One could imagine, in a classic machine, to write a whole program with just a combination of logic gates, namely by giving a circuit. It would be possible to do the same thing with quantum circuits, but no one would ever write a program that way: we need a language. At the very least, we need an *assembly language*.

Since any assembly language is deeply connected to the architecture it is written for, we'll just outline some characteristics that such a language for a quantum machine should have.

When thinking about a quantum machine, it usually seen as a classic machine that does all the work with a quantum unit that can be used when quantum computation is needed. Let us assume a few characteristics of the architecture we are working with. We assume to have: a **Random Access Quantum Memory** device, so that we are able to store data as quantum objects and address the data easily; a classic control flow.

The basic elements of a quantum assembly language are:

- quantum register
- quantum gates
- measurement

A *quantum register* is what we use to address the data: it refers to a specific group of qubits in memory, giving them a name. A register needs to be initialize before it can be used, so the language will need an operation that does just that. What the initialization should do, can vary: set all the qubits of the register at $|0\rangle$, set them all at $|1\rangle$, let the user initialize each qubit to either $|0\rangle$ or $|1\rangle$.

A set of *quantum gates* is needed to do the actual computation. The set can be made up of any unitary operator, as long as the set is universal. The operation needed to work with gates are:

- something to apply the gate to the register (or registers),

- something to build circuits: we need to be able to combine gates together either sequentially (by concatenation of gates) or in parallel (by tensoring gates).

Once the computation is done, we need something that *measures* its result. This is done by collapsing the register into a deterministic state, and then store the result in a classic register. The measure operation is the only one that can be irreversible

Since we assumed to be working with an hybrid machine (quantum data and classic control flow), we can hypothesize the existence of an `if` operator that can control the flow of the computation on the base of some classic register. The condition of the `if` cannot be done with quantum registers because to do a comparison there needs to be a measurement, which can't be done without collapsing the quantum state.

2.3 High Level Languages

It may seem odd that one would start working on high level languages when there is barely a machine to perform operations on. But unlike some technological advances such as transistors, with quantum computing the power does not lie in the hardware itself but in how it is used. That is why working on quantum algorithms is essential. Without a proper language, one should think of new algorithms in the terms of registers and gates, or in vectors and matrices, but with an high level language working on quantum algorithms becomes more accessible.

That is why, even though the quest to build a quantum machine is still in its early days, there's no shortage of attempts to define high level quantum languages in both the imperative and functional paradigms. The approach taken to define the languages is similar to the one taken to define the architecture of the machines: quantum data + classic control.

If we look at the imperative paradigm, we usually have languages that act very much like normal imperative languages, but with the addition of quantum data type commands to apply unitary operators to the quantum data. One of the first quantum language was QCL [15], a C like language which integrates the idea of quantum data types and operation with classic operations. One of the most important features of this language is the possibility to define new (quantum) operators, much like you would do with a function.

In the functional world, the line between data and control is blurry, but the basic idea is to have some sort of quantum data type and let the unitary operators be functions that work with those data types. One of such languages is QML [1], which also introduces a sort of quantum control. Another language is Quipper [5], which is thought as an extension for the Haskell language.

Chapter 3

QASM

QASM is a language built to serve as an assembly language for IBM's quantum computer. It can be used to interact with the *quantum experience* platform, accessible online[1].

From the platform one can interact with a real 5 qubits device or with its simulator. It is possible to interface with the device both through a QASM program and a graphic interface, the *Composer*[2]. IBM also has a 16 qubits device which cannot be accessed directly through the online platform. To interact with it, it is necessary to use QISKit [20], a software development kit that allows to connect remotely to the quantum device or the simulator true a Python program.

The full definition of QASM can be found at [3]. We will give here a brief overview of the language.

3.1 Language

The primary function of QASM is the definition of quantum circuits. The language uses a set of base gates from which one can define new gates with a mechanism akin to subroutines. The language disregards white spaces and is

[1]https://quantumexperience.ng.bluemix.net/qx/experience
[2]https://quantumexperience.ng.bluemix.net/qx/editor

case sensitive. Every statement must finish with a semicolon. The identifiers used to name registers and gates can use alphanumeric characters and must begin with a lowercase letter.

3.1.1 Preamble

The first line in a QASM document should be `OPENQASM m.n`, where `m` and `n` are the version numbers (e.g. `2.0`). It is possible to include libraries, using the command `include filename`, where `filename` is the path to the library file. This command should come right after the `OPENQUASM m.n` line.

```
OPENQASM m.n;
include "filename";
...
```

3.1.2 Registers

A quantum register is defined by `qreg name[size]`, where `size` indicates how many qubits the register is going to be formed of, while `name` is the name that one should use to address the register. One could access the whole register at once by using `name`, or access just one qubit of the register with `name[i]`, where `i` is the index of the qubit they wishes to access (very much like one would do with an array in C). When a `qreg` is created it is initialized to be in state $|0\rangle$.

A classic register is created in a similar way to a quantum register: `creg name[size]` declares an array of bits (or a register) with the given name and size.

```
...
qreg a[2];
qreg b[2];
creg c[3];
...
```

3.1.3 Gates

The built-in universal gate basis is formed by **CNOT** and a general 2 by 2 unitary operator U.

The **CNOT** gate is implemented by the command `CX a, b` that applies a CNOT gate with `a` as the control qubit and `b` as the target qubit.

If `a` and `b` are the identifiers of a register, and not qubits, then they must be of the same size s. To do `CX a, b` would be like doing `CX a[i], b[i]` for all $i = 0 \dots s-1$. If `a` is a single qubit and `b` a register, then the command stands for `CX a, b[i]` for all $i = 0 \dots d-1$ (where d is the dimension of `b`), and vice-versa. The base's unitary operator U can be represented by the matrix :

$$U(\theta,\phi,\lambda) = R_z(\phi)R_y(\theta)R_z(\lambda) = \begin{bmatrix} e^{-i(\phi+\lambda)/2}\cos(\theta/2) & -e^{-i(\phi-\lambda)/2}\sin(\theta/2) \\ e^{i(\phi-\lambda)/2}\sin(\theta/2) & e^{i(\phi+\lambda)/2}\cos(\theta/2) \end{bmatrix} \tag{3.1}$$

which can be used to define any 2 by 2 unitary matrix. The R_y and R_z in (3.1) are the phase shift gates seen in (2.8) and (2.9).

For example the Pauli gate **X** can be implement in QASM as `U(pi,0,pi)`.

```
...
CX a[1], b[2];
CX a[0], b;
CX a, b;
U(pi, 0, pi) a;
...
```

3.1.4 User Defined Gates

The command to define new gates is:

```
gate name(params) qargs{
    body
}
```

Where `params` is a comma-separated list of parameter names and the argument list `qargs` is a comma-separated list of qubits. There has to be at least one qubit as argument. It is allowed to have no parameters and when that is the case, the parenthesis can be omitted. Recursive gates are not allowed. In the body one can only use built-in gate statements and calls to previously defined gates. When using the quantum arguments in the body, it is not allowed to index them (e.g. if `a` is the argument, one cannot use `a[i]` in the body). The way to use a custom gate is similar to the one used for built-in gates.

```
gate g qb0,qb1,qb2,qb3{
   // body
}
qreg qr0[1];
qreg qr1[4];
qreg qr2[3];
qreg qr3[4];
g qr0[0],qr1,qr2[0],qr3;
```

Notice that the gate can be applied to both single qubits and registers. In case the arguments are registers they must be of the same size, because the application of a gate to one or more registers is seen as: `g qr0[0], qr1[j], qr2[0], qr3[j];` for j that goes from 0 to 3.

3.1.5 Measurements

The command used for measurements is `measure qubit|qreg -> bit|creg`. This operation measures the qubit in the Z axis and stores the result in a bit, leaving the qubit to be immediately reused. The arguments of `measure` can be either single qubits and bits or `qreg` and `creg`. In the second case the `qreg` and the `creg` must be of the same size.

The `reset qreg|qubit` statement is used to set a qubit or a `qreg` back to state $|0\rangle$.

```
...
measure a[1] -> c[0];
measure a -> c;

reset b;
...
```

3.1.6 Control Flow

The *if* statement permits to execute a quantum operation if the condition is met. The command is `if (c==n) operation`, where `c` is a `creg` and `n` is an integer number. The register is interpreted as a binary integer where the bit at index zero is seen as the low order bit.

The condition of the *if* is met only when the integer represented by the `creg c` is equal to the number `n`.

```
...
if (c == 9) <body>;
...
```

Chapter 4

QScript

QScript is a language designed to work with Google's platform *quantum playground* [18].

The platform is a browser-based application that offers an environment to implement, compile and run quantum programs for a 22 qubits simulator. It offers debugging features and a graphical environment for 3D visualization of quantum states. On the platform one can find, already implemented, both Shore's algorithm[9][1] and Grover's algorithm[6][2].

QScript is implemented using JavaScript as its base, it is therefore possible to take advantage of some constructs from JavaScript in QScript.

4.1 Language

QScript is a high level language composed of both classic and quantum aspects. We will start describing the classic aspects and later focus on the quantum aspects.

[1] http://www.quantumplayground.net/#/playground/5191954035900416
[2] http://www.quantumplayground.net/#/playground/5185026253651968

4.1.1 Variables and Typing

In QScript, contrary to JavaScript, the commands are terminated by a new line and not by a semicolon. Variables are declared automatically at first assignment and their type is inferred like in JavaScript. If a variable identifier starts with an underscore it has global scope.

The simulator works with a quantum register of a variable number of qubits that goes from 6 to 22. One must declare how big the register should be with the command `VectorSize n`, where $6 \leq n \leq 22$.

To access a qubit of the register, one should use the index of the qubit. For example, to access the fourth qubit, one should use `3` (the indexes of the vector start from 0).

```
VectorSize n
_global_variable = 3
variable = "a string"
```

4.1.2 Control Flow

The language has two control-flow constructs, `if...else...endif` and `for...endfor`, that work as normal *if* and *for* of a C-like language.

```
for initialization; termination; increment
    <body>
endfor
```

```
if condition
  <if_body>
else
  <else_body>
endif
```

There are no brackets to define where a block starts or ends, so it is done through the use of keywords.

4.1.3 Procedures

It is possible to define *procedures*. The command used is:

```
proc Name <param>
   <body>
endproc
```

where `Name` is the name of he procedure and `param` a list of identifiers. There is no recursion in QScript, but procedures can be nested meaning that one can define a new procedure in the body of a procedure:

```
proc Proc1 <param1>
   ...
   proc Proc2 <param2>
      <body>
   endproc
   ...
endproc
```

It is possible to use JavaScript's library *Math* functions.

4.1.4 Gates

The language has many built-in quantum gate:

- `Hadamard n`
  It applies the **Hadamard** gate to the qubit `n`. The unitary matrix **H** that represents the gate is:

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

- `SigmaX n`
  `SigmaY m`
  `SigmaZ p`
  The three commands apply the three **Pauli** gates respectively to the qubits `n`, `m` and `p`. The matrix representation of the gate can be found in section 2.6

- `Rx n,` α
  `Ry m,` θ
  `Rz p,` ϕ
  The three commands apply the three general phase shift gates respectively to the qubits `n`, `m` and `p` with degrees α, θ and ϕ. The matrix representation of these gates can be found in (2.7), (2.8) and (2.9)

- `CNot n, m`
  It applies a **CNOT** gate with `n` as the control qubit and `m` as the target. The definition of the gate can be found in (2.4).

- `Toffoli n, m, p`
  It applies the **Toffoli** gate where `n` and `m` are control qubits and `p` is the target. The definition of the gate can be found in (2.5).

- `Phase n,` θ
  The gate applied by this command maps the state $|1\rangle$ to $e^{i\theta}|1\rangle$ changing the phase of the qubit `n`. If the state is $|0\rangle$ nothing happens. The matrix $\mathbf{R}_\theta$ representing this gate is:

  $$\mathbf{R}_\theta = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$$

- `CPhase n, m,` θ
  It is a controlled `Phase` gate, where the control qubit is `n`.

- `Swap n, m`

It swaps the amplitude of the two qubits. It matrix representation is:

$$\textbf{Swap} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- `Unitary n, r00, im00, r01, im01, r10, im10, r11, im11`
  The command is used to define a 2 by 2 unitary matrix to apply to the qubit `n`. The other arguments are the matrix's entries where `r00` represent the real part in position 00, `im00` the imaginary part in position 00, etc.

- `QFT m, n`
  It applies a *Quantum Fourier Transform* where `m` is the starting qubit and `n` is the width of the `QFT` given in qubits. The corresponding unitary matrix is:

$$\textbf{QFT} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{N-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2(N-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \dots & \omega_n^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{(N-1)} & \omega_n^{2(N-1)} & \omega_n^{3(N-1)} & \dots & \omega_n^{(N-1)(N-1)} \end{bmatrix}$$

  where $N = 2^n$ and $\omega_n = e^{\frac{2\pi i}{2^n}}$

- `InvQFT m, n`
  It is similar to `QFT` but it performs an inverse transform.

- `ShiftLeft n`
  `ShiftRight n`
  The shifting gates move the entire quantum vector to the left or the right by `n` positions. The qubits destroyed in the shifting operation are considered as qubits that have not been used or have been just measured.

- `Decoherence n`
  It is used to apply a random phase shift to the qubits, with strength `n` (which is a number between `0.1` and `1`). To understand its functioning refer to [14]

4.1.5 Measurements

We have two command to execute a measurement:

- `Measure`: which measures the whole register.
- `MeasureBit i`: which measures only the qubit `i`; this operation could affect other qubits of the vector.

In both cases the measured value is stored in the variable `measured_value`.

4.1.6 Graphic Environment

The simulator offers a graphic environment used to visualize the computation of the program as well as a console and debugging tools. There are a set of commands used to interact with this side of the simulation:

- `Display arg` is used to show something in the graphic environment of the simulator. The argument (`arg`) can be a string with html markup.
- `Delay n` is used to delay the execution of the program for `n` seconds.
- `Print arg` is used to print in the console of the simulator, where `arg` can be a string or a variable.

Chapter 5

Compiler

In this chapter we will describe a compiler capable to translate QASM programs into QScript programs.

A compiler can be structured into different phases, as in Figure 5.1.

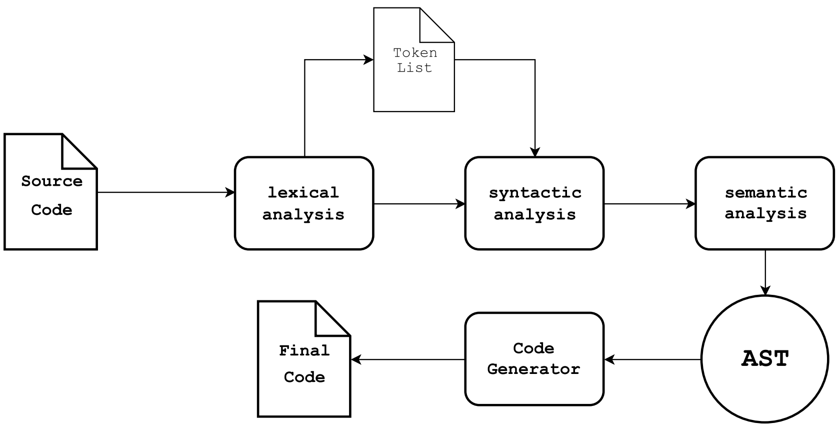


Figure 5.1: Compiler structure

The compiler takes as input a string representing the program in a source language and, through the generation of different internal representations, gives in output a string representing the program in the target language.

The **lexical analysis** has the role of recognizing the different *words* of a language, by separating the stream of character given in input into logical units called *tokens*. To define which words make up the language we use *regular expressions*. The **syntactic analysis** has the responsibility of identify *phrases* according to a given *grammar*. In this phase we build a logical representation of the program. The **semantic analysis** is used to give *meaning* to the different *phrases*.

At the end of this first three phases we have an intermediate code representation of the program, usually an abstract syntax tree (or AST)[1].

We can then find an additional phase, the **optimization**, with the task to produce an optimized version of the intermediate representation, so that the generation of the final code can be done more efficiently.

The last phase is the **code generation** that, by making use of the intermediate representation, generates the final code for the program in the target language.

There are numerous tools used to implement each phase of the compiler. The most common tool for a lexical analyzer generator is Lex [12], a software used by Unix systems for lexical analysis. Its core function is the construction of an automata capable to recognize patterns in an input stream through the use of regular expressions. The output of Lex is a C program that, given a stream of character in input, can recognize the patterns that compose it.

Lex is usually used together with Yacc [7], a software that generates a syntactic analyzer, or parser. It takes in input a grammar and produces a C program that is a parser for that grammar. It often happens that semantic analysis is done simultaneously with the syntactic analysis, by defining the action to take for each grammar rule.

In the implementation of our compiler we used PLY [2], a package that offers a Python implementation for both Lex and Yacc. It consists of two separate modules `lex.py`, that implements Lex, and `yacc.py`, that imple-

[1]An AST is a representation in the form of a tree of the structure of a source code. Each node of the tree stand for a construct of the source code.

ments Yacc. The two modules work together: `lex.py` provides the list of token to `yacc.py` that invokes grammar rules. The output of `yacc.py` is usually an AST.

The main difference between the modules offered by PLY and the standard Lex and Yacc is that with PLY there is no need to generate separate code. Moreover, the modules are valid Python programs, so there is no need of extra steps to generate the compiler. The generation of the parser is quite expensive, thus a parsing table is saved and used as cache unless there are major changes to the grammar.

5.1 Implementation

The compiler is implemented in Python and a package, `qs_compiler`, was created to provide a command line tool to use it.

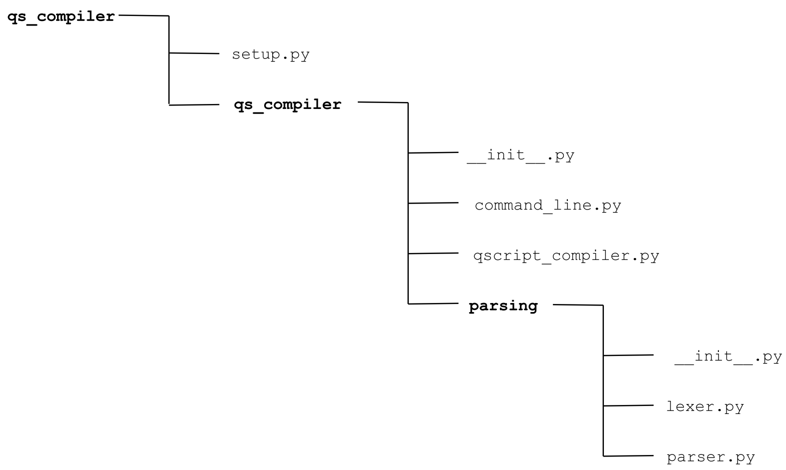


Figure 5.2: `qs_compiler` structure

As shown in Figure 5.2, the `qs_compiler` package contains a sub-directory, with the same name as the package, `qs_compiler` and a file `setup.py`. The file is needed so that it is possible to install the package through `pip install`.

In the `qs_compiler` subdirectory we can find the sub directory, `parsing`, and three files:

- `__init__.py`: needed for packaging purposes;
- `command_line.py`: needed to start the compiler from the command line;
- `qscript_compiler.py`: code to generate the final QScript file;

In the `parsing` directory we can find the files:

- `__init__.py`: needed for packaging purposes;
- `lexer.py`: code to implement the lexical analyzer by using *lex*, a tool to build a lexical analyse;
- `parser.py`: code to implement the syntax analyzer by using *yacc*, a tool to build a syntax analyzer, and to create an intermediate representation of the program;

The implementation of the lexer and the parser was fairly straightforward, since the documentation for QASM given in [3] is very detailed. The biggest challenge, in this first phase of the implementation, was to find the best way to implement the structure for the AST.

The implementation of the code generator, on the other end, was not as clear. QASM and QScript are fundamentally different. The former is more focused around registers and gates. The latter focuses more on the high level aspects of the program. This is reflected by the definitions of the two languages. The differences can be seen starting from the definition of qubits. While in QASM they are grouped in different registers defined by the user, in QScript all the qubits are part of the same register. Moreover, in QASM the way to identify a specific qubit is easy and readable, while in QScript it is just a number. Therefore it was necessary to create functions and data structures that would help with keeping the correspondence between qubit in QASM and qubit in QScript.

The different approach to the qubits becomes a problem particularly in the application of quantum gates. In QASM it is possible to apply a gate to a whole register, while in QScript it can be done only with a single qubit. To handle the situation it was necessary to find a way to distinguish the two cases in QASM (the grammar treats references to both quantum registers and qubits as IDs), in order to treat them accordingly. The application to custom gates was especially challenging due to the variable number of inputs.

The peculiar function of the `if` in QASM does not have a direct translation in QScript. It was necessary to convert the integer in the condition in its binary form and save the result in an array, so that the bit-by-bit comparison could be carried out in QScript.

QScript does not provide a way to define custom gates. After some experiments, it has been shown that the `proc` construct could be used for this purpose.

Another construct that is not present in QScript is QASM' `creg`. The best way to translate the classic register would have been through an array, but it was not possible to do so since arrays in QScript are constants. Variables were used instead.

QScript offers two different constructs for measuring qubits but neither could directly translate QASM's `measure`. The problem was mainly with the measure of a whole register, where there are similar problems to those found with the gate application.

In QASM, once measured, a quantum register needs to be saved in a classic register. Since classic registers are saved as variables in QScript, it was necessary to repeat the code as if the measurement was done one qubit at a time.

QASM's `reset` command has no equivalent in QScript. Since the purpose of the `reset` is to bring a qubit from a superposition back to a deterministic state, the `reset` was achieved by measuring the qubit.

5.1.1 `lexer.py`

The QASM program in input is analyzed to recognize the different tokens (a name-value pair) that make up the program. For example, the line:

```
qreg q[3];
```

is made up of six tokens :

- `QREG : qreg`
- `ID : q`
- `LSPAREN : [`
- `NNINTEGER : 3`
- `RSPAREN: ]`
- `SEMICOLON: ;`

To do so we use the lexical analyzer (or lexer) generator *Lex*, specifically we use the Python implementation of *Lex* provided by PLY [2]. To build the lexer, we need to provide a list of token names and the regular expressions to specify them. The list of token is given by:

```
tokens = [
   'ID',
   'OPENQASM',
   'REAL',
   'NNINTEGER',
   'SEMICOLON',
   'COMMA',
   'LPAREN',
   'LSPAREN',
   'LCPAREN',
   'RPAREN',
```

```
12     'RSPAREN',
13     'RCPAREN',
14     'ARROW',
15     'EQUAL',
16     'PLUS',
17     'MINUS',
18     'TIMES',
19     'DIVIDE',
20     'POWER',
21     'U',
22     'CX'
23 ] + list(reserved.values())
```

In line 23, `list(reserved.values())` is used to add to the token list the tokens for reserved words. Reserved words are stored in a dictionary, where the key is the *regular expression* corresponding to the reserved word.

```
reserved = {
'qreg' : 'QREG',
'creg' : 'CREG',
'barrier' : 'BARRIER',
'gate' : 'GATE',
'measure' : 'MEASURE',
'reset' : 'RESET',
'include' : 'INCLUDE',
'opaque' : 'OPAQUE',
'if' : 'IF',
'sin' : 'SIN',
'cos' : 'COS',
'tan' : 'TAN',
'exp' : 'EXP',
'ln' : 'LN',
'sqrt' : 'SQRT',
'pi' : 'PI'
```

```
}
```

The regular expression rules for the remaining tokens are defined by declaring variables in the format `t_`⟨*name*⟩, where ⟨*name*⟩ is the name of the token associated with the regular expression.

Simple rules are:

```

t_SEMICOLON = r';'
t_COMMA = r','
t_LSPAREN = r'\['
t_RSPAREN = r'\]'
t_LCPAREN = r'\{'
t_RCPAREN = r'\}'
t_EQUAL = r'=='
t_ARROW = "->"
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE = r'/'
t_POWER = r'\^'
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

More complex rules are defined through functions :

```
def t_REAL(t):
  r'([0-9]+\.[0-9]*|[0-9]\.[0-9]+)([eE][-+]?[0-9]+)?'
  t.value = float(t.value)
  return t

def t_NNINTEGER(t):
  r'[1-9]+[0-9]*|0'
  t.value = int(t.value)
  return t
```

```python

def t_ID(t):
  r'[a-z][A-Za-z0-9_]*'
  t.type = reserved.get(t.value,'ID') #this cheks if there are
      reserved character in the string.
  return t
```

There always is a single argument `t` which is a LexToken[2] object.

The tokens to be ignored (such as white spaces) are specified using the prefix `t_ignore`:

```python
t_ignore = ' \t'
t_ignore_COMMENT = r'//.*'
```

To handle errors we use the function:

```python
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

The lexer is built with the command:

```python
  lexer = lex.lex()
```

This function uses Python reflection to read the regular expression rules out of the calling context and build the lexer.

[2]LexToken's attributes are:

- t.type which is the token type (set to the name following the `t_` prefix),
- t.value which is the text matched
- t.lineno which is the current line number
- t.lexpos which is the position of the token relative to the beginning of the input text

5.1.2 `parser.py`

The next steps in the compiling process are the *syntax* and *semantic* analyses.

The *syntax* of a language is expressed in the form of a **BFN** grammar. A **BNF** grammar is a collection of rules in the form:

```
<symbol> ::= expression1 | expression2 | ...
```

Where the '::=' means that the symbol on the left is to be replaced with one of the expressions on the right, separated by a '|'. All the symbols that appear on the left side are *non-terminals*, while all symbols that appear on the right side but never on the left side are *terminals*.

The grammar for QASM can be found in Appendix A of [3]. In the grammar definition all that is not surrounded by ⟨⟩ is a *terminal* and corresponds directly to a token. The statements surrounded by ⟨⟩ are *non-terminal*.

The first rule in the grammar is ⟨`mainprogram`⟩, that will recognize the header of the program and is the entry point to the rest of the program. We can group the other rules in four sets:

- **Statements and declarations**: the rules that recognize the declarations of `qregs` and `creg`, the declaration of custom gates, and the `if` statement. The left-hand-side *non-terminal* for these rules are: ⟨`statement`⟩, ⟨`decl`⟩, ⟨`gatedecl`⟩.
- **Operations** : the rules that recognize the application of `CX`, `U`, custom gates, `measure` and `reset`. The left-hand-side *non-terminal* for these rules are: ⟨`goplist`⟩, ⟨`qop`⟩, ⟨`uop`⟩.
- **Identifiers** : the rules used to recognize the various identifiers. The left-hand-side *non-terminal* for these rules are: ⟨`anylist`⟩, ⟨`idlist`⟩, ⟨`mixedlist`⟩, ⟨`argument`⟩.

- **Expressions** : the rules to identify mathematical expressions. The left-hand-side *non-terminal* for these rules are: ⟨`explist`⟩, ⟨`exp`⟩, ⟨`unaryop`⟩.

A syntax analyzer should say which grammar rules were applied to yield a specific expression.

There are different ways to achieve this goal. Yacc uses **LR-parsing** (or *shift-reduce* parsing).

It is a bottom up technique, meaning that it tries to recognize in the grammar rules a right-hand-side expression that corresponds to the given input and immediately replaces it with the corresponding left-hand-side symbol.

In our compiler the yacc is implemented using the tools offered by PLY. First we need to import the token list built from the lexer:

```
from .lex import tokens
```

Then we define a structure `Node` that will be used to build the AST given in output by the yacc:

```

class Node:
 def __init__(self,type,children=None,node=None, param=None):
    self.type = type
    if children:
       self.children = children
    else:self.children = [ ]
    if param:
       self.param = param
    else:self.param = [ ]
    self.node = node

 def __str__(self, level=0):
    if self.node:
       ret="\n|" + "  |"*level +"\n|" + " |"*level + "---"
          +str(self.node)
```

```
    else:
        ret=""

    for child in self.children:
        if self.node:
            ret +=child.__str__(level+1)
        else:
            ret+=child.__str__(level)

    return ret
```

The class has three fields:

- `node` : is a construct of the language;
- `type` : is the type of the node;
- `children` : is a list of children of the node;
- `param` : is a list of objects that will be needed in the translation process;

The class has a method (`__str__`) used to print the AST in a nice format.

Each grammar rule is defined by a Python function. The appropriate context-free grammar is specified in the documentation strings of the function (by including a string constant as the first statement in the function definition). The statements in the body of the function describe the semantic of the grammar rule.

Each function has one argument `p` which is a sequence containing the value of the symbols of the grammar rule, for example:

```
def p_exp_binary(p):
  '''exp : exp PLUS exp
         | exp MINUS exp
         | exp TIMES exp
         | exp DIVIDE exp
         | exp POWER exp'''
```

```
    if (p[2] == '+'):
        p[0] = p[1] + p[3]
    elif (p[2] == '-'):
        p[0] = p[1] - p[3]
    elif (p[2] == '*'):
        p[0] = p[1] * p[3]
    elif (p[2] == '/'):
        p[0] = p[1] / p[3]
    elif (p[2] == "^"):
        p[0] = p[1] ** p[2]
```

The full implementation of the parser for QASM can be found in Appendix A.

5.1.3 qscript_compiler

The translator takes in input the tree built by parsing and than translates node by node the various constructs into QScript, our target language.

It is implemented in the file `qscript_compiler`. The program flow starts from the `main` function:

```
def main(qscript, file_in, table):

    with open(file_in, "r") as f:
        data=f.read()

    result=pars(data, table)

    compileTree(result)

    if (vector_size < 6 or vector_size > 22 or vector_size % 2 !=
        0):
        sys.exit("VectorSize should be even, more than 6 and less
            than 22 " + str(vector_size))
```

```
12
13     final = "VectorSize " + str(vector_size) +"\n" + output
14
15
16     if (qscript == 'outfile'):
17         if file_in.endswith('.qasm'):
18             qscript = file_in[:-5]
19         qscript += ".qscript"
20
21     with open(qscript,"w") as f:
22         f.write(final)
```

The function has three arguments `qscript, file_in,` taken from the command line (more on that in section [ref]), where `qscript` is the name of the output file where to save the translated QScript program, `file_in` is the input QAMS program, and `table` is a flag used to not save the parse table.

Line 10-13 are needed because QScripts limits the size of the vector. QASM permits to declare quantum register separately and anywhere in the code, that is why the `VectorSize` is decided after the whole input has been processed by `compileTree(tree)`

The function `compileTree(tree)`, implements the tree visit. The argument `tree` is an `Node` object.

The full definition of the function can be found in Appendix B, lines 44-321. First we examine the node and than move on to examine the children of the node.

To discriminate which type of node we are analyzing so to trigger the right translation a check is done on `tree.type`. We have an *if* branch for each construct of the language. In each brunch the result of the translation is stored in a global string `output`.

Next, we will see how each construct is translated.

5.1.4 Registers

In QScript the qubits are all declared as part of the same quantum register. In the translator, a global dictionary is used to store the correspondence between the name of the registers in QASM and the index of the qubit vector in QScript. The entry of the dictionary are in the form

```
{ qreg_name : [qreg_size , qcount] }
```

The global variable `qcount` is used to keep count of the number of qubits that have been declared. In the dictionary it is used to store the VectorSize index corresponding to the first qubit in the qreg. In the dictionary it is used to store the VectorSize index corresponding to the first qubit in the qreg.

If in QASM we have the following gate declaration:

```
qreg a[5];
qreg b[3];
```

In QScript this will be translated into

```
VectorSize 8
```

And the global variable of qs_compiler will have the values:

```
qregs  = {
  a : [5, 0]
  b : [3, 5]
}
qcount = 8
```

The function `get_greg(id, index)` is used to find the index of the QScript vector that corresponds to the qubit at index `index` of the qreg `id`

```
def get_qreg(id, index):
  qreg = qregs[id]
  return index+qreg[1]
```

The full definition of the code used to handle quantum registers can be found in Appendix B, lines 101-104.

QScript does not provide classic registers, so they are translated to simple variables.

If in QASM the classic register are:

```
creg c[2];
creg d[3];
```

In QScript it will be translated in :

```
c_0 = 0
c_1 = 0
d_0 = 0
d_1 = 0
d_2 = 0
```

We use variables and not arrays because, even though it is possible to use arrays in QScript, once declared the arrays cannot be modified.

We use the global dictionary `cregs`, with entries in the format `id : size` , for error checking purposes.

To get the QScript variable from the QASM register, we use the function

```
def get_creg(id, index):
   creg = cregs[id]
   if (index < creg):
       return id + "_" + str(index) + " "
   else:
       sys.exit("out of range")
```

The full definition of the code used to handle classic register can be found in Appendix B, lines 107-110.

From now on we assume that we are working in an environment where we have defined the registers as follows:

```
qregs  = {
  a : [4, 0]
  b : [4, 4]
}
creg c[4]
```

5.1.5 Gates

The basis gate `U`(ϕ, λ, θ) is translated by using the `Unitary` operator of QScript. To calculate the `r00...im11` arguments of `Unitary` we use:

```
     r_oo = (math.cos(-(phi + llambda)/2))*math.cos(theta/2)
     r_oi = (math.cos(-(phi - llambda)/2))*math.sin(theta/2)
     r_io = (math.cos((phi - llambda)/2))*math.sin(theta/2)
     r_ii = (math.cos((phi + llambda)/2))*math.cos(theta/2)
     i_oo = (math.sin(-(phi + llambda)/2))*math.cos(theta/2)
     i_oi = (math.sin(-(phi - llambda)/2))*math.sin(theta/2)
     i_io = (math.sin((phi - llambda)/2))*math.sin(theta/2)
     i_ii = (math.sin((phi + llambda)/2))*math.cos(theta/2)
```

where `phi` $= \phi$, `llambda` $= \lambda$ and `theta` $= \theta$, in `r_oo..r_ii` we save the values corresponding to the real part of the entries of the unitary matrix, while in `i_oo..i_ii` we save the imaginary part. The expressions on the right hand side were obtained from the definition of `U`(ϕ, λ, θ) (3.1).

The `U` in QASM can be applied either to qubits or qregs. In the second case it is translated through the use of a `for` statement.

For example, the QASM command

```
U (0, 0, 0) a;
```

Would be translated to:

```
for i = 0; i < 4; i++
      Unitary i , 1.0, 0.0, 0.0, 1.0, -0.0, -0.0, 0.0, 0.0
endfor
```

In case U is applied to a single qubit, we have that:

```
  U (0, 0, 0) a[1];
```

translates to:

```
Unitary 1 , 1.0, 0.0, 0.0, 1.0, -0.0, -0.0, 0.0, 0.0
```

The full definition of the code used to handle U gate can be found in Appendix B, lines 169-192.

For the QASM CX gate the situation is similar to the U gate: we can use the QScript CNot gate. This time we have two input, therefore, in case CX is applied to two quantum registers, we first have to check if the two registers have the same dimension. For example, the command

```
CX a, b;
```

would be translated to

```
for i = 0; i < 4; i++
      CNot 0+i , 4+i
endfor
```

The command

```
CX a b[0];
```

Would be translated to

```
for i = 0; i < 4; i++
        CNot i , 4
endfor
```

And the command

```
  CX a[0] b[0];
```

Would be translated to

```
  CNot 0 , 4
```

The full definition of the code used to handle the `CX` gate can be found in Appendix B, lines 195-223.

The QASM custom gate are translated using the `proc` construct in QScript. For example :

```
gate g x,y,z {
  U(0,0,0) x;
  CX y,z;
}
```

Would be translated to:

```
proc g x, y, z

      Unitary x, 1.0, 0.0, 0.0, 1.0, -0.0, -0.0, 0.0, 0.0
      CNot y, z

endproc
```

The application of the gate is more complicated, because custom gates can be applied to both qubits and quantum registers.

We use the following global variables:

```
for_string=""
flag_anylist = False
gate_regs = {}
```

As it is the case for `CX` and `U`, when the gate is applied to quantum registers, it is translated with a *for*. The `for_string` variable is necessary because to parse a command like:

```
custom_gate (params) a, b;
```

we go through multiple grammar rules, so to format the final string consistently we use the `for_string` variable. The `flag_anylist` is used to keep track of the father of the node. As it can be seen in Appendix A at lines 200-214, a nodes of type `ID` and `ID_list` can be a children of an `anylist` node. When that is the case, it means that the ids are for qregs (not qubits) and so they need to be translated accordingly.

The `gate_regs` variable is used to store the name and size of the quantum registers the current custom gate is applied to. The function:

```
def equal_size(regs):
  size = 0
  for reg in regs:
      if (size == 0) :
          size = regs[reg]
      elif (regs[reg] != size):
          return False
  return (True, size)
```

is used to check if the the register in `gate_regs` are all of the same size. The above example would be translated in:

```
for i = 0; i < 4; i++
      custom_gate (params) i+0, i+4
```

```
endfor
```

The full definition of the code used to handle custom gates can be found in Appendix B, lines 116-123, 62-70, 228-235.

5.1.6 Control Flow

To translate an `if` statement, for example:

```
if ( c == 9) body;
```

We need a binary representation of the integers in the condition.

The binary representation is then stored in an array with the format *creg*_`if`, where at index 0 there is the low order bit.

The condition of the QScript *if* statement will check if the value in the `creg[i]` and *creg*_`if[i]` are the same.

The translation of the *if* statement above will be:

```
c_if = [1,0,0,1]
if c_0 == c_if[0] && c_1 == c_if[1] && c_2 == c_if[2]
   && c_3 == c_if[3]
        body
endif
```

The full definition of the code used to handle a *if* statement can be found in Appendix B, lines 73-95.

5.1.7 Measurements

The `measure` command is translated using the `MeasurBit` command.

```
measure a[0] -> c[0]
```

Will be translated to

```
MeasureBit 0
c_0= measured_value
```

The argument for measure can be either single qubit/bit or qreg/creg. In the second case, we will have the repetition of the above command for each qubit-bit pair:

```
  measure a -> c;
```

is translated to:

```
MeasureBit 0
c_0 = measured_value
MeasureBit 1
c_1 = measured_value
MeasureBit 2
c_2 = measured_value
MeasureBit 3
c_3 = measured_value
```

The full definition of the code used to handle a `measure` command can be found in Appendix B, lines 137-152.

The `reset` command is translated in QScript by measuring (trough MeasureBit) the qubit or qreg that needs to be resetted. For example:

```
//QASM
reset a[0];

//QScript
MeasureBit 0

//QASM
```

```
reset a;

//QScript
for i = 0; i < 4; i++
      MeasureBit i
endfor
```

The full definition of the code used to handle a `reset` command can be found in Appendix B, lines 155-163.

5.2 Command Line

The Python package *Click* [17] was used to implement the command line tool for the compiler, `qsc`. *Click* can be used to create commands, with relative arguments ed options, and automatically generate an help page. The command for the compiler is define in `setup.py` through the use of `setuptools`

```
setup(
  name='qs_compiler',
  version='0.1',
  description = 'Compiler from qasm to qscript',
  packages=find_packages(),
  py_modules=['qscript_compiler', 'yacc', 'lex'],
  install_requires=[
        'ply', 'Click',
    ],
  dependency_links=['https://github.com/dabeaz/ply'],
  entry_points={
      'console_scripts': ['qsc=qs_compiler.command_line:main'],
      }
)
```

The command and options are defined in `qs_compiler`

```
@click.command(context_settings={'help_option_names':['-h','--help']},
help='Compiler form a .qasm file in to a .qscript file, where FILE_IN is the path of the .qasm file to be translated')
@click.option('--qscript', '-q', default='outfile', help='Used to specify the name of the output file. If none is given, the name of the .qasm file will be used')
@click.option('--table', '-t', is_flag=True, help='With this flag, the compiler will NOT save the parse table, but create it every time')
@click.argument('file_in')
```

The help page generated from this code can be seen in figure 5.3.

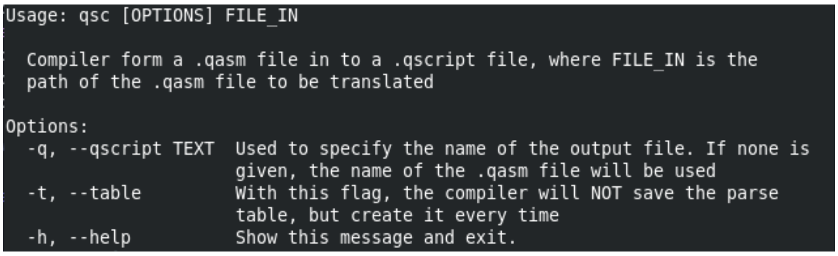


Figure 5.3: `qsc` help page

Conclusions and Future Developments

The quantum paradigm can bring about improvements in computer science, thanks to the quantum properties that make it possible to solve, in manageable time, problems that were considered until recently computationally infeasible.

At the core of these advantages are quantum algorithms. The technology alone does not provide any advantages, in contrast with what happened with, for example, the introduction of the transistor.

The algorithms are, though, deeply tied to their physical implementation, and that can be daunting from a computer scientist's point of view. To make the field more accessible it is necessary to have more tools that allow to experiment and get familiar with the new paradigm. The compiler implemented in this thesis is a step in that direction.

Future developments for this project could be:

- widen the pool of target languages of the compiler;
- implement a compiler from QScript to QASM;
- implement a compiler that has QASM as target language and as source languages a high level language, such as Quipper.

Appendix A

parser.py

```
import ply.yacc as yacc
import sys
from .lexer import tokens


# precedences to avoid shift/reduce conflicts

precedence = (
('left', 'PLUS', 'MINUS'),
('left', 'TIMES', 'DIVIDE'),
('right', 'POWER'),
('right', 'UMINUS'),
)

#Tree structure definition

class Node:
   def __init__(self,type,children=None,node=None, param=None):
      self.type = type
      if children:
         self.children = children
```

```
        else:self.children = [ ]
        if param:
            self.param = param
        else:self.param = [ ]
        self.node = node

    def __str__(self, level=0):
        if self.node:
            ret="\n|" + "  |"*level +"\n|" + " |"*level + "---" +str(self.node)
        else:
            ret=""

        for child in self.children:
            if self.node:
                ret +=child.__str__(level+1)
            else:
                ret+=child.__str__(level)

        return ret

#start

def p_start(p):
    'start : mainprogram'
    p[0] = p[1]

#The inclusion of files is not supported by this compiler, but the word 'include'
#is stiil reserved. This simple rule was made to avoid a warning of the type
# "token defined but not used"

```

```
def p_include(p):
   'start : INCLUDE mainprogram'
   p[0] = p[2]

   #mainprogram

def p_mainprogram(p):
    'mainprogram : OPENQASM REAL SEMICOLON program'
    p[0] = Node("openqasm",[p[4]],p[1]+" "+str(p[2])+p[3], [p[1]])


#program

def p_program_s(p):
    'program : statement'
    p[0] = Node("statement",[p[1]],None,None)

def p_program_ps(p):
    'program : program statement'
    p[0]=Node("program_statement",[p[1],p[2]],None,None)


#statement

def p_statement_decl(p):
    'statement : decl'
    p[0] = Node("decl",[p[1]],None,None)

def p_statement_gate_goplist(p):
    'statement : gatedecl goplist RCPAREN'
    p[0] = Node("statement_gate_goplist",[p[1],p[2]], p[1],None)

def p_statement_gate_decl(p):
```

```
    'statement : gatedecl RCPAREN'
    p[0] = Node("statement_gatedecl", [p[1]], None,None)

def p_statement_opaque(p):
    '''statement : OPAQUE ID idlist SEMICOLON
                 | OPAQUE ID LPAREN RPAREN idlist SEMICOLON
                 | OPAQUE ID LPAREN idlist RPAREN idlist SEMICOLON'''
    if(p[4]==";"):
        p[0] = Node("opaque", [p[3]], p[1] + " " + p[2], [p[3]])
    elif(p[6] == ";"):
        p[0] = Node("opaque", [p[5]], p[1] + " " + p[2] + " " + p[3]
            + p[4],[p[5]])
    elif(p[7] == ";"):
        temp = ""
        for child in p[4].children:
            temp += child.node + " "
        p[0] = Node("opaque", [p[6]], p[1] + " " + p[2] + " " + p[3]
            + temp + p[5],[p[4],p[6]])

def p_statement_qop(p):
    'statement : qop'
    p[0] = Node("qop", [p[1]], None, None)

def p_statement_if(p):
    'statement : IF LPAREN ID EQUAL NNINTEGER RPAREN qop '
    temp = p[2] + p[3] + p[4] + str(p[5]) + p[6]
    p[0] = Node("if",[p[7]], p[1] + " " + p[2] + p[3] + p[4] +
        str(p[5]) + p[6], [p[3],p[5]])

def p_statement_barrier(p):
    'statement : BARRIER anylist SEMICOLON'
    p[0] = Node("barrier", [p[2]], p[1], None)
```

```
#decl

def p_decl(p):
   '''decl : QREG ID LSPAREN NNINTEGER RSPAREN SEMICOLON
           | CREG ID LSPAREN NNINTEGER RSPAREN SEMICOLON'''
   p[0] = Node(p[1],None,p[1]+" "+p[2]+p[3]+str(p[4])+p[5]+p[6],
       [p[2],p[4]])


#gatedecl

def p_gatedecl(p):
    '''gatedecl : GATE ID idlist LCPAREN
                | GATE ID LPAREN RPAREN idlist LCPAREN
                | GATE ID LPAREN idlist RPAREN idlist LCPAREN'''
    if (p[4] == "{"):
        p[0] = Node("gatedecl", [p[3]], p[1] + " " + p[2],
            [p[2],p[3],None])
    elif(p[6] == "{"):
        p[0] = Node("gatedecl", [p[5]], p[1] + " " + p[2] + " " +
            p[3] + p[4], [p[2],p[5],None] )
    elif (p[7] == "{"):
      temp = ""
      for child in p[4].children:
         temp += child.node + " "
      p[0] = Node("gatedecl", [p[6]], p[1] + " " + p[2] + " " +
          p[3] + temp + p[5], [p[2],p[4],p[6]])


#goplist

def p_goplist_uop(p):
```

```
    'goplist : uop'
    p[0] = Node("goplist_uop", [p[1]], None, None)

def p_goplist_barrier(p):
    'goplist : BARRIER idlist SEMICOLON'
    p[0] = Node("goplist_barrier", [p[2]], p[1], [p[2]])

def p_goplist_gopuop(p):
    'goplist : goplist uop'
    p[0] = Node("goplist_gopuop", [p[1],p[2]], None, None)

def p_goplist_gopbarrier(p):
    'goplist : goplist BARRIER idlist SEMICOLON'
    p[0] = Node("goplist_gopbarrier", [p[1],Node("goplist_barrier",
        [p[3]], p[2])], None, [p[3]])


#qop

def p_qop(p):
    'qop : uop'
    p[0] = Node("qop_uop", [p[1]], None,None)

def p_qop_meas(p):
    'qop : MEASURE argument ARROW argument SEMICOLON'
    p[0] = Node("measure", [p[2],p[4]], p[1] + " " + p[3],
        [p[2],p[4],p.lineno(1)])

def p_qop_reset(p):
    'qop : RESET argument SEMICOLON'
    p[0] = Node("reset", [p[2]], p[1], [p[2]])
```

```
#uop

def p_uop_U(p):
    'uop : U LPAREN explist RPAREN argument SEMICOLON'
    p[0] = Node("uop_U", [p[5]], p[1] + "(" + str(p[3]) + ")",
        [p[3],p[5]])

def p_uop_CX(p):
    'uop : CX argument COMMA argument SEMICOLON'
    p[0] = Node("uop_CX", [p[2],p[4]], p[1],[p[2],p[4],p.lineno(1)])

def p_uop_id(p):
    'uop : ID anylist SEMICOLON'
    p[0] = Node ("uop_id", [p[2]], p[1], [p[1], p.lineno(1)])

def p_uop_idparen(p):
    'uop : ID LPAREN RPAREN anylist SEMICOLON'
    p[0] = Node("uop_id", [p[4]], p[1] + p[2] + p[3], [p[1],
        p.lineno(1)])

def p_uop_idexp(p):
    'uop : ID LPAREN explist RPAREN anylist SEMICOLON'
    p[0] = Node("uop_id", [p[5]], p[1] + "(" + str(p[3]) + ")",
        [p[1], p.lineno(1), p[3]])


#anylist

def p_anylist(p):
   '''anylist : idlist
              | mixedlist'''
   p[0] = Node("anylist", [p[1]], None, None )

```

```
#idlist

def p_idlist_id(p):
    'idlist : ID'
    p[0] = Node("ID", None, p[1], [p[1]])

def p_idlist(p):
    'idlist : idlist COMMA ID'
    p[0] = Node ("ID_list", [p[1], Node("ID", None, p[3])], None,
        [p[3]])


#mixedlist

def p_mixedlist_id(p):
    'mixedlist : ID LSPAREN NNINTEGER RSPAREN'
    p[0] = Node ("mixedlist1", None, p[1] + p[2] + str(p[3]) +
        p[4], [p[1],p[3]])

def p_mixedlist_mixid(p):
    'mixedlist : mixedlist COMMA ID'
    p[0] = Node ("mixedlist2", [p[1],p[3]], None, [p[3]])

def p_mixedlist_mixdidint(p):
   '''mixedlist : mixedlist COMMA ID LSPAREN NNINTEGER RSPAREN
                | idlist COMMA ID LSPAREN NNINTEGER RSPAREN'''
   p[0] = Node("mixedlist3", [p[1],Node("mixedlist",None, p[3] +
       p[4] + str(p[5]) + p[6])], None, [p[3],p[5]])


#argument

```

```
def p_argument_id(p):
    'argument : ID'
    p[0] = Node ("arg_id", None, p[1], None)

def p_argument_idint(p):
    'argument : ID LSPAREN NNINTEGER RSPAREN'
    p[0] = Node("arg_idint", None, p[1] + p[2] + str(p[3]) + p[4],
        [p[1],p[3]])


#explist

def p_explist_exp(p):
    'explist : exp'
    p[0] = p[1]

def p_explist(p):
    'explist : explist COMMA exp'
    p[0] = str(p[1]) + ", " + str(p[3])


#Da sistemare

#exp

def p_exp_term(p):
    '''exp : REAL
           | NNINTEGER
           | ID
           | PI'''
    p[0] = p[1]

def p_exp_binary(p):
```

```
    '''exp : exp PLUS exp
           | exp MINUS exp
           | exp TIMES exp
           | exp DIVIDE exp
           | exp POWER exp'''
    if (p[2] == '+'):
        p[0] = p[1] + p[3]
    elif (p[2] == '-'):
        p[0] = p[1] - p[3]
    elif (p[2] == '*'):
        p[0] = p[1] * p[3]
    elif (p[2] == '/'):
        p[0] = p[1] / p[3]
    elif (p[2] == "^"):
        p[0] = p[1] ** p[2]

def p_exp_negative(p):
    'exp : MINUS exp %prec UMINUS'
    p[0] = - p[2]

def p_exp_paren(p):
    'exp : LPAREN exp RPAREN'
    p[0] = p[2]

def p_exp_unary(p):
    'exp : unaryop LPAREN exp RPAREN'
    p[0] = Node("exp_unary", [p[3]], p[1], None)


#unaryop

def p_unaryop(p):
    '''unaryop : SIN
```

```
                | COS
                | TAN
                | EXP
                | LN
                | SQRT'''
    p[0] = p[1]

# Error rule for syntax errors
def p_error(p):
   sys.exit("Syntax error in input, at line " + str(p.lineno))


# Build the parser
def pars(data, w_t):
    parser = yacc.yacc(write_tables=w_t, debug=False)
    return parser.parse(data)
```

Appendix B

qscript_compiler.py

```python
from .parsing.parser import pars
import math
import sys
import click


qcount = 0
output = ""
qregs = {}
cregs = {}
def_gates = []
vector_size = 0
for_string=""
flag_anylist = False
gate_regs = {}


def get_qreg(id, index):
    qreg = qregs[id]
    return index+qreg[1]

```

```
def get_creg(id, index):
     creg = cregs[id]
     if (index < creg):
         return id + "_" + str(index) + " "
     else:
         sys.exit("out of range")


def equal_size(regs):
    size = 0
    for reg in regs:
        if (size == 0) :
            size = regs[reg]
        elif (regs[reg] != size):
            return False
    return (True, size)

def declared_gate(gate):
    return gate in def_gates



def compileTree(tree):
    global qcount
    global vector_size
    global output
    global flag_anylist
    global gate_regs
    global for_string

    #mainprogram


```

```
    #program
    if (tree == None):
        sys.exit("ERROR: the input file is empty")
        return

    #statement

    if (tree.type == "statement_gate_goplist"):
        compileTree(tree.children[0])
        compileTree(tree.children[1])
        output += "\nendproc\n\n"


    elif (tree.type == "statement_gatedecl"):
        compileTree(tree.children[0])
        output += "\nendproc\n\n"


    elif (tree.type == "if"):
        id_reg = tree.param[0]
        binary = bin(tree.param[1])[2:]
        if (len(binary) > cregs[id_reg] ):
            output += "//if statement always false\n"
        else:
            output += id_reg +"_if = ["
            i=len(binary) - 1
            while (i >= 0):
                output += binary[i] + ","
                i-=1
            if (len(binary) < cregs[id_reg]):
                for j in range(cregs[id_reg]-len(binary)):
                    output += '0,'
            output = output[:-1] + "]\n\n"
```

```
            output += "if "
            for k in range(cregs[id_reg]):
                output += id_reg+"_"+ str(k) + " == " + id_reg+"_if[" + str(k) + "]"
                if (k < cregs[id_reg] - 1):
                    output += " && "
            output+= "\n\t"
            compileTree(tree.children[0])
            output += "endif\n\n"



    #decl

    elif (tree.type == "qreg"):
        qregs[tree.param[0]] = [tree.param[1], qcount]
        qcount+=tree.param[1]
        vector_size=qcount


    elif (tree.type == "creg"):
        cregs[tree.param[0]] = tree.param[1]
        for index in range(tree.param[1]):
            output += tree.param[0] + "_" + str(index) + " = 0\n"



    #gatedecl

    elif( tree.type == "gatedecl" ):
        def_gates.append(tree.param[0])
        output += "\nproc " + tree.param[0] + " "
        compileTree(tree.param[1])
```

```
        if (tree.param[2] != None):
            output += ", "
            compileTree(tree.param[2])
        output += "\n\n\t"



    #goplist

    elif (tree.type == "goplist_uop"):
        compileTree(tree.children[0])
        output +="\t"



    #qop

    elif (tree.type == "measure"):
        if tree.param[0].node in qregs:
            if (qregs[tree.param[0].node][0] == cregs[tree.param[1].node]):
                id_reg=tree.param[1].node
                for index in range(qregs[tree.param[0].node][0]):
                    output += "MeasureBit " + str(get_qreg(tree.param[0].node, index)) +"\n"
                    output += id_reg + "_" + str(index) + " = measured_value\n"
                output += "\n"
            else:
                sys.exit("ERROR at line "+ str(tree.param[2]) +": qreg and creg must be of same dimention when measured")
        else:
```

```
            output += "MeasureBit "
            compileTree(tree.children[0])
            output+="\n"
            compileTree(tree.children[1])
            output += " = measured_value\n\n"


    elif (tree.type == "reset"):
        if tree.param[0].node in qregs:
            output += "for i = " + str(get_qreg(tree.param[0].node, 0)) + "; i < " + str(get_qreg(tree.param[0].node, qregs[tree.param[0].node][0])) + "; i++\n\t"
            output += "MeasureBit i \n"
            output+= "endfor\n"
        else:
            output += "MeasureBit"
            compileTree(tree.children[0])
            output+= "\n"



    #uop

    elif (tree.type == "uop_U"):
        ide=tree.param[1]
        params=tree.param[0].split(", ")
        theta=float(params[0])
        phi=float(params[1])
        llambda=float(params[2]) #lambd is a reserved world in python, that's why this variable is called llambda
        r_oo = (math.cos(-(phi + llambda)/2))*math.cos(theta/2)
        r_oi = (math.cos(-(phi - llambda)/2))*math.sin(theta/2)
        r_io = (math.cos((phi - llambda)/2))*math.sin(theta/2)
```

```
        r_ii = (math.cos((phi + llambda)/2))*math.cos(theta/2)
        i_oo = (math.sin(-(phi + llambda)/2))*math.cos(theta/2)
        i_oi = (math.sin(-(phi - llambda)/2))*math.sin(theta/2)
        i_io = (math.sin((phi - llambda)/2))*math.sin(theta/2)
        i_ii = (math.sin((phi + llambda)/2))*math.cos(theta/2)

        if tree.param[1].node in qregs:
            output += "for i = " + str(get_qreg(tree.param[1].node, 0)) + "; i < " + str(get_qreg(tree.param[1].node, qregs[tree.param[1].node][0])) + "; i++\n\t"
            output += "Unitary i "
            output+=", " + str(r_oo) + ", " + str(r_oi) + ", " + str(r_io) + ", " + str(r_ii) +", " + str(i_oo) + ", " + str(i_oi) + ", " + str(i_io) + ", " + str(i_ii) + "\n"
            output+= "endfor\n"
        else:
            output += "Unitary "
            compileTree(tree.children[0])
            output+=", " + str(r_oo) + ", " + str(r_oi) + ", " + str(r_io) + ", " + str(r_ii) +", " + str(i_oo) + ", " + str(i_oi) + ", " + str(i_io) + ", " + str(i_ii) + "\n"


    elif (tree.type == "uop_CX"):

        if (tree.param[0].node in qregs) and (tree.param[1].node in qregs):
            if (qregs[tree.param[0].node][0] == qregs[tree.param[1].node][0] ):
                fin= qregs[tree.param[0].node][0]
                output += "for i=0; i < " + str(fin) + "; i++\n\n\t"
```

```
                output += "CNot "+ str(get_qreg(tree.param[0].node, 0)) +"+i, " + str(get_qreg(tree.param[1].node, 0)) + "+i\nendfor\n\n"
            else:
                sys.exit("ERROR at line "+ str(tree.param[2])+" : cregs must be same dimention\n")
        elif (tree.param[0].node in qregs):
            start = get_qreg(tree.param[0].node, 0)
            fin= start + qregs[tree.param[0].node][0]
            output += "for i=" + str(start) + "; i < " + str(fin) + "; i++\n\n\t"
            output += "CNot i, "
            compileTree(tree.children[1])
            output += "\nendfor\n\n"
        elif (tree.param[1].node in qregs):
            start = get_qreg(tree.param[1].node, 0)
            fin= start + qregs[tree.param[1].node][0]
            output += "for i=" + str(start) + "; i < " + str(fin) + "; i++\n\n\t"
            output += "CNot "
            compileTree(tree.children[0])
            output += ", i\nendfor\n\n"
        else:
            output += "CNot "
            compileTree(tree.children[0])
            output += ", "
            compileTree(tree.children[1])
            output += "\n"




    elif(tree.type == "uop_id"):
```

```
        if (not declared_gate(tree.param[0])):
            sys.exit("ERROR ("+ str(tree.param[1]) + "): the gate '" + tree.param[0] + "' is never declared" )
        for_string += tree.param[0] + " "
        if (len(tree.param)>2):
            for_string += str(tree.param[2]) + ", "

        compileTree(tree.children[0])



    #anylist
    elif (tree.type == "anylist"):
        flag_anylist = True
        compileTree(tree.children[0])
        reg_size = equal_size(gate_regs);
        if (not(gate_regs)):
            output += for_string +"\n"
        elif (reg_size):
            output += "for i = 0; i < " + str(reg_size[1]) + "; i++\n"
            output += "\t"+for_string
            output+= "\nendfor\n\n"
        else:
            sys.exit("ERROR: the gate can be applide only to qregs with the same dimention\n")

        flag_anylist = False
        gate_regs = {}
        for_string = ""


    #idlist
```

```
    elif (tree.type == "ID"):
        if (flag_anylist):
            gate_regs[tree.node] = qregs[tree.node][0]
            for_string += str(get_qreg(tree.node, 0)) + "+i "
        else:
            output += tree.node


    elif(tree.type == "ID_list"):
        if (flag_anylist):
            gate_regs[tree.children[1].node] =
                qregs[tree.children[1].node][0]
            compileTree(tree.children[0])
            for_string += ", "
            compileTree(tree.children[1])

        else:
            compileTree(tree.children[0])
            output += ", "
            compileTree(tree.children[1])



    #mixedlist

    elif (tree.type == "mixedlist1" ):
        qreg=get_qreg(tree.param[0],tree.param[1])
        for_string += str(qreg)


    elif (tree.type == "mixedlist2" ):
        compileTree(tree.children[0])
```

```
        gate_regs[tree.children[1]] = qregs[tree.children[1]][0]
        for_string += ", " + str(get_qreg(tree.children[1], 0)) +
            "+i "


    elif (tree.type == "mixedlist3" ):
        compileTree(tree.children[0])
        for_string += ", "
        qreg=get_qreg(tree.param[0],tree.param[1])
        for_string += str(qreg)



    #argument

    elif (tree.type == "arg_id"):
        output += tree.node

    elif (tree.type == "arg_idint"):
        if tree.param[0] in qregs:
            qreg=get_qreg(tree.param[0],tree.param[1])
            output += str(qreg) + " "
        elif tree.param[0] in cregs:
            creg=get_creg(tree.param[0],tree.param[1])
            output += creg
        else:
            sys.exit("ERROR : the reg" + str(tree.param[0]) + "is
                not defined")


    else:
        for child in tree.children:
            compileTree(child)
```

```


@click.command(context_settings={'help_option_names':['-h','--help']},
help='Compiler form a .qasm file in to a .qscript file, where
    FILE_IN is the path of the .qasm file to be translated')
@click.option('--qscript', '-q', default='outfile', help='Used to
    specify the name of the output file. If none is given, the name
    of the .qasm file will be used')
@click.option('--table', '-t', is_flag=True, help='With this flag,
    the compiler will NOT save the parse table, but create it every
    time')
@click.argument('file_in')

#Main function

def main(qscript, file_in, table):

    with open(file_in, "r") as f:
        data=f.read()

    result=pars(data, table)

    compileTree(result)

    if (vector_size < 6 or vector_size > 22 or vector_size % 2 !=
        0):
        sys.exit("VectorSize should be even, more than 6 and less
            than 22 " + str(vector_size))

    final = "VectorSize " + str(vector_size) +"\n" + output


    if (qscript == 'outfile'):
```

```
        if file_in.endswith('.qasm'):
            qscript = file_in[:-5]
        qscript += ".qscript"

    with open(qscript,"w") as f:
        f.write(final)

if __name__ == "__main__":
    main()
```

Bibliography

[1] Altenkirch, Thorsten, and Jonathan Grattage. *"A functional quantum programming language."* Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on. IEEE, 2005.

[2] Beazley, David. *"Ply (python lex-yacc) v3.10"* See `http://www.dabeaz.com/ply` (2001).

[3] Cross, Andrew W., et al. *"Open quantum assembly language."* arXiv preprint arXiv:1707.03429 (2017).

[4] Feynman, Richard P. *"Simulating physics with computers."* International journal of theoretical physics 21.6-7 (1982): 467-488.

[5] Green, Alexander S., et al. *"Quipper: a scalable quantum programming language."* ACM SIGPLAN Notices. Vol. 48. No. 6. ACM, 2013.

[6] Grover, Lov K. *"Quantum mechanics helps in searching for a needle in a haystack."* Physical review letters 79.2 (1997): 325.

[7] Johnson, Stephen C. *"YACC: yet another compiler-compiler."* Bell Laboratories (1986).

[8] Kandala, Abhinav, et al. *"Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets."* Nature 549.7671 (2017): 242.

[9] Shor, Peter W. *"Algorithms for quantum computation: Discrete logarithms and factoring."* Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. Ieee, 1994.

[10] Shor, Peter W. *"Scheme for reducing decoherence in quantum computer memory."* Physical review A 52.4 (1995): R2493.

[11] Kalai, Gil, and Guy Kindler. *"Gaussian noise sensitivity and BosonSampling."* arXiv preprint arXiv:1409.3093 (2014).

[12] Lesk, Michael E., and Eric Schmidt. *"Lex: A lexical analyzer generator."* (1975).

[13] Moll, Nikolaj, et al. *"Quantum optimization using variational algorithms on near-term quantum devices."* arXiv preprint arXiv:1710.01022 (2017).

[14] Nielsen, Michael A., and Isaac L. Chuang. *"Quantum computation and quantum information."* (2000). Chapter 8.3.6.

[15] Ömer, Bernhard. *"Quantum Programming in QCL"* (2000). Available at `http://tph.tuwien.ac.at/$~oemer/qcl.html`.

[16] RistÃ¨, Diego, et al. *"Demonstration of quantum advantage in machine learning."* npj Quantum Information 3.1 (2017): 16.

[17] *Click* can be found at `http://click.pocoo.org/`

[18] Google's *"Quantum Playground"* can be found at `http://www.quantumplayground.net/#/home`

[19] IBM's *"Quantum Experience"* can be found at `https://quantumexperience.ng.bluemix.net/qx/experience`

[20] *QISKit* can be found at `https://www.qiskit.org/`