UNIVERSIDADE FEDERAL DE SANTA MARIA
PROGRAMA DE CIÊNCIA DA COMPUTAÇÃO

João Gabriel da Cunha Schittler

COMPILAÇÃO E EXECUÇÃO DE CÓDIGO DA LINGUAGEM QML NO COMPUTADOR QUÂNTICO DA IBM

Santa Maria, RS
2022

João Gabriel da Cunha Schittler

COMPILAÇÃO E EXECUÇÃO DE CÓDIGO DA LINGUAGEM QML NO COMPUTADOR QUÂNTICO DA IBM

Trabalho de Conclusão de Curso apresentado ao Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Bacharel em Ciência da Computação**

Orientadora: Profa. Dra. Juliana Kaizer Vizzotto

501
Santa Maria, RS
2022

da Cunha Schittler, João Gabriel

Compilação e Execução de código da linguagem QML no Computador Quântico da IBM / por João Gabriel da Cunha Schittler. – 2022.
75 f.: il.; 30 cm.

Orientadora: Juliana Kaizer Vizzotto
Trabalho de Conclusão de Curso - Universidade Federal de Santa Maria, Universidade Federal de Santa Maria, Ciência da Computação, RS, 2022.

1. Computação Quântica. 2. Semântica Formal. 3. Compiladores. I. Kaizer Vizzotto, Juliana. II.Compilação e Execução de código da linguagem QML no Computador Quântico da IBM.

E-mail: jgschittler@inf.ufsm.com

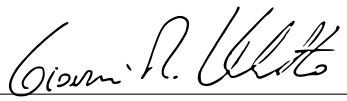
João Gabriel da Cunha Schittler

COMPILAÇÃO E EXECUÇÃO DE CÓDIGO DA LINGUAGEM QML NO COMPUTADOR QUÂNTICO DA IBM

Trabalho de Conclusão de Curso apresentado ao Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Bacharel em Ciência da Computação**

Aprovado em 11 de Fevereiro de 2022:

Juliana Kaizer Vizzotto, Profa. Dra. (UFSM)
(Presidente/Orientadora)



Giovani Librelotto, Prof. Dr. (UFSM)

Jonas Maziero, Prof. Dr. (UFSM)

Santa Maria, RS

2022

RESUMO

COMPILAÇÃO E EXECUÇÃO DE CÓDIGO DA LINGUAGEM QML NO COMPUTADOR QUÂNTICO DA IBM

AUTOR: JOÃO GABRIEL DA CUNHA SCHITTLER
ORIENTADORA: JULIANA KAIZER VIZZOTTO

Nesse trabalho é apresentado um compilador para a linguagem de programação quântica funcional QML, que possui estruturas para controle quântico, juntamente com um ambiente de execução de código compilado no computador quântico da IBM, o IBMQ. Foram usadas as ferramentas FLEX e BISON, juntamente com um programa em C++, para implementar o *parser* da linguagem e o pacote Qiskit com um programa em Python, para a construção e execução dos circuitos quânticos. O compilador consegue corretamente traduzir programas QML que usam o tipo qubit simples. Alguns exemplos simples que são compilados e executados corretamente no IBMQ são apresentados.

Palavras-chave: Computação Quântica. Semântica Formal. Compiladores.

ABSTRACT

COMPILING AND EXECUTING QML LANGUAGE CODE ON IBM QUANTUM COMPUTERS

AUTHOR: JOÃO GABRIEL DA CUNHA SCHITTLER
ADVISOR: JULIANA KAIZER VIZZOTTO

In this work we present a compiler for the functional quantum programming language QML, which has structures for quantum control, together with a code execution environment compiled on IBM's quantum computer, the IBMQ. The FLEX and BISON tools were used, together with a program in C++, to implement the language's *parser* and the Qiskit package with a program in Python, for the construction and execution of the quantum circuits. The compiler can correctly translate QML programs that use the basic type of qubits. Some simple examples that compile and run correctly on IBMQ are presented.

Keywords: Quantum Computing. Formal Semantics. Compilers.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

MDT	Manual de Dissertação e Tese
UFSM	Universidade Federal de Santa Maria
CQ	Computação Quântica
ER	Expressão Regular
GLC	Gramática Livre de Contexto

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTO E MOTIVAÇÃO

A computação quântica (CQ) tem sua origem no trabalho de Richard Feynman (FEYNMAN, 1982), onde o autor trata questões sobre simular sistemas físicos quânticos com a utilização de computadores clássicos. Feynman faz a conjectura sobre utilizar fenômenos da mecânica quântica, como superposição e emaranhamento, para resolver problemas mais rapidamente que na computação clássica. Essa questão levantada por Feynman levou a algumas propostas para um modelo quântico de computação. Porém, a realização de que realmente pode-se ter algum ganho veio somente alguns anos depois, com os resultados de Grover (GROVER, 1996) e Shor (SHOR, 1997). Grover propôs um algoritmo quântico para executar busca em registros desordenados, com um ganho quadrático na complexidade temporal, comparando com qualquer algoritmo clássico conhecido. O trabalho de Shor definiu um algoritmo quântico de tempo polinomial para fatorar números inteiros. Para esse problema somente se conhece algoritmos clássicos com tempo exponencial. Pode-se dizer que esses foram os primeiros resultados que geraram um grande interesse de pesquisadores da ciência de computação.

Assim, criada como uma alternativa para a computação clássica, um dos objetivos atuais da CQ é atingir a *supremacia quântica* (PRESKILL, 2012), i.e., provar que sistemas quânticos podem solucionar problemas que não poderiam ser solucionados por sistemas clássicos em tempo viável.

Uma das dificuldades no desenvolvimento de novos algoritmos quânticos é a complexidade dos circuitos quânticos que são executados pelos computadores quânticos. Nesse contexto, uma linguagem de programação quântica de alto nível tem por objetivo abstrair as complexidades desses circuitos ao programador, oferecendo uma interface para que seja possível descrever algoritmos quânticos de maneira mais simples.

Principalmente com o objetivo de investigar a área da programação quântica e proporcionar o desenvolvimento e estudo de algoritmos quânticos, as linguagens de programação quânticas de alto nível têm sido desenvolvidas nos últimos anos. A linguagem funcional quântica QML (ALTENKIRCH; GRATTAGE, 2005) é um exemplo. QML é baseada nas construções de alto nível das linguagens funcionais convencionais e integra computações quânticas reversíveis e irreversíveis, usando lógica linear de primeira ordem para os *weakenings*, i.e., para controlar

o descarte de valores quânticos. Os programas *estritos* não apresentam decoerência quântica e, consequentemente, preservam superposições e emaranhamento, que são essenciais para o paralelismo quântico. Uma outra característica importante da linguagem QML é que ela segue o paradigma "dados e controle quânticos", i.e., ela suporta estruturas de controle quântico, que permite que sejam executadas operações condicionais quânticas. Além disso, os autores apresentam uma maneira natural de compilar os termos funcionais da linguagem em circuitos quânticos, ou seja, traduzir um programa seguindo as especificações da linguagem para circuitos quânticos. Essa especificação de compilador usa uma estrutura de pilha para armazenar os qubits, as operações unitárias são executadas no topo da pilha, sendo possível, também, fazer operações auxiliares que modificam a ordem dos qubits da pilha.

O computador quântico da International Business Machines (IBM) e sua plataforma IBM Quantum Experience (IBM-QE) (IBM QUANTUM WEBSITE, 2021) proporcionam o uso de um computador quântico em nuvem, que pode ser acessado de forma remota por qualquer um. O IBM-QE fornece, entre outros serviços, uma interface gráfica para a criação e execução de circuitos quântico, chamado IBM Quantum composer. Além disso, existe uma SDK open-source para a linguagem Python, chamada Qiskit (ANIS et al., 2021), que possibilita ao programador conectar-se com o IBM-QE, montar circuitos equivalentes aos circuitos que podem ser montados no composer, selecionar qual computador (ou simulador) quântico o circuito deve ser executado e então enviar esse circuito para ser executado, recebendo os resultados após a execução do mesmo.

1.2 OBJETIVO GERAL

O objetivo geral do presente trabalho de graduação é desenvolver um compilador para a linguagem de programação quântica QML e executar os circuitos gerados pelo compilador no computador quântico da IBM, através da integração com o pacote Qiskit (ANIS et al., 2021).

Como objetivos específicos destacamos:

- estudar os princípios da área de computação quântica;
- estudar a linguagem de programação quântica QML;
- estudar a descrição da compilação das construções da linguagem QML para circuitos quânticos;

- investigar sobre a construção de compiladores para linguagens de programação de alto nível;
- estudar a plataforma IBM-QE e a biblioteca Qiskit;
- projetar um compilador para QML baseado na sua compilação para circuitos quânticos e sua tradução para o Qiksit;
- implementar o compilador com dois programas: o primeiro deles , o parser, que vai ler o programa QML e traduzir ele em portas lógicas quânticas, e o segundo deles, o montador, vai ler a saída do parser, montar o circuito Qiskit e o enviar para um computador quântico do IBM-QE

Em seguida, será feita uma revisão bibliográfica, explicando conceitos de Computação Quântica e Compiladores. Após isso, o trabalho desenvolvido será apresentado, com todas as suas partes explicadas. E então, será feita uma conclusão, explicando possíveis trabalhos futuros.

2 COMPUTAÇÃO QUÂNTICA

2.1 INTRODUÇÃO

A Computação quântica (NIELSEN; CHUANG, 2011), considerando um ponto de vista algorítmico, apresenta um novo modelo de computação, que incorpora novas maneiras de projetar e estruturar algoritmos utilizando as características da física quântica.

2.2 QUBITS E SUAS PROPRIEDADES

A unidade básica de informação clássica computável é o bit, um sistema físico clássico binário. Em computação quântica, a unidade básica de informação é representada pelo bit quântico ou qubit, um sistema físico quântico binário. O qubit é comumente representado como uma superposição de estados, através da notação *braket*[1] de Dirac: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

Os números complexos α e β representam as amplitudes de probabilidade de medir o qubit como os estados $|0\rangle$ e $|1\rangle$ respectivamente. O qubit pode ser definido como um vetor em um espaço vetorial complexo (espaço de Hilbert), tal que $|\alpha|^2 + |\beta|^2 = 1$.

Formalmente, a combinação de dois ou mais sistemas quânticos pode ser obtida usando uma operação de produto tensorial $\otimes$. Se $q = \alpha|0\rangle + \beta|1\rangle$ e $p = \gamma|0\rangle + \delta|1\rangle$ são dois qubits não co-relacionados, ao aplicar o produto tensorial temos $q \otimes p = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$. Outro tipo de operação sobre os bits quânticos é a *medição*, esta é uma operação não reversível. Ela é obtida com um colapso na função de onda de um sistema quântico. Dessa forma, ao aplicar a medição na base $\{|0\rangle, |1\rangle\}$ em um sistema quântico com estado $\alpha|0\rangle + \beta|1\rangle$ obtemos $|0\rangle$ com $|\alpha|^2$ de probabilidade associada e $|1\rangle$ com $|\beta|^2$ de probabilidade associada.

2.2.1 Operações e Circuitos Quânticos

Existem dois tipos de operações em bits quânticos: *transformações unitárias* e *medições*, a primeira muda o estado do qubit e a segunda observa o estado do qubit para achar o seu valor. As *transformações unitárias* são operações reversíveis, ou seja, não há perda de informação sobre o qubit, *medições*, por outro lado, colapsam o estado quântico do qubit, sendo impossível de reverter ao estado original depois da medição.

[1] O nome *braket* surge através de uma convenção em que um vetor coluna é chamado "ket" e sua notação é demonstrada por $|\ \rangle$ e um vetor linha é chamado "bra" e tem como notação $\langle\ |$.

Uma *transformação unitária* pode ser representada como uma matriz unitária, ou seja, para um estado de *N* qubits, é necessário de uma matriz de tamanho 2^N x 2^N.

Por exemplo, uma transformação T pode ser aplicada a um estado $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ através de uma matriz do formato:

$$T|\phi\rangle = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Essas matrizes unitárias são normalmente chamadas de *portas quânticas* e são usadas no desenvolvimento de *circuitos quânticos*. Alguns exemplos de portas quânticas importantes são:

$$\textit{NOT} \text{ ou Pauli-X } = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

$$\text{Hadamard ou } H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

$$\textit{CNOT} \text{ ou Not controlado} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

$$\text{U3}(\theta, \phi, \lambda) \text{ ou Rotação Universal} = \begin{pmatrix} cos(\theta/2) & -sin(\theta/2) * e^{i*\lambda} \\ sin(\theta/2) * e^{i*\phi} & cos(\theta/2) * e^{i*(\lambda+\phi)} \end{pmatrix}.$$

A porta NOT muda os coeficientes de $\alpha|0\rangle + \beta|1\rangle$ para $\beta|0\rangle + \alpha|1\rangle$, ou seja, troca o valor falso pelo verdadeiro e vice-versa. A porta CNOT é uma porta NOT controlada, onde a porta NOT é aplicada ao segundo qubit se o primeiro qubit (chamado de qubit de controle) for 1, caso contrário, não faz nada. A porta Hadamard é usada com frequência, ela transforma um estado quântico em uma *superposição coerente*, onde ambos os estado-base possuem pesos iguais. A porta U3 é chamada de porta universal, pois ela pode realizar qualquer transformação em um estado de 1 qubit, ou seja, qualquer outra porta de 1 qubit pode ser descrita em uma porta U3 através dos seus 3 parâmetros.

A figura 1 mostra um circuito quântico usando algumas das portas apresentadas acima.

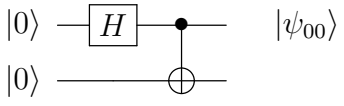


Figura 1 – Circuito quântico usando uma porta de Hadamard e uma porta Not controlada.

No circuito quântico os fios representam qubits e as operações são portas lógicas quânticas.

2.3 ALGORITMOS QUÂNTICOS

Um exemplo de algoritmo quântico que usa essas portas lógicas é o Algoritmo quântico de Deutsch (DEUTSCH; JOZSAT, 1992), que determina se uma função $\mathbf{f} : \mathbf{0}, \mathbf{1} \rightarrow \mathbf{0}, \mathbf{1}$ é constante ($\mathbf{f}(\mathbf{0}) = \mathbf{f}(\mathbf{1})$) ou balanceada ($\mathbf{f}(\mathbf{0})! = \mathbf{f}(\mathbf{1})$).

Para fazer essa verificação em um computador clássico, seria necessário calcular $\mathbf{f}(\mathbf{0})$, $\mathbf{f}(\mathbf{1})$ e comparar os dois. Em um computador quântico, basta rodar o circuito da Figura 2.

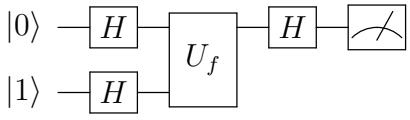


Figura 2 – Circuito do Algoritmo de Deutsch.

Ao medir o primeiro qubit, se ele for 0 significa que a função é constante, senão a função é balanceada. Esse algoritmo pertence a uma classe de algoritmos chamados de algoritmos de consulta, pois assumem uma implementação de um Oráculo U_f, que calcula $\mathbf{f}(x)$. A vantagem desse algoritmo ao clássico é que precisa-se consultar o oráculo menos vezes.

Existe uma extensão desse algoritmo, chamado algoritmo de Deutsch-Jozsa, que funciona para funções com domínio maior que 1 qubit.

2.4 IBM QUANTUM EXPERIENCE E QISKIT

A empresa IBM criou um serviço chamado de IBM Quantum Experience (IBM QUANTUM WEBSITE, 2021), que disponibiliza gratuitamente diversos computadores e simuladores quânticos para desenvolvedores. Esse site possui uma interface gráfica de montagem de circuitos como apresentado na Figura 3, onde é possível posicionar diversas portas quânticas no circuito e adicionar qubits novos.

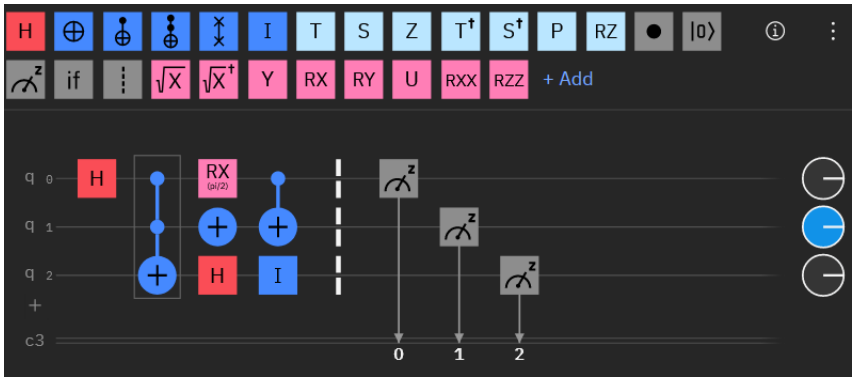


Figura 3 – Compositor de Circuitos do IBMQ.

Depois de montar um circuito, pode-se selecionar um computador/simulador para execução, juntamente com o número de execuções do circuito. Recomenda-se usar um valor alto para minimizar erro no resultado final. Esse procedimento pode ser analisado na Figura 4.

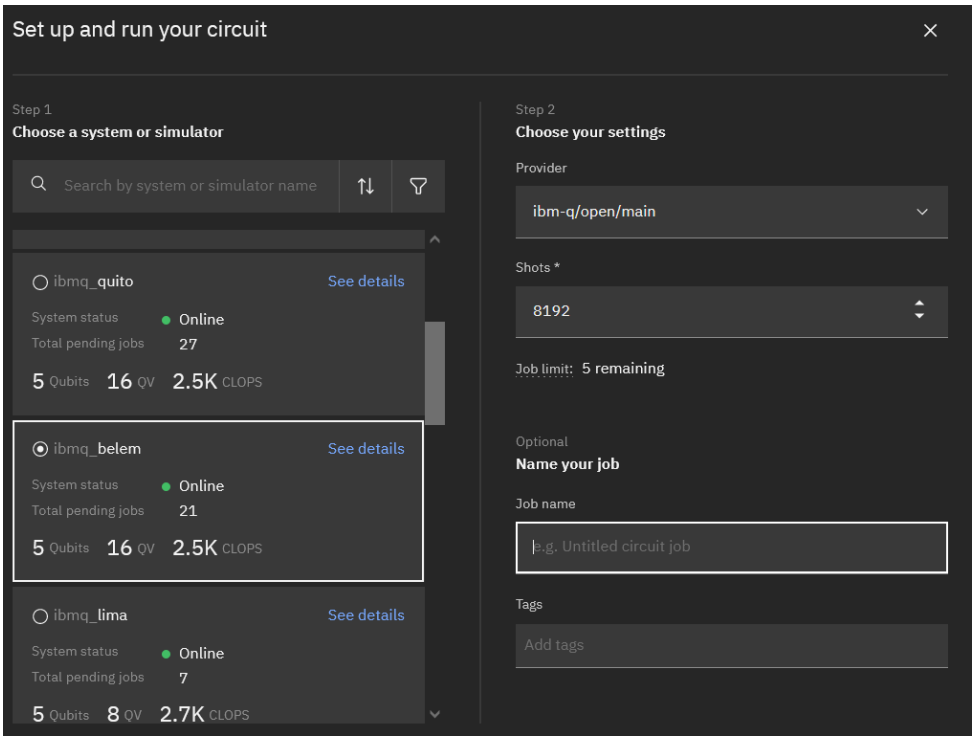


Figura 4 – Seleção de ambiente de execução do IBMQ.

Com o pacote Qiskit, é possível realizar as funcionalidades principais do IBMQ via código Python. A Figura 5 mostra um exemplo de código que monta um circuito igual ao da Figura 3.

```python
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(3, 'q')
creg_c = ClassicalRegister(3, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.h(qreg_q[0])
circuit.ccx(qreg_q[0], qreg_q[1], qreg_q[2])
circuit.rx(pi/2, qreg_q[0])
circuit.x(qreg_q[1])
circuit.h(qreg_q[2])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.id(qreg_q[2])
circuit.barrier(qreg_q[2])
circuit.barrier(qreg_q[0])
circuit.barrier(qreg_q[1])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
circuit.measure(qreg_q[2], creg_c[2])
```

Figura 5 – Código Qiskit para criar o circuito da Figura 3

O Qiskit também permite a comunicação com os computadores disponibilizados pelo IBMQ usando as seguintes funções.

```
provider = IBMQ.get_provider('ibm-q')
qcomputer = provider.get_backend('ibmq_belem')
job = execute(QCircuit, backend = qcomputer, shots = nShots)
result = job.result()
```

A função $IBMQ.get\_provider$ retorna um provedor de serviços específico, no caso desse trabalho, estamos usando o ibm-q de acesso público. A função $get\_backend$ permite selecionar um computador quântico específico do provedor. A função $execute$, recebe o circuito, onde esse circuito vai ser executado e quantas vezes esse circuito vai ser executado, retornando a requisição de execução. A função $job.result()$ espera a execução completa terminar, e retorna os resultados dela.

3 LINGUAGEM QML

A linguagem QML é uma linguagem de programação funcional quântica, que apresenta estruturas de controle quânticas. Em questões sintáticas e semânticas ela se assemelha muito com a linguagem de programação funcional Haskell (HASKELL LANGUAGE WEBSITE, 2022). O diferencial da linguagem QML é o **if**^{0}, apresentado na sua forma mais geral como **case**^{0}. Esse **if** quântico é uma estrutura de controle que não realiza a medição do qubit, o valor retornado por esse **if**^{0} é uma combinação dos valores do **then** e do **else**, de acordo com as amplitudes de probabilidade do valor de controle do **if**^{0}.

O **if**^{0} possui uma restrição, os valores do **then** e do **else** devem ser ortogonais, para que a operação seja reversível.

3.1 SINTAXE

Programas na linguagem QML são expressões tipadas definidas pelas seguintes regras:

$$
\begin{aligned}
&\text{(Tipos)} \quad \sigma \;\; ::= 1 \mid \sigma \oplus \tau \mid \sigma \otimes \tau \\
&\text{(Termos)} \;\; t \;\; ::= x \mid \textbf{let} \;\; x \;=\; t \;\; \textbf{in} \;\; u \\
&\qquad\qquad\quad (\,) \mid x \uparrow ys \\
&\qquad\qquad\quad (t, u) \mid \;\; \textbf{let} \; (x, y) = t \;\; \textbf{in} \;\; u \\
&\qquad\qquad\quad \text{qintL } t \mid \text{qinR } t \\
&\qquad\qquad\quad \textbf{case} \;\; t \; \textbf{of} \; \{\text{qintL} \;\; x \to u \mid \text{qintR} \;\; y \to \;\; u'\} \\
&\qquad\qquad\quad \textbf{case}^{\,0} \; t \; \textbf{of} \; \{\text{qintL} \;\; x \to u \mid \text{qintR} \;\; y \to \;\; u'\} \\
&\qquad\qquad\quad \{ \; (k) \; t \mid (L) \; u \; \}
\end{aligned}
$$

Os construtores de tipo da linguagem QML são o produto tensorial, $\otimes$, i.e., o tipo produto, o coproduto tensorial, $\oplus$, que corresponde a união disjunta em linguagens de programação convencionais. Os qubits não são primitivos, mas podem ser definidos como $Q_2 = 1 \oplus 1$, usando a união disjunta. A linguagem também tem duas construções de controle **case**: uma que mede o qubit de controle e outra, **case**^{0}, que não mede o qubit de controle, mas requer que os *ramos* de escolha estejam sempre em subespaços ortogonais. Para definir superposições usa-se os números complexos K e $L \in \mathbf{C}$, representando as amplitudes de probabilidades. Um programa

na linguagem QML é uma sequência de definições de funções de acordo com a sintaxe acima. Como um exemplo de um programa em QML, considere a codificação do algoritmo de Deutsch.

$$
\begin{array}{l}
had: \ Q2 \ \rightarrow Q2 \\
had\ x = \textbf{if}^0\ x \\
\qquad\qquad \textbf{then}\ \{\textbf{qfalse} \mid (-1)\textbf{qtrue}\} \\
\qquad\qquad \textbf{else}\ \{\textbf{qfalse} \mid \textbf{qtrue}\} \\
\\
deutsch: \ Q2 \ \rightarrow Q2 \ \rightarrow Q2 \\
deutsch\ a\ b = \\
\qquad\qquad \textbf{let}(x,y) = \ \textbf{if}^0\{\ \textbf{qfalse} \mid (-1)\ \textbf{qtrue}\ \} \\
\qquad\qquad\qquad\qquad \textbf{then}(\ \textbf{qtrue}\ ,\ \textbf{if}^0\ a \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}(\ \{\textbf{qfalse} \mid (-1)\textbf{qtrue}\ \},\ (\ \textbf{qtrue}\ ,\ b\ )\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}(\ \{(-1)\ \textbf{qfalse} \mid \textbf{qtrue}\ \},\ (\ \textbf{qfalse}\ ,\ b\ )\ )\ ) \\
\qquad\qquad\qquad\qquad \textbf{else}(\ \textbf{qfalse}\ ,\ \textbf{if}^0\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}(\ \{(-1)\ \textbf{qfalse} \mid \textbf{qtrue}\ \},\ (\ a\ ,\ \textbf{qtrue}\ )\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}(\ \{\textbf{qfalse} \mid (-1)\textbf{qtrue}\ \},\ (\ a\ ,\ \textbf{qfalse}\ )\ )\ ) \\
\qquad\qquad \textbf{in}\ had\ x
\end{array}
$$

Esse programa, dependendo do qubit de entrada x, retorna uma das duas possíveis superposições de qubits, representadas pelos termos após o **then** e **else**. É importante notar que o **if**^{0} é um caso especial do **case**^{0}. Pelo semântica da linguagem podemos verificar que aplicando essa operação had duas vezes, retornamos ao qubit original, cancelando as amplitudes. Ainda é interessante salientar que **qtrue** e **qfalse** são açúcar sintático para os termos qinL () e qinR ().

Além disso, o sistema de tipos da linguagem QML é baseado na lógica linear *estrita*, i.e., lógica linear com contração, mas sem *weakenings* implícitos. Isto é possível, pois a contração de variáveis é modelada, como linguagens de programação funcional clássicas como compartilhamento (*sharing*) e não como duplicação (ou clonagem). Na semântica de circuitos da linguagem, o compartilhamento de variáveis é implementado utilizando a porta quântica Not controlada $CNOT$.

3.2 SEMÂNTICA

A semântica operacional da linguagem QML é definida através da tradução de programas para computações quânticas representadas como circuitos modelados como uma sequência de FQCs (*finite quantum computations*). As FQCs são representadas com a seguinte quíntupla $\textbf{FQC} = \{A, H, B, G, \phi\}$, onde A representa o número de qubits da entrada, H representa o

número de qubits auxiliares necessários (ou *heap*), B representa o número de qubits da saída, G representa o número de qubits lixo gerados e ϕ representa uma operação unitária reversível, como ilustrado na Figura 6.

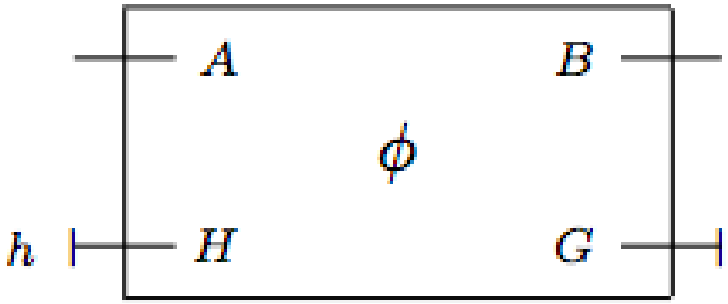


Figura 6 – FQC Simples

As operações Unitárias por padrão são aplicadas no primeiro qubit do circuito, para acessar e alterar os outros qubits é necessário usar o operador $\otimes$. Além das operações unitárias, existem as operações $swapN$, $Permutaion$ e $Cond$. A operação $swapN$ trasporta um bloco de qubits de qualquer lugar da pilha para o topo e possui 3 parâmetros: "o", o *offset* inicial, "l", o tamanho do bloco e "n", o tamanho total da pilha. A operação $Permutation$ recebe uma lista de índices de qubits e os coloca no topo da pilha. E a operação $Cond$ usa o qubit no topo da pilha como condição para uma operação unitária, seguido pela negação desse qubit para servir como condição de outro operação unitária.

Dados dois sistemas computacionais, eles podem ser compostos combinando as *heaps* iniciais e finais como ilustrado na Figura 7.

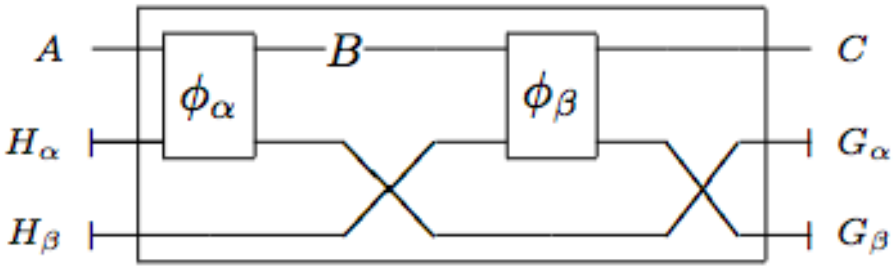


Figura 7 – FQC Composta

A implementação do compilador apresentado nesse trabalho considera uma tradução das construções sintáticas da linguagem para FQCs.

4 COMPILADORES

Compiladores (AHO et al., 2007) são programas que traduzem código escrito em uma linguagem fonte para uma linguagem alvo. Normalmente, compiladores são construídos para traduzir código de linguagens de nível alto para alguma linguagem de baixo nível, como linguagem de máquina. As partes dos compiladores são:

- *análise léxica*;
- *análise sintática*;
- *análise semântica*;
- *geração de código intermediário*;
- *otimização de código intermediário*;
- *geração de código da linguagem alvo*.

A *análise léxica* lê o programa fonte e traduz ele para um conjunto de símbolos léxicos, também chamados de tokens. Esses tokens variam de linguagem fonte em linguagem fonte, mas normalmente são palavras reservadas, variáveis, constantes ou operadores da linguagem. Os tokens, além do seu identificador, podem conter alguma outra informação, esses dados adicionais são chamados de atributos e eles são úteis para diferenciar tokens da mesma classe mas com valores diferentes, como por exemplo dois tokens constante "1" e "2" podem conter um atributo que guarda o valor inteiro dele.

A detecção dos diferentes tokens pode ser feita através de Expressões Regulares (ER), que são sequências de caracteres e símbolos representando uma certa linguagem. O símbolo "|" indica uma possibilidade de escolha entre o que esta na esquerda ou direita do símbolo , o símbolo "*" indica uma sequência de tamanho maior ou igual a zero, o símbolo "+" indica uma sequência de tamanho pelo menos um, o símbolo.Por exemplo, a expressão "alb"representa uma linguagem que pode ser "a" ou "b", a expressão "(alb)*" representa uma linguagem que é uma sequência de tamanho >= 0 de "a"s ou "b"s. Com ERs, é possível detectar tokens com padrões de alta variação, como números reais, que podem apresentar parte fracionaria e podem ser negativos. A ferramenta FLEX (FLEX WEBSITE, 2022) realiza a análise léxica baseada em um arquivo de especificação.

A *análise sintática* recebe os tokens criados pela análise léxica e verifica se eles estão gramaticalmente corretos de acordo com a sintaxe da linguagem. Uma forma comum de implementar a análise sintática é com uma Gramática Livre de Contexto (GLC). Uma GLC é uma quadrupla {V,T,P,S}, onde V é o conjunto de símbolos não terminais, T é o conjunto dos símbolos terminais, P é o conjunto de regras de produção (ou produções), e S é o símbolo inicial da gramática. Uma produção de uma GLC pode conter símbolos terminais e não terminais em qualquer ordem.

Exemplificando, a seguinte GLC cria palíndromos com o alfabeto "a"e "b".

G = { {S},{a,b},P,S }

P = {

$\langle S \rangle \rightarrow$ aSa

$\langle S \rangle \rightarrow$ bSb

}

As produções da GLC da análise sintática são usadas para representar a sintaxe da linguagem, implicitamente ou explicitamente, em formato de árvore, chamada árvore de derivação sintática. A ferramenta (BISON WEBSITE, 2022) realiza a análise sintática e semântica baseada em um arquivo de especificação da gramática, as ferramentas FLEX e BISON podem ser usadas em conjunto, simplificando o processo da implementação da análise léxica, sintática e semântica.

A *análise semântica* acontece em conjunto com a análise sintática e verifica se o conteúdo do que foi escrito faz sentido. Como ela acontece em conjunto com a sintática, podem ser posicionadas ações semânticas entre os símbolos das regras da GLC, esses termos podem atribuir valores aos atributos dos símbolos da gramática e podem chamar funções do compilador geral.

A *geração de código intermediário* é uma fase opcional que, a partir da representação sintática e semântica do código fonte, gera-se um código intermediário. Esse código intermediário não apresenta certos detalhes, como os registradores e endereços de memória que vão ser usados. Códigos intermediários possibilitam um maior grau de otimização de código.

A *otimização de código* visa a deixar o código intermediário mais eficiente em termos de velocidade de execução e espaço de memória.

A *geração de código* da linguagem alvo é feita a partir do código intermediário, ou pode

ser feita diretamente durante a análise sintática/semântica, caso não for implementada a fase de código intermediário. Caso seja usado código intermediário, essa fase vai adicionar os detalhes omitidos anteriormente.

5 IMPLEMENTAÇÃO DO COMPILADOR PARA QML

O presente capítulo apresenta detalhadamente a implementação do compilador implementado neste trabalho. Primeiramente, será apresentada a arquitetura geral do trabalho. Em seguida, será explicada a implementação do parser QML e do construtor e executor de circuitos. E por fim serão apresentados exemplos de uso do trabalho desenvolvido. Os arquivos desenvolvidos estão disponíveis no anexo desse texto e pela url "https://github.com/JoaoSchittler/QMLCompiler"

5.1 ARQUITETURA GERAL DO TRABALHO

A arquitetura desenvolvida no trabalho é apresentada na Figura 8.

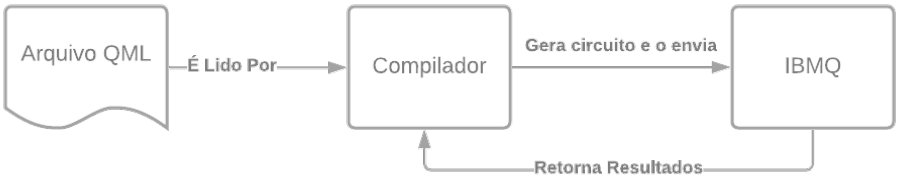


Figura 8

Um programa qualquer seguindo o padrão da linguagem QML é lido pelo compilador. O compilador vai traduzir esse programa para um circuito quântico, que é enviado para execução em algum computador quântico disponibilizado pelo serviço IBMQ. Os resultados da computação são retornados e então mostrados a quem está usando o compilador.

A estrutura interna no compilador funciona como ilustrado na Figura 9.

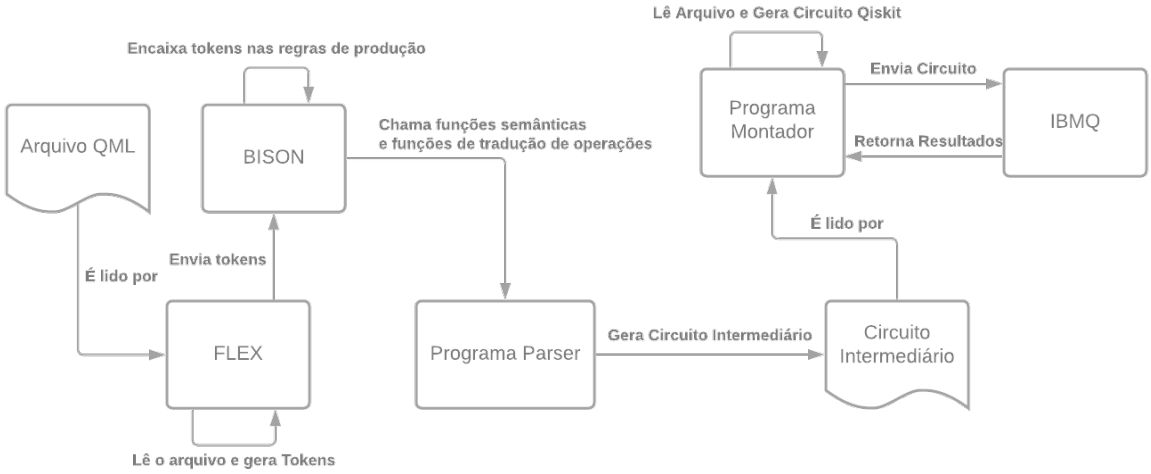


Figura 9

O programa QML é lido pelo programa parser, que por sua vez envia o texto lido para a ferramenta FLEX, que a partir de um arquivo de especificação, realiza a análise léxica da entrada, gerando tokens. Esses tokens são enviados para a ferramenta BISON, que a partir de um arquivo de especificação, realiza a análise sintática e semântica. Durante execução do BISON, são chamadas funções de classes internas do parser, que tratam de fazer verificações semânticas, como verificar se uma função sendo chamada já foi declarada antes, e também armazenam as informações necessárias para que a tradução do programa seja feita, como quais operações são feitas por cada função do programa. Após o programa QML ser lido por completo, um arquivo intermediário é gerado, contendo instruções de como montar o circuito que implementa o programa QML de entrada. Esse arquivo por sua vez é lido pelo programa montador, que monta o circuito descrito pelo arquivo, envia para a execução nos computadores da IBM e apresenta os resultados.

5.2 IMPLEMENTAÇÃO DO PARSER

O parser para a linguagem QML foi desenvolvido usando a linguagem C++ e as ferramentas *FLEX* e *BISON*.

5.2.1 Análise Léxica

A análise léxica é realizada pela ferramenta *FLEX*, que, a partir de uma especificação (arquivo .l) e um arquivo de entrada, gera uma sequência de tokens. O arquivo .l segue o

seguinte formato :

- %{ Funções auxiliares, inclusão de bibliotecas %}
- %% Lista de Regras %%

Dentro das regras, podem ser utilizadas expressões regulares para descrever tokens mais complexos, como nomes de variáveis e números constantes. Para facilitar a escrita, foram criadas duas macros:

- DIGIT [0-9], ou seja, DIGIT pode ser um número de 0 a 9
- LETTER [a-z]|[A-Z], ou seja, LETTER pode ser qualquer letra minúscula ou qualquer letra maiúscula

O anexo A contém o arquivo .l desenvolvido para esse trabalho. As regras são uma *string*, que pode ter uma ER, seguida por uma ação semântica. Em seguida, serão destacadas duas regras desse arquivo.

```
{LETTER}({LETTER}|{DIGIT})*
("-"|"+")?{DIGIT}+("."{DIGIT}+)?
```

A primeira regra é responsável pela detecção de nomes de variáveis e funções. Ela pode ser entendida como: Letra, podendo ser seguida por uma sequência de letras ou dígitos. Quando isso é detectado, é adicionado a *string* lida (`yytext`) ao *token* ID retornado (`yylval`). A segunda regra é detecta um número real, ela pode ser entendida como: O sinal do número (opcional) seguido por uma sequência de pelo menos um digito e então opcionalmente seguida por um "."e uma sequência de pelo menos um dígito. Quando o número é detectado, é chamada a função `ParseComplex`, que gera um número complexo a partir de dois valores *float*, a parte real é uma conversão de *string* para *float* de `yytext`, e a parte imaginária é 0.

Os valores `yytext` e `yylval` são acessíveis nesse arquivo, pois é incluído o arquivo header gerado pelo BISON, que define todos os *tokens*, `yylval` e `yytext`.

Esses atributos serão explicados com maiores detalhes na seção da Análise Semântica, mas foram apresentados aqui para mostrar que a análise semântica está envolvida em todas as partes do parser.

5.2.2 Análise Sintática

A Análise Sintática é feita pela ferramenta *BISON*, a partir de uma especificação (arquivo .y) e os *tokens* gerados pelo léxico. O arquivo .y apresenta um formato parecido com o do arquivo .l. O arquivo .y está no anexo B.

- $\%\{$ Funções auxiliares, inclusão de bibliotecas $\%\}$
- Declarações dos símbolos terminais, não terminais e dos atributos dos símbolos
- %% Lista de Regras %%

Em seguida, será apresentada a especificação sintática, no formato de GLC. Para melhor entendimento, foram omitidas as ações semânticas envolvidas e alguns símbolos terminais foram trocados pelas strings literais deles.

S ::= FUNCS MAINF
MAINF ::= main : STM
FUNCS ::= FUNC FUNCS | ϵ
FUNC ::= ID : ARGS ID ARGNAMES = STM
STM ::= VAR | LETIN | FUNCCALL | COND
ARGS ::= VARTYPE -o ARGS |VARTYPE
ARGNAMES ::= ID ARGNAMES | ϵ
VARTYPE ::= Q1++Q1 | VARTYPE ** VARTYPE
VAR ::= ID | CONSTVALUE
CONSTVALUE ::= { NUM qinl () | NUM qinr () }
FUNCCALL ::= ID (IDLIST) | ID ()
IDLIST ::= ID | ID , IDLIST
LETIN ::= let ID = STM in STM
COND ::= if STM then STM else STM | if^o STM then STM else STM

5.2.3 Análise Semântica

A Análise Semântica é feita com o ponto de partida sendo os arquivos .l e .y do FLEX e BISON a partir de ações semânticas, denotadas pelas chaves. Símbolos terminais e não terminais da gramática podem conter atributos, a estrutura de dados responsável pelo tipo desses atributos é definida no arquivo .y e é a seguinte:

- %union { std::string* strName; int varSize; Complex* complexValue; QBit* qubitValue; FQC* fqc; GenericList* genList;};

Uma union, é como uma struct do C++, mas uma variável desse tipo pode apenas ter um dos valores da union. Devido a limitações da linguagem, valores que não são de tipos primitivos devem ser representados como ponteiros. O membro strName é usado pelo símbolo ID, que representa os nomes de variáveis ou funções. O membro varSize é usado pelo símbolo VARTYPE e representa o tamanho dos tipos das variáveis. O membro complexValue é usado pelo símbolo NUM, que representa um número complexo. O membro qubitValue é usado pelo símbolo CONSTVALUE, que representa um qubit de valor constante. O membro fqc é usado pelos símbolos MAINF, STM, VAR LETIN, FUNCCALL, COND e CONDQ, que armazenam FQCs de diversas operações QML.

O membro genList é do tipo GenericList*, que é uma estrutura que possui um dado do tipo intptr_t, podendo armazenar ponteiros de qualquer tipo, e um dado do tipo GenericList*, que é um ponteiro para o próximo elemento da lista. Esse tipo é usado pelos símbolos ARGS, ARGNAMES e IDLIST, representando uma lista de IDs ou de tipos de variáveis. Não foi usada uma estrutura já implementada por bibliotecas do C++ (como vector ou list) pois dessa forma, cada símbolo que tem esse atributo possui apenas o seu elemento da lista, e o encadeamento desses elementos acontece naturalmente devido às regras de produção. O exemplo a seguir mostra um uso desse tipo.

ARGNAMES ::= ID ARGNAMES { $$ = NewElement($2,(intptr_t)$1);}
ARGNAMES ::= { $$ = NULL;}

```
GenericList* NewElement(GenericList* list, intptr_t data)
{
    GenericList* newElement = new GenericList();
    newElement->data = data;
    newElement->next = list;
    return newElement;
}
```

As ações semânticas das regras de produção anteriores constroem uma GenericList* de IDs de variáveis usando a função NewElement, que cria um elemento novo da lista e aponta para o elemento recebido como argumento.

As ações semânticas são divididas em duas classes. A primeira delas é *FunctionSemantics*, que é responsável por fazer as verificações semânticas relacionadas às funções e por implementar funções de criação/chamada de função, as funções dessa classe são as seguintes:

- static void CreateTable(std::string* name) : Cria uma função nova, com o nome passado pelo argumento, também verifica se já não existe uma função com esse nome.
- static void PopTable() : Finaliza a análise da função e adiciona ela para a lista global de funções.
- static void CreateFuncArgVars(GenericList* typeList, GenericList* nameList) : Cria as variáveis dos argumentos dentro do objeto da função sendo analisada atualmente, também verifica se existe o mesmo número de argumentos na declaração do tipo da função e de argumentos na declaração do corpo da função (caso o programador tenha declarado que a função tem 2 argumentos de entrada, mas da declaração do corpo da função colocou apenas 1 deles, por exemplo).
- static void SetFuncFQC(FQC* funcFqc): Atribui a FQC funcFQC à função sendo analisada atualmente.
- static void SetReturnType(int size): Define o tipo de retorno da função atual.
- static void VerifyFuncName(std::string* name): Verifica se o nome escrito no corpo da função é o mesmo que foi escrito na declaração da função.
- static void AddSymbol(FuncVar var): Adiciona uma variável nova à função, verifica se já não existe uma variável com esse mesmo nome antes.
- static bool VerifySymbolExists(std::string* name): Verifica se existe uma variável com o nome "name" na função.
- static void VerifyValidArgs(std::string* funcName, GenericList* varList): Verifica se as variáveis "varList" correspondem em quantidade e em tipo aos argumentos da função "funcName", verifica se essa função existe também.
- static void VerifyReturnType(FQC* ret): Verifica se a FQC de retorno da função tem o mesmo número de qubits de output que o declarado pela função atual.

A segunda classe de ações semânticas é a *SemanticOperations*, que implementa as traduções das diversas operações da linguagem (let in, if, if^o, qinl). Essas funções são as seguintes:

- static FQC* OperationFuncCall(std::string* fname,GenericList* argList): Implementa a operação de Chamada de função.
- static FQC* OperationIF(FQC* c, FQC* t, FQC* e): Traduz para FQC a operação IF clássico.
- static FQC* OperationIFQ(FQC* c, FQC* t, FQC* e): Traduz para FQC a operação IF com controle quântico.
- static bool VerifyOrtho(Unitary* t, Unitary* f): Verifica se t e f são ortogonais entre si, foi desenvolvido apenas a verificação base (verdadeiro-falso e falso-verdadeiro)
- static FQC* OperationLET(std::string* x, FQC* t, FQC* u): Traduz para FQC a operação de criação de variáveis auxiliares let in.
- static FQC* OperationCONST(QBit* constValue): Traduz para FQC uma declaração de constante.
- static FQC* OperationVAR(std::string* varname): Traduz para FQC uma chamada a uma variável.
- static void AddLetContext(std::string* x, FQC* t): Adiciona a variável x no contexto da função, para que a segunda parte do let tenha acesso a essa variável.

5.3 TRADUÇÃO DAS OPERAÇÕES DA LINGUAGEM

Como visto antes, programas QML podem ser vistos como uma sequência de FQCs do formato **FQC** $= \{a, h, b, g, \phi\}$, cada operação da linguagem é traduzida em uma FQC. As classes mais importantes do *parser* que são usadas nessa tradução são $Unitary$ e FQC.

A classe `Unitary` foi feita para implementar o tipo $Unitary$ da linguagem QML, que apresenta os seguintes construtores:

$$\begin{aligned} data\ Unitary\ \ =\ \ & \otimes[Unitary]\mid\ \oplus[Unitary]\mid Perm\ [Int] \\ & \mid Cond\ Unitary\ Unitary\mid Rotate\ (C,C)\ (C,C) \end{aligned}$$

Assim, a classe `Unitary` é constituída da seguinte maneira:

- Um enum, que indica que tipo de operação unitária esse objeto representa. As possíveis operações são: Rotation, Swap, Copy e Permutation.
- Uma union, que possui um objeto da classe da operação unitária indicada pelo enum.
- Dois ponteiros para outros objetos do tipo Unitary: nextP e nextT, que apontam para unitários do $\oplus$ e $\otimes$.
- Um Variable*, que indica se a operação é condicionada por uma variável (ou não, se for NULL).

A classe `Unitary` também possui um método `Compile()`, que gera uma *string* que é essa operação unitária traduzida em portas lógicas quânticas, esse método chama os métodos Compile de nextT e nextP, ou seja, se for chamado pela operação unitária da FQC final, vai gerar a string do arquivo de saída do parser.

A classe `FQC` possui os 4 inteiros da quíntupla descrita anteriormente e o último elemento da quíntupla ϕ, que é do tipo `Unitary*`. Além disso, essa classe possui duas listas de variáveis, uma para variáveis de entrada e outra para variáveis auxiliares.

Uma FQC que aparece em algumas operações é a FQC de compartilhamento de contexto, que recebe contextos, verifica se existem variáveis que são usadas em mais de um contexto e realiza a permutação de todas essas variáveis, colocando elas no topo da pilha de execução. Quando é detectado que existe uma variável em mais de um contexto, são atribuídos qubits novos para esses outros usos da variável, e são feitas operações CNOT dos qubits da variável original com esses novos qubits, compartilhando essas regiões da memória, fazendo com que essas variáveis novas sejam extensões da variável original.

5.3.1 Tradução da Regra da Variável

Uma referência a uma variável é traduzida da seguinte maneira pela seguinte FQC:

$$\textbf{FQC} = \{\sigma, 0, \sigma, 0, \phi = Ide(\sigma)\}$$

Onde σ é o tamanho dessa variável e Ide(n) gera "n" matrizes identidade.

5.3.2 Tradução da Regra dos construtores qinL e qinR

Os construtores $qinL$ e $qinR$ são usados na declaração de constantes de 1 qubit. No formato $\{X * qinL \mid Y * qinR\}$. A FQC de tradução dessa regra é a seguinte:

$$\textbf{FQC} = \{0, 1, 1, 0, \phi = Rot(X, Y)(Y, X)\}$$

5.3.3 Tradução da Regra da Operação Let

A operação Let (Let x = t in u) cria uma variável auxiliar x, atribui um valor t a ela e usa-as numa operação u. A tradução dessa operação é feita da seguinte maneira. Primeiro, é traduzido o circuito "t" e é armazenado ele no objeto da função atual com o nome x. Quando "x" é usado em "u", o circuito "t" é colocado no seu lugar. x é como se fosse uma declaração de função local.

5.3.4 Tradução da Regra da Operação If

A operação if retorna um de dois valores, baseado em um valor de condição. Seguindo o formato if C then T else F, onde c, T e F são FQCs. É primeiramente feita a operação de compartilhamento de contexto "c"com os três contextos dessa operação, "C", "T" e "F". Gerando uma permutação de todas essas variáveis, com os qubits de "C" ficando no topo da pilha, seguidos pelos qubits de "T" e então os qubits de "F". Após isso, é feita uma medição de "C" e então é feita uma operação condicional com "T" e "F", seguindo circuito da figura 10:

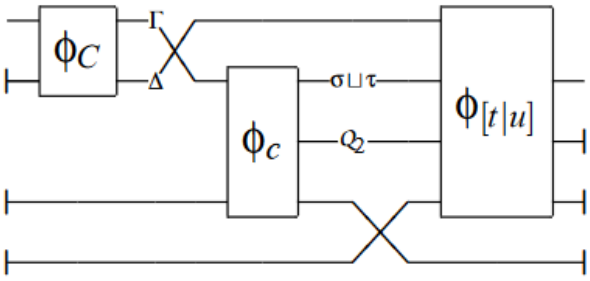


Figura 10 – Diagrama da FQC da operação IF

5.3.5 Tradução da Regra da Operação If^o

A operação If^o funciona da mesma maneira que a operação if, porém não ocorre a medição da condição. Ao invés disso, é feita uma verificação se os retornos de "T" e "F" são ortogonais. A verificação ortogonal desenvolvida para o trabalho foi uma versão simplificada, que só é aceita se os lados tiverem valores Verdadeiro e Falso ou Falso e Verdadeiro, que é o caso base dessa operação.

5.4 GERAÇÃO DE CIRCUITOS

O processo de geração de circuitos inicia quando a FQC da função main é finalizada. Primeiramente a pilha de qubits é criada, seguido pela atribuição dos qubits dessa pilha às variáveis dessa FQC, seguido pela chamada do o método Compile da operação unitária dessa FQC, para então colocar essa string do arquivo de saída do parser.

As operações de rotação são traduzidas em uma portá lógica U3. A porta U3 tem uma matriz do seguinte formato:

$$\text{U3}(\theta, \phi, \lambda) = \begin{pmatrix} cos(\theta/2) & -sin(\theta/2) * e^{i*\lambda} \\ sin(\theta/2) * e^{i*\phi} & cos(\theta/2) * e^{i*(\lambda+\phi)} \end{pmatrix}$$

E as rotações podem ser interpretadas como

$$\text{Rot} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Então, a conversão de rotação para porta U3 funciona seguindo as seguintes formulas:

- $\theta = 2 * arccos(a)$,
- $\phi = -i * ln(\frac{c}{sin(arccos(a))})$,
- $\lambda = -i * ln(\frac{-b}{sin(arccos(a))})$.

A operação swapN, que possui os parâmetros offset,length,size, são traduzidas em operações de swap, seguindo o seguinte pseudocódigo:

```
for(int i = 0; i < length; i++)
{
    if(offset+i != i) //Evita Swap com si mesmo
            GeraSwap(offset+i,i);
}
```

Internamente, o parser também deve ajustar a sua pilha de execução, para que as variáveis estejam nas posições corretas de acordo com o programa, então são feitas as seguintes operações com essa pilha:

- Desempilha "offset" elementos da pilha de execução e os empilha na pilha auxiliar A

- Desempilha "length" elementos da pilha de execução e os empilha da pilha auxiliar B
- Desempilha toda a pilha auxiliar A e empilha esses elementos na pilha de execução.
- Desempilha toda a pilha auxiliar B e empilha esses elementos na pilha de execução.

Isso é feito internamente no parser, o que é realmente gerado no arquivo intermediário é uma sequência de portas swap que tem efeito equivalente a essa operação.

As operações Permutation são traduzidas operações de swap, mas o algoritmo de geração é diferente do da swapN. Nessa tradução, o conteúdo da pilha é passado todo para um vetor, e então, as variáveis apontadas pela operação de permutação são colocadas nas posições iniciais do vetor, e então o conteúdo desse vetor é colocado de volta na pilha de execução.

Para todas as operações, a variável de condição é verificada antes de fazer a tradução, caso ela for diferente de nulo, é adicionado o controle na operação. A porta U3 vira a porta CU3, a porta SWAP é convertida numa porta CSWAP, etc.

5.5 CONSTRUÇÃO E EXECUÇÃO DE CIRCUITOS USANDO QISKIT

O Montador de circuitos é um programa em Python que tem o papel de ler o arquivo gerado pelo parser, montar o circuito, usando o pacote Qiskit, e enviá-lo à execução nos computadores quânticos da IBM. O código desse programa está no Anexo I.

A função *ReadParsedCircuit* recebe o nome do arquivo de entrada, lê o *header* desse arquivo e constrói o circuito Qiskit, usando outra das funções declaradas, *AddOperationToCircuit*. Essa outra função recebe a string da linha atual do arquivo, o circuito atual, os registradores quânticos e clássicos do circuito; faz um parsing da string para descobrir qual operação a linha indica e quais são os argumentos dessa operação e em seguida, adiciona a operação ao circuito atual.

Para a comunicação com o IBMQ, primeiro é carregada a conta da IBMQ (o token de entrada foi omitido, pois é pessoal de cada conta do site do IBMQ), cria o circuito a partir do arquivo "circuito.txt" e a função *ReadParsedCircuit* (definida antes), define o número de execuções do circuito, executa o circuito em um simulador quântico, mostra os resultados e então executa o circuito no computador quântico "ibmq_belem", mostrando os resultados também.

5.6 EXEMPLOS DE USO

Nessa seção, serão apresentados exemplos de programas QML, os circuitos que eles geram e os resultados da execução deles em computadores quânticos.

5.6.1 Exemplo 1: Constante

Um programa básico que só cria uma constante metade verdadeiro e metade falso.

```
main: { 0.5 qinl () | 0.5 qinr ()}
```

Arquivo gerado pelo parser:

```
1
U3 0 1.570796 -0.000000 3.141593
Measure 0
```

Resultado da operação no computador do IBMQ (Figura 11).

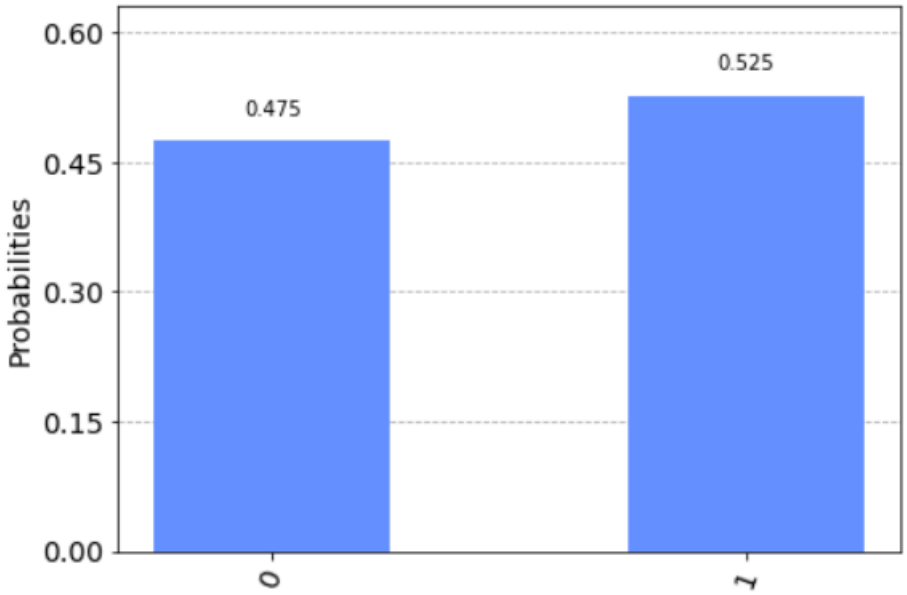


Figura 11 – Resultado do Exemplo 1

Como é possível ver na Figura 11, o resultado chegou bem próximo do 0.5 Falso 0.5 Verdadeiro da constante. Isso acontece pois a porta U3 apresenta um grau maior de ruído que outras portas de um qubit, gerando resultados um pouco menos precisos quando usadas em computadores quânticos, esse comportamento não acontece em alguns simuladores quânticos que não simulam ruído.

5.6.2 Exemplo 2: Let,If e Funções

Um programa que declara três funções e usa elas na main com as operações *let* e *if* 1. Três qubits são necessários para esse programa, um para o x, um para o resultado do *then* e outro para o resultado do *else*. O que deve acontecer nesse programa é que, como x é verdadeiro, então o resultado é false. Ou seja, o primeiro qubit, x, fica verdadeiro sempre, e os outros dois qubits ficam falsos.

```
true :Q1++Q1
true = { 0 qinl () | 1 qinr () }

false: Q1++Q1
false = { 1 qinl () | 0 qinr () }

halftrue: Q1++Q1
halftrue = { 0.5 qinl () | 0.5 qinr() }

main:
let x = true() in
if x then false() else halftrue()
```

Arquivo gerado pelo parser: (Nota-se a medição do qubit 0 para poder fazer o if)

```
3
U3 0 3.141593 0.000000 3.141593
Measure 0
NOT 0
CU3 0 2 1.570796 -0.000000 3.141593
NOT 0
Measure 0
Measure 1
Measure 2
```

Resultado da operação no computador do IBMQ (Figura 12).

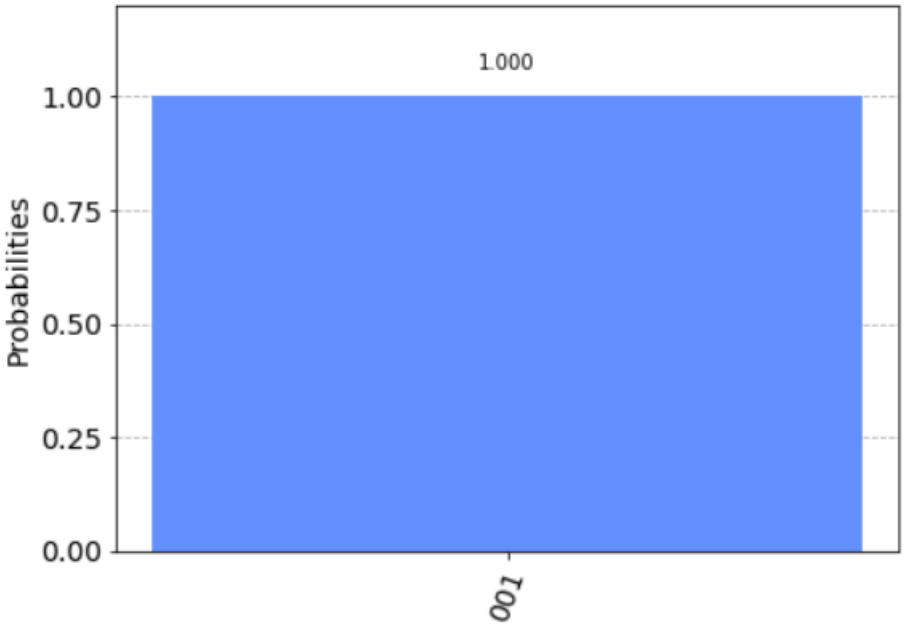


Figura 12 – Resultado do Exemplo 2.

5.6.3 Exemplo 3: Let e Ifo

O programa abaixo usa o Ifo, porém os resultados do then e else não são ortogonais, então o parser retorna o seguinte erro:

```
halftrue :Q1++Q1
halftrue = { 0.5 qinl () | 0.5 qinr () }

main:
let x = halftrue () in
ifo x then { 1 qinl () | 0 qinr () }
else { 1 qinl () | 0 qinr () }
```

```
-------------------------------------------------------
IFq returns not orthogonal
SYN ERROR - Non orthogonal ifq returns on line 6
```

Aqui tem o programa corrigido, ele deve resultar em 0.5 verdadeiro e 0.5 falso.

```
halftrue :Q1++Q1
```

```
halftrue = { 0.5 qinl () | 0.5 qinr () }

main:
let x = halftrue () in
ifo x then { 0 qinl () | 1 qinr () }
else { 1 qinl () | 0 qinr () }
```

Arquivo gerado pelo parser (Preste atenção na falta de Medição do Qubit 0)

```
3
U3 0 1.570796 -0.000000 3.141593
CU3 0 1 3.141593 0.000000 3.141593
NOT 0
NOT 0
Measure 0
Measure 1
Measure 2
```

O resultado mostra que metade das vezes, o qubit 0 é verdadeiro, e então o qubit 1 é aplicado a CU3, e metade das vezes o qubit 0 é falso, resultado na não aplicação da CU3 no qubit 1.

5.6.4 Exemplo 4: Swaps Excessivos

Esse programa foi descrito para demonstrar um problema com o compilador desenvolvido. Usar as operações de Permutação e SwapN causam SWAPS desnecessários nos circuitos, piorando o desempenho do circuito e tendo mais erros de decoerência.

```
true: Q1++Q1
true = { 0 qinl () | 1 qinr () }

false:Q1++Q1
false = { 1 qinl () | 0 qinr () }
```

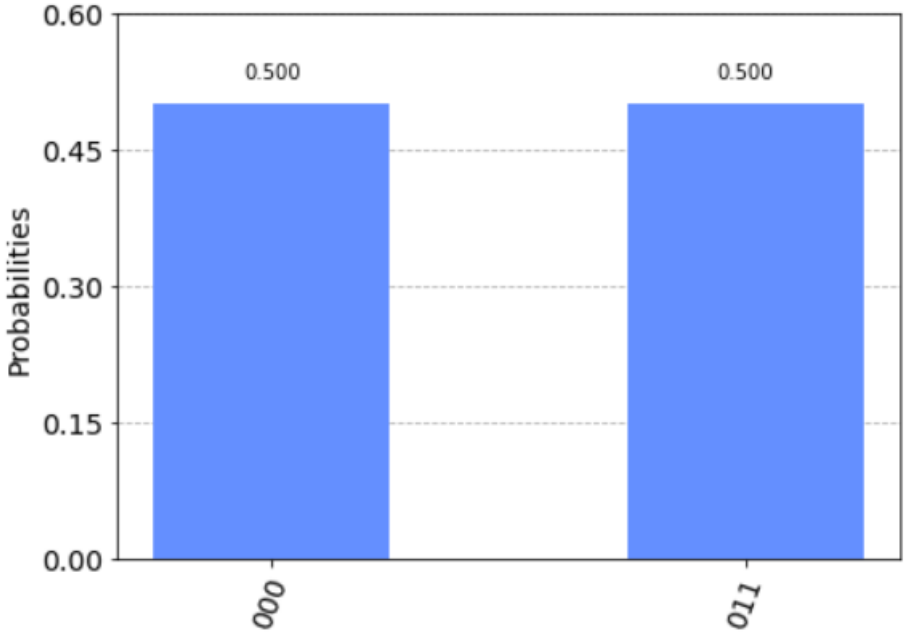


Figura 13 – Resultado do Exemplo 3.

```
halftrue:Q1++Q1
halftrue= { 0.5 qinl () | 0.5 qinr () }

func:Q1++Q1 -o Q1++Q1
func z = if z then true() else false()

main:
let x = halftrue () in
let y = func(x) in
if y then false() else true()
```

Arquivo gerado pelo parser, demonstrando os SWAPs que podem ser otimizados.

```
5
SWAP 1 4
SWAP 2 3
SWAP 3 2
SWAP 4 1
SWAP 1 4
```

```
SWAP 2 3
Measure 0
CU3 0 1 3.141593 0.000000 3.141593
NOT 0
NOT 0
Measure 0
NOT 0
CU3 0 2 3.141593 0.000000 3.141593
NOT 0
Measure 0
Measure 1
Measure 2
Measure 3
Measure 4
```

O circuito gerado pelo código do exemplo 4 apresenta muitos SWAPs, que são muito ruidosos para o circuito, com tantos SWAPs, os resultados desse circuito seriam quase aleatórios.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

No presente trabalho foi apresentada a implementação detalhada de um compilador para a linguagem funcional quântica QML, seguindo o paradigma de controle quântico, que traduz as construções da linguagem para circuitos quânticos, que posteriormente são executados no computador quântico da IBM, o IBMQ. Utilizou-se as ferramentas FLEX e BISON para a construção do *parser*. O compilador para circuitos foi implementado em `C++` e o tradutor para a biblioteca Qiskit, que executa no IBMQ foi implemetado em Python. O trabalho desenvolvido consegue corretamente criar e executar circuitos de acordo com uma especificação de um programa escrito em QML. Até o presente momento, o *parser* possibilita apenas a criação de variáveis de tamanho 1. A implementação das traduções das operações usa uma pilha para guardar essas variáveis, usando operações de *swap* e permutação para que as variáveis corretas fiquem no topo, ambas as operações precisam ser traduzidas em portas $SWAPs$, que são bem pesadas computacionalmente e causam maiores taxas de erros nos qubits, fazendo com que os circuitos gerados pelo parser sejam menos eficientes que circuitos feitos manualmente.

Como trabalhos futuros, planeja-se completar o desenvolvimento do parser e interpretador para tipos de mais de um qubit e também desenvolver uma foma nova de guardar os qubits (que não seja a pilha). Caso seja decidido trabalhar com outro ambiente de execução, o programa montador deve ser ajustado para montar os circuitos para esse novo ambiente.

REFERÊNCIAS

AHO, A. et al. Compilers Principles, Techniques, Tools. , [S.l.], p.1009, 01 2007.

ALTENKIRCH, T.; GRATTAGE, J. A functional quantum programming language. In: ANNUAL IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 20. **Anais...** [S.l.: s.n.], 2005.

ANIS, M. S. et al. **Qiskit**: an open-source framework for quantum computing. 2021.

BISON website. acessado em 17/02/2022, https://www.gnu.org/software/bison/manual/bison.html.

DEUTSCH, D.; JOZSAT, R. Rapid solution of problems by quantum computation. In: IN PROC. **Anais...** [S.l.: s.n.], 1992. p.553–558.

FEYNMAN, R. P. Simulating Physics with Computers. **International Journal of Theoretical Physics**, [S.l.], v.21, n.6, p.467–488, 1982.

FLEX website. acessado em 17/02/2022, https://github.com/westes/flex.

GROVER, L. K. A fast quantum mechanical algorithm for database search. In: STOC-1996. **Proceedings...** ACM, 1996. p.212–219.

HASKELL language website. acessado em 31/1/2022, https://www.haskell.org/.

IBM Quantum website. acessado em 24/09/2021, https://quantum-computing.ibm.com/.

NIELSEN, M. A.; CHUANG, I. L. **Quantum Computation and Quantum Information**: 10th anniversary edition. 10th.ed. USA: Cambridge University Press, 2011.

PRESKILL, J. **Quantum computing and the entanglement frontier**. 2012.

SHOR, P. W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. **SIAM Journal on Computing**, [S.l.], v.26, n.5, p.1484–1509, 1997.

ANEXOS

ANEXO A – Arquivo lexical.l

```
%option noyywrap
%option yylineno
%{
#include <stdlib.h>
#include <string>
#include "y.tab.h"
void yyerror(char *) { printf("Lex Error on Line %d\n",yylineno); }
Complex ParseComplex(bool plus)
{
    std::string str = std::string(yytext);
    int impos = str.find(plus?"+":"-",1); /*Finds second minus position*/
    float real = std::stof(str.substr(0,impos));
    float imag = std::stof(str.substr(impos,str.length()-impos-1));
    printf("Read Complex (%f,%f)\n",real,imag);
    return Complex(real,imag);
}
%}
%option yylineno
DIGIT [0-9]
LETTER [a-z]|[A-Z]
%%
 /* KEYWORDS */
"main" {return MAIN;}
"if" {return IF;}
"ifo" {return IFQ;}
"else" {return ELSE;}
"then" {return THEN;}
"qinl" { return INL; }
"qinr" { return INR; }
"let" {return LET;}
"in" {return IN;}

 /* TYPES AND OPERATIONS */
"Q1" {return Q1;}
"=" {return EQU;}
"++" {return PLUS;}
"**" {return TENP;}
 /* Other */
"-o" {return RARROW;}
":" {return COL;}
"(" {return OPPAR;}
")" {return CLPAR;}
"{" {return OPBRK;}
"}" {return CLBRK;}
"," {return COMMA;}
"|" {return VBAR;}
 /* Numbers */
 /* Real Number + Negative Imaginary Number */
("-"|"+")?{DIGIT}+("."{DIGIT}+)?"-"{DIGIT}+("."{DIGIT}+)?"i"
{ yylval.complexValue = new Complex(ParseComplex(false)); return NUM;}

/* Real Number + Positive Imaginary Number */
("-"|"+")?{DIGIT}+("."{DIGIT}+)?"+"{DIGIT}+("."{DIGIT}+)?"i"
{ yylval.complexValue = new Complex(ParseComplex(true));  return NUM;}
```

```
/*Real Number*/
("-"|"+")?{DIGIT}+("."{DIGIT}+)?
{yylval.complexValue = new Complex(std::stof(yytext),0); return NUM; }

{LETTER}({LETTER}|{DIGIT})*
{yylval.strName = new std::string(yytext);return ID;}

[\t\n]+        {;}       /* ignore whitespace */

%%
```

ANEXO B – Arquivo sintatical.y

```
%code requires {
    #include "Function.h"
    #include "SemanticActions.h"
}
%{
#include "Function.h"
#include "SemanticActions.h"
#include <iostream>
#include <fstream>
extern FILE *yyin;
extern int yylineno;
std::stack< Function* > functionStack;
std::list< Function* > allFunctions;
int globalConstID = 0;
int yylex();
void yyerror(const char *s);

GenericList* NewElement(GenericList* list, intptr_t data)
{
    GenericList* newElement = new GenericList();
    newElement->data = data;
    newElement->next = list;
    return newElement;
}

std::string AddMeasures(std::string str,int nqubits)
{
    std::string finalstr;
    finalstr.append(str);
    for(int i = 0; i < nqubits; i++)
    {
         char buffer[25];
         sprintf(buffer,"Measure %d\n",i);
         finalstr.append(buffer);
    }
    return finalstr;
}
void AttributeQubits(std::stack<QBit*> qubitStack,FQC* mainFQC)
{
     std::stack<QBit*> auxStack;
     printf("ATTRIBUTING %d qubits, %d input and %d aux\n",
     mainFQC->a+mainFQC->h,mainFQC->a,mainFQC->h);
     for(auto var : mainFQC->inputVars)
     {
         var->value = qubitStack.top();
         qubitStack.pop();
         printf("%s[0] = %d\n",var->name.c_str(),auxStack.size());
         auxStack.push(var->value);
         // Accounts for variable sizes
         for(int i = 1; i < var->size; i++)
         {
             auxStack.push(qubitStack.top());
             qubitStack.pop();
         }
     }
```

```
    for(auto var : mainFQC->auxVars)
    {
        var->value = qubitStack.top();
        qubitStack.pop();
        printf("%s[0] = %d\n",var->name.c_str(),auxStack.size());
        auxStack.push(var->value);
        for(int i = 1; i < var->size; i++) // Accounts for variable sizes
        {
            auxStack.push(qubitStack.top());
            qubitStack.pop();
        }
    }
    //Restores Original Stack
    while(!auxStack.empty())
    {
        qubitStack.push(auxStack.top());
        auxStack.pop();
    }
}
void CompileMain(FQC* mainFQC)
{
    //Creates Output file
    std::ofstream  outputfile;
    outputfile.open("circuit.txt");
    if(!outputfile) {
       printf(" ERROR OPENING circuit.txt\n");
    }

    //Inputs Header
    int totalSize =  mainFQC->a + mainFQC->h;
    outputfile << (totalSize) << std::endl;

    //Creates Qubit stack
    std::stack<QBit*> qubitStack;
    for(int i = 0; i < totalSize; i++)
    {
        qubitStack.push(new QBit());
    }
    //printf("MAIN FQC =\n");
    //mainFQC->printFQC();
    AttributeQubits(qubitStack,mainFQC);

    std::string fqcStr = mainFQC->phi->Compile(0,&qubitStack);
    printf("CIRCUIT =\n%s",fqcStr.c_str());
    //Add measures for all qubits
    fqcStr = AddMeasures(fqcStr,totalSize);
    outputfile << fqcStr;
    outputfile.close();

    printf("FINAL VAR IDXs\n");
    for(auto var : mainFQC->inputVars)
        printf("%s ends on [%d]\n",
        var->name.c_str(),var->GetStackIdx(&qubitStack));
    for(auto var : mainFQC->auxVars)
        printf("%s ends on [%d]\n",
        var->name.c_str(),var->GetStackIdx(&qubitStack));

}
```

```
%}
%locations
%union {
    GenericList* genList;
    std::string* strName;
    int varSize;
    FQC* fqc;
    Complex* complexValue;
    QBit* qbitValue;
};

%token MAIN IF IFQ ELSE THEN INL INR LET IN EQU INC DEC
%token OPPAR CLPAR OPBRK CLBRK COMMA VBAR COL RARROW
%token PLUS TENP
%token TRUE FALSE Q1
%token <strName> ID
%token <complexValue> NUM
%type <qbitValue> CONSTVALUE
%type <genList> ARGS ARGNAMES IDLIST
%type <varSize> VARTYPE
%type <fqc> MAINF STM VAR LETIN FUNCCALL COND
// Operator precedence and associativity:
%right '='
%%
// RULES SECTION:

S : FUNCS MAINF{ FunctionSemantics::PopTable(); CompileMain($2); exit(0); }

MAINF: MAIN COL {FunctionSemantics::CreateTable(new std::string("MAIN"));}
       STM { $$ = $4; }
FUNCS: FUNC FUNCS  |
FUNC: ID COL {FunctionSemantics::CreateTable($1);}
      ARGS ID {FunctionSemantics::VerifyFuncName($5);}
      ARGNAMES {FunctionSemantics::CreateFuncArgVars($4,$7);}
      EQU STM
      {
          FunctionSemantics::VerifyReturnType($10);
          FunctionSemantics::SetFuncFQC($10);
          FunctionSemantics::PopTable();
          ("FUNC FQC");($10)->printFQC();
      }

ARGS: VARTYPE RARROW ARGS { $$ = NewElement($3,(intptr_t)$1); } |
      VARTYPE { FunctionSemantics::SetReturnType($1); $$ = NULL; }
VARTYPE: Q1 PLUS Q1 { $$ = 1; } | VARTYPE TENP VARTYPE { $$ = $1 + $3; }
ARGNAMES: ID ARGNAMES { $$ = NewElement($2,(intptr_t)$1);} | {$$ = NULL;}
STM: VAR { $$ = $1; } |
     LETIN { $$ = $1; } |
     FUNCCALL { $$ = $1; } |
     COND { $$ = $1; }

VAR: ID { $$ = SemanticOperations::OperationVAR($1); } |
     CONSTVALUE { $$ = SemanticOperations::OperationCONST($1); }

CONSTVALUE: OPBRK NUM INL OPPAR CLPAR VBAR NUM INR OPPAR CLPAR CLBRK
            { $$ = new QBit(*$2,*$7);}
```

```
LETIN: LET ID EQU STM  {SemanticOperations::AddLetContext($2,$4);}
       IN STM {
           $$ = SemanticOperations::OperationLET($2,$4,$7);
           FunctionSemantics::RemoveLetContext();
       }

FUNCCALL:ID OPPAR IDLIST CLPAR {
         FunctionSemantics::VerifyValidArgs($1,$3);
         $$ = SemanticOperations::OperationFuncCall($1,$3);
        }
FUNCCALL:ID OPPAR CLPAR {
         FunctionSemantics::VerifyValidArgs($1,NULL);
         $$ = SemanticOperations::OperationFuncCall($1,NULL);
        }

IDLIST: ID  {
            FunctionSemantics::VerifySymbolExists($1);
            $$ = NewElement(NULL,(intptr_t)$1);
        } |

          ID COMMA IDLIST {
           FunctionSemantics::VerifySymbolExists($1);
           $$ = NewElement($3,(intptr_t)$1);
          }

COND: IF STM THEN STM ELSE STM { $$ = SemanticOperations::OperationIF($2,$4,$6);} |
    IFQ STM THEN STM ELSE STM { $$ = SemanticOperations::OperationIFQ($2,$4,$6);}

;

%%

int main() {
     yyin = fopen("input.txt","r");
     yyparse();
     return yylex();
}

void yyerror(const char *s){ printf("\n SYN ERROR - %s on line %d\n",s,yylineno); }
int yywrap(){ return 1; }
```

ANEXO C – Arquivo SemanticActions.h

```
#ifndef SEMANTIC
#define SEMANTIC
#include <iostream>
#include "Function.h"
extern std::stack<QBit> globalVarStack;
extern std::stack< Function* > functionStack;
extern std::list< Function* > allFunctions;
extern int globalConstID;
extern FILE *yyin;
extern int yylineno;
int yylex();
void yyerror(const char *s);

class FunctionSemantics
{
     public:
     //Function Semantic Actions
     static void CreateTable(std::string* name)
     {
         for(auto& fi : allFunctions)
         {
             if(fi->name == (*name))
             {
                 printf("Duplicate definition of function %s\n",name->c_str());
                 yyerror("Duplicate function\n");
                 exit(0);
             }
         }

         functionStack.push(new Function(*name));
         printf("Created Function %s\n",name->c_str());
     }
     static void PopTable()
     {
         Function* f = functionStack.top();
         //printf("Popped function\n");
         allFunctions.push_back(f);
         printf("Added Function %s to list\n",f->name.c_str());
         functionStack.pop();
     }
     static void CreateFuncArgVars(GenericList* typeList, GenericList* nameList)
     {
         Function* f = functionStack.top();
         while(typeList != NULL && nameList != NULL)
         {
             AddSymbol(
             FuncVar(*((std::string*)nameList->data),(int)typeList->data)
             );
             f->symbolTable->nargs++;
             typeList = typeList->next;
             nameList = nameList->next;
         }
         if(!(typeList == NULL && nameList == NULL))
         {
             printf("function %s has too many/too few arguments \n",f->name.c_str());
```

```
        yyerror("No matching argument type");
        exit(0);
    }
}

static void SetFuncFQC(FQC* funcFqc)
{
    Function* f = functionStack.top();
    f->funcFQC = funcFqc;
    f->funcFQC->g = funcFqc->a + funcFqc->h - funcFqc->b;
}
static void SetReturnType(int size)
{
    Function* f = functionStack.top();
    f->returnType = size;
    //printf("Func %s returns type of size %d\n",f->name.c_str(),size);
}

static void VerifyFuncName(std::string* name)
{
    Function* f = functionStack.top();
    if(!(f->name == (*name)))
    {
        printf("Expected implementation of function %s\n",f->name.c_str());
        yyerror("Wrong Function Name\n");
        exit(0);
    }
}

static void AddSymbol(FuncVar var)
{
    SymbolTable* st = functionStack.top()->symbolTable;
    if(!st->AddSymbol(var))
    {
        printf("Duplicate FuncVar %s in function %s\n",
        var.name.c_str(),functionStack.top()->name.c_str());
        yyerror("Duplicate symbol");
        exit(0);
    }
    //printf("Added Symbol %s of type size %d\n",var.name.c_str(),var.size);

}
static bool VerifySymbolExists(std::string* name)
{
    if(!functionStack.top()->symbolTable->contains(*name))
    {
        printf("Unknown var %s \n",name->c_str());
        yyerror("Unknown var");
        exit(0);
        return false;
    }
    return true;
}
static void AddLetVar(std::string name,FQC* fqc)
{
    SymbolTable* st = functionStack.top()->symbolTable;
    // *fqc because it stores only a copy of the FQC, not a reference to it
    st->letvars.push_back(LetVar(name,*fqc));
```

```
}
static void RemoveLetContext()
{
    SymbolTable* st = functionStack.top()->symbolTable;
    st->RemoveLetVar();
}

static void VerifyValidArgs(std::string* funcName, GenericList* varList)
{
    Function* currentFunc = functionStack.top();
    Function* func = NULL;
    //Verify if func exists
    for(auto& f : allFunctions)
    {
        if(f->name == *funcName)
        {
            func = f;
        }
    }
    if(func == NULL)
    {
        printf("Unknown function %s\n",funcName->c_str());
        yyerror("Unknown function");
        exit(0);
    }
    int n = 0;
    //Verify correct number of args
    GenericList* genericList = varList;
    while(genericList != NULL) {
        n++;
        printf("Var %s\n",((std::string*)genericList->data)->c_str());
        genericList = genericList->next;
    }
    if(n != func->symbolTable->nargs)
    {
        printf("Incorrect number of arguments (%d) on function %s call
        (has %d args)\n",n,func->name.c_str(),func->symbolTable->nargs);
        yyerror("Incorrent number of arguments");
        exit(0);
    }

    //Verify Arg types
    n = 0;
    while(varList != NULL)
    {
        std::string varName = *((std::string*)varList->data );
        FuncVar* var = (currentFunc->symbolTable->GetSymbol(varName));
        if(var != NULL)
        {
            if(func->symbolTable->vars.at(n).size != var->size)
            {
                printf("Incorrect argument size (%d) on argument %d
                of function %s\n",var->size,n,func->name.c_str());
                yyerror("Wrong argument size");
            }
        }
        else
        {
```

```
                    LetVar* lvar = currentFunc->symbolTable->GetLetVar(varName);
                    if(func->symbolTable->vars.at(n).size != lvar->varFQC.b)
                    {
                        printf("Incorrect argument size (%d) on argument %d
                        of function %s\n",lvar->varFQC.b,n,func->name.c_str());
                        yyerror("Wrong argument size");
                    }
                }
                n++;
                varList = varList->next;
            }
        }

        static void VerifyReturnType(FQC* ret)
        {
            Function* f = functionStack.top();
            if(ret->b != f->returnType)
            {
                printf("Function %s return FuncVar of size %s does not match
                its return type %s\n",f->name.c_str(),ret->b,f->returnType);
                yyerror("Function return type does not match declaration");
            }
        }
};
class SemanticOperations
{
        public:
        static FQC* OperationFuncCall(std::string* fname,GenericList* argList)
        {
            Function* fatherFunc = functionStack.top();
            Function* f = NULL;
            for(auto& func : allFunctions)
            {
                if(func->name == (*fname))
                {
                    f = func;
                }
            }
            if(f == NULL)
            {
                yyerror("Unkown function name");
                return NULL;
            }

            int i = 0;
            FQC* funcFQCCopy = new FQC(
            f->funcFQC->a,f->funcFQC->h,f->funcFQC->b,f->funcFQC->g,f->funcFQC->phi );

            funcFQCCopy->inputVars = f->funcFQC->inputVars;
            funcFQCCopy->auxVars = f->funcFQC->auxVars;
            while(argList != NULL)
            {
                std::string argName = *((std::string*)argList->data);
                // Adds Function Prefix
                std::string newVarName = fatherFunc->name; newVarName.append(argName);

                FuncVar correspondingVar = f->symbolTable->vars.at(i);
                std::string newCorrVarName = f->name;
```

```
            newCorrVarName.append(correspondingVar.name);

            Variable* fqcVar = f->funcFQC->GetVar(newCorrVarName);
            fqcVar->aliases.push_back(newVarName); //Add alias

            argList = argList->next;
            i++;
        }
        return funcFQCCopy;
    }

    static FQC* OperationIF(FQC* c, FQC* t, FQC* f)
    {
        //Check for output type of c
        if(c->b != 1)
        {
            printf("Size of conditional operator different than 1 (is %d) \n",c->b);
            yyerror("Invalid condition size");
            exit(0);
        }
        //Check for size difference in output from t and e
        if(t->b != f->b)
        {
            printf("Difference in size of \"then\" and \"else\" operators
            (%d and %d) \n",t->b,f->b);
            yyerror("If size mismatch");
            exit(0);
        }
        Function* func = functionStack.top();
        // Con =  context for operations on then and else
        FQC* Con = MakeContextFQC(MakeContextFQC(c,t),f);

        FQC* ifFQC = new FQC(Con->a, Con->h ,t->b);
        c->CreateCondition(t,f,true);
        ifFQC->phi = Con->phi;
        ifFQC->phi->PlusUnitary(c->phi);
        ifFQC->inputVars = c->inputVars;
        ifFQC->auxVars = c->auxVars;

        ifFQC->printFQC();
        return ifFQC;
    }
    static FQC* OperationIFQ(FQC* c, FQC* t, FQC* f)
    {
        //Check for output type of c
        if(c->b != 1)
        {
            printf("Size of conditional operator different than 1 (is %d) \n",c->b);
            yyerror("Invalid condition size");
            exit(0);
        }
        //Check for size difference in output from t and e
        if(t->b != f->b)
        {
            printf("Difference in size of \"then\" and \"else\" operators
            (%d and %d) \n",t->b,f->b);
            yyerror("Ifq size mismatch");
            exit(0);
```

```
    }

    Function* func = functionStack.top();
    //Apply ortho verification
    if(!VerifyOrtho(t->phi,f->phi))
    {
        printf("IFq returns not orthogonal\n");
        yyerror("Non orthogonal ifq returns");
        exit(0);
    }
    // Con =  context for operations on then and else
    FQC* Con = MakeContextFQC(MakeContextFQC(c,t),f);
    FQC* ifqFQC = new FQC(Con->a, Con->h,t->b);

    c->CreateCondition(t,f,false);

    ifqFQC->phi = c->phi;
    ifqFQC->inputVars = c->inputVars;
    ifqFQC->auxVars = c->auxVars;

    return ifqFQC;
}
static bool VerifyOrtho(Unitary* t, Unitary* f)
{
    Rotation inl = Rotation(); inl.InitNot();
    Rotation inr = Rotation(); inr.InitIdentity();
    if(t->nextP == NULL && t->nextT == NULL
    && f->nextP == NULL && f->nextT == NULL)
    {
        if(t->type == Unitary::UType::Rot && f->type == Unitary::UType::Rot)
        {
            if(*(t->rot)== inl && *(f->rot)== inr)
                return true;
            if(*(f->rot)== inl && *(t->rot)== inr)
                return true;
        }
    }
    return false;
}
static FQC* OperationLET(std::string* x, FQC* t, FQC* u)
{
    return u;
}
static FQC* OperationCONST(QBit* constValue)
{
    constValue->Normalize();
    FQC* constFQC = new FQC(0,1,1,0,new ConstRot(constValue->left,constValue->rig
    std::string constName = std::string("CONST");
    constName.append(std::to_string(globalConstID));
    globalConstID++;
    constFQC->AddAuxVar(new Variable(constName,1));
    return constFQC;
}
static FQC* OperationVAR(std::string* varname)
{
    Function* f = functionStack.top();
    // Gets FuncVar in context
    FuncVar* v = f->symbolTable->GetSymbol(*varname);
```

```
    // If var does not exist, it might be a let var
    if(v == NULL)
    {
        LetVar* lv = f->symbolTable->GetLetVar(*varname);
        if(lv != NULL)
        {
            FQC* varFQC = new FQC(
            lv->varFQC.a,lv->varFQC.h,lv->varFQC.b,lv->varFQC.g);
            varFQC->inputVars = lv->varFQC.inputVars;
            varFQC->auxVars = lv->varFQC.auxVars;

            varFQC->phi = new Unitary();
            varFQC->phi->type = lv->varFQC.phi->type;
            varFQC->phi->rot = lv->varFQC.phi->rot;
            varFQC->phi->nextT = lv->varFQC.phi->nextT;
            varFQC->phi->nextP = lv->varFQC.phi->nextP;
            varFQC->phi->condition = lv->varFQC.phi->condition;
            return varFQC;
    }
    else
    {
        printf("Unknown var %s in function %s\n",
        varname->c_str(),f->name.c_str());
        yyerror("Unknown ID\n");
        exit(0);
    }
    }
    int size = v->size;
    IdentityRot* identity = new IdentityRot(size);

    FQC* varFQC = new FQC(size,0,size,0,identity);
    // Adds Function Prefix
    std::string newVarName = f->name;
    newVarName.append(v->name);
    varFQC->AddInputVar(new Variable(newVarName,v->size));
    printf("Added var %s to context\n",varname->c_str());
    return varFQC; //Returns that FQC
}
//let x = t in u
static void AddLetContext(std::string* x, FQC* t)
{
    Function* f = functionStack.top();
    if(t == NULL)
    {
        yyerror("Let Size Mismatch");
        exit(0);
    }
    //Makes x and alias of t's output
    if(t->inputVars.size() == 1)
    {
        t->inputVars.at(0)->aliases.push_back(*x);
    }
    else if (t->auxVars.size() == 1)
    {
        t->auxVars.at(0)->aliases.push_back(*x);
    }
    FunctionSemantics::AddLetVar(*x,t);
```

```
}
static FQC* MakeContextFQC(FQC* gamma, FQC* delta)
{
    std::vector<Variable*> gammaVars = gamma->inputVars;
    std::vector<Variable*> deltaVars = delta->inputVars;


    FQC* contextFQC = new FQC(0,0,0,0);
    std::vector<Variable*> permutationVars;
    std::vector<Variable*> copyVars;
    //Add all gamma vars
    for(auto gVar : gammaVars){
        contextFQC->AddInputVar(gVar);
        permutationVars.push_back(gVar);
    }

    //Check for conflicts
    Unitary* copies = NULL;
    for(auto dVar : deltaVars)
    {
        bool conflict = false;
        std::string copyName = std::string("Copy of ");
        for(auto pVar : permutationVars)
        {
            if((*dVar) == (*pVar))
            {
                copyName.append(pVar->name);
                delta->RenameVar(dVar->name,copyName);
                copyVars.push_back(dVar);
                contextFQC->AddAuxVar(dVar);
                if(copies == NULL)
                {
                    copies = new CopyOp(pVar,dVar);
                }
                else
                {
                    copies->PlusUnitary(new CopyOp(pVar,dVar));
                }
                conflict = true;
                break;
            }
        }
        if(!conflict ) contextFQC->AddInputVar(dVar);
        permutationVars.push_back(dVar);

    }
    //This vector exists to add aux vars to the permutation,
    //since they dont need duplicate check
    std::vector<Variable*> varsToPermutate;
    //Gamma vars
    for(int i = 0; i < gamma->inputVars.size();i++)
        varsToPermutate.push_back(permutationVars.at(i));
    for(int i = 0; i < gamma->auxVars.size();i++)
    {
        varsToPermutate.push_back(gamma->auxVars.at(i));
        contextFQC->AddAuxVar(gamma->auxVars.at(i));
    }
    int gammaSize = gamma->inputVars.size();
```

```
        //Delta vars
        for(int i = 0; i < delta->inputVars.size();i++)
        varsToPermutate.push_back(permutationVars.at(i+gammaSize));
        for(int i = 0; i < delta->auxVars.size();i++)
        {
            varsToPermutate.push_back(delta->auxVars.at(i));
            contextFQC->AddAuxVar(delta->auxVars.at(i));
        }

        //Gera FQC do contexto
        int naux = varsToPermutate.size() - permutationVars.size();
        contextFQC->a = permutationVars.size() - copyVars.size();
        contextFQC->b = varsToPermutate.size();
        contextFQC->h = naux+copyVars.size();
        contextFQC->phi = new PermOp(varsToPermutate);
        contextFQC->phi->PlusUnitary(copies);

        return contextFQC;
    }
};
#endif
```

ANEXO D – Arquivo Function.h

```
#ifndef FUNCTION
#define FUNCTION
#include <stdio.h>
#include <list>
#include <vector>
#include <stack>
#include <string>
#include "FQC.h"

//flex lexical.l && bison -dy sintatical.y
typedef struct GenericList
{
     intptr_t data;
     struct GenericList * next;
} GenericList;

class LetVar
{
     public:
     LetVar(std::string n, FQC fqc): name(n),varFQC(fqc) {}
     FQC varFQC;
     std::string name;
};
class FuncVar
{
     public:
     FuncVar(std::string n,int s): name(n),size(s){}
     int size;
     std::string name;
};

class SymbolTable
{
     public:
     std::vector<FuncVar> vars;
     std::vector<LetVar> letvars;
     int nargs = 0;
     bool AddSymbol(FuncVar v)
     {
         if(contains(v.name)) return false;
         vars.push_back(v);
         return true;
     }
     bool contains(std::string name)
     {
         for(auto& var : vars)
         {
             if(var.name == name) return true;
         }
         for(auto&var : letvars)
         {
             if(var.name == name) return true;
         }
         return false;
     }
```

```
    FuncVar* GetSymbol(std::string name)
    {
        for(auto& var : vars)
        {
            if(var.name == name) return &var;
        }
        return NULL;
    }
    LetVar* GetLetVar(std::string name)
    {
        for(auto& var : letvars)
        {
            if(var.name == name) return &var;
        }
        return NULL;
    }
    void RemoveLetVar()
    {
        letvars.pop_back();
    }
};
class Function
{
    public:
    Function(std::string str): name(str) { symbolTable = new SymbolTable();}
    SymbolTable* symbolTable;
    std::string name;
    int returnType;
    FQC* funcFQC;

};
#endif
```

ANEXO E – Arquivo Variable.h

```
#ifndef VARIABLE
#define VARIABLE

class Variable
{
     public:
     Variable(std::string n, int s):name(n), size(s), value(NULL) { }
     std::string name;
     std::vector<std::string> aliases;
     int size;
     QBit* value;
     void printAllNames()
     {
         printf("Base name %s | ",name.c_str());
         for(auto a : aliases) printf("%s ",a.c_str());
         printf("\n");
     }
     int GetStackIdx(std::stack<QBit*>* qStack)
     {
         std::stack<QBit*> auxStack;
         int idx = -1, i = 0;
         while(!qStack->empty())
         {
             QBit* top = qStack->top();
             if(top== this->value)
             {
                 idx = i;
                 break;
             }
             auxStack.push(top);
             qStack->pop();
             i++;
         }
         //Restores stack
         while(!auxStack.empty()) {
             qStack->push(auxStack.top()); auxStack.pop(); }
         if(idx == -1)
         {
             printf("STACK VAR %s NOT FOUND\n",name.c_str());
         }
         return idx;
     }
     int GetVecIdx(std::vector<QBit*> vec)
     {
         int i = 0;
         for(auto qbit : vec)
         {
             if(qbit == this->value) return i;

             i++;
         }
         printf("VEC VAR %s NOT FOUND\n",name.c_str());
         return -1;
     }
     //Checks if two variables share a name
```

```
    bool operator == (Variable& v)
    {
        //Test Name and Name
    if(v.name == name) return true;
        //Test Aliases and Name
    for(auto myalias : this->aliases) if(myalias == v.name) return true;

    for(auto alias : v.aliases)
    {
        // Test Name and Aliases
        if(this->name == alias) return true;
        //Test Aliases and Aliases
        for(auto myalias : this->aliases) if(myalias == alias) return true;
    }
    return false;
    }
    bool equals(std::string name)
    {
        if(name == this->name) return true;
        for(auto a : this->aliases)  if(name == a) return true;

        return false;
    }
};
#endif
```

ANEXO F – Arquivo FQC.h

```
#ifndef FQCCLASS
#define FQCCLASS
#include "Unitary.h"
class FQC
{
     public:
     int a,h,b,g;
     Unitary* phi;
     std::vector<Variable*> inputVars;
     std::vector<Variable*> auxVars;
     FQC(int aa,int hh,int bb,int gg):a(aa), h(hh), b(bb), g(gg) { }
     FQC(int aa,int hh,int bb):a(aa), h(hh), b(bb), g(aa+hh-bb){}
     FQC(int aa,int hh,int bb,int gg,Unitary* p):a(aa),h(hh),b(bb),g(gg),phi(p){}

     void CreateCondition(FQC* t, FQC* f, bool measure)
     {

         //Execute condition , add measure, execute t, not on first qubit,
         //execute f, not on first qubit
         if(measure)
         {
             MeasureOp* measureCon = inputVars.size() == 1 ?
             new MeasureOp(inputVars.at(0)) : new MeasureOp(auxVars.at(0)) ;
             this->phi->PlusUnitary(measureCon);
             measureCon->TensorUnitary(t->phi);
         }
         else
         {
             this->phi->TensorUnitary(t->phi);
         }
         NotRot* not0 = new NotRot(1);
         // Add Padding the size of T so that F is on the correct qubits
         for(int i = 0; i < t->a+t->h;i++)
         not0->TensorUnitary(new Unitary());
         not0->TensorUnitary(f->phi);

         this->phi->PlusUnitary(not0);
         this->phi->PlusUnitary(new NotRot(1));


         AddFQCVars(t);
         AddFQCVars(f);
         if(inputVars.size()== 1)
         {
             t->phi->SetConditionVar(inputVars.at(0));
             f->phi->SetConditionVar(inputVars.at(0));
         }
         else if (auxVars.size() >= 1)
         {
             t->phi->SetConditionVar(auxVars.at(0));
             f->phi->SetConditionVar(auxVars.at(0));
         }
     }

     void printFQC()
```

```
    {
        printf("----------------------------------------\n");
        printf("FQC = (%d , %d , %d , %d , Phi)\n",a,h,b,g);
        printf("Phi = \n");
        if(phi!= NULL) phi->PrintUnitary(0);
        printf("This FQC uses vars : ");
        for(auto var : inputVars) printf("%s, ",var->name.c_str());
        printf("and ");
        for(auto var : auxVars) printf("%s, ",var->name.c_str());
        printf("\n----------------------------------------\n");
    }
    Variable* GetVar(std::string name)
    {
        for(auto var : inputVars)
        {
            if(var->equals(name)) return var;
        }
        for(auto var : auxVars)
        {
            if(var->equals(name)) return var;
        }
        return NULL;
    }
    void AddFQCVars(FQC* fqc)
    {
        for(auto var : (fqc->inputVars))
        {
            this->inputVars.push_back(var);
        }
        for(auto var : (fqc->auxVars))
        {
            this->auxVars.push_back(var);
        }
    }
    void RenameVar(std::string oldName,std::string newName)
    {
    for(auto var : inputVars)
    {
        if(var->name == oldName) var->name = newName;
        for(auto alias : var->aliases) if(alias == oldName) alias = newName;
    }
    for(auto var : auxVars)
    {
        if(var->name == oldName) var->name = newName;
        for(auto alias : var->aliases) if(alias == oldName) alias = newName;
    }
    }
    void AddInputVar(Variable* v)
    {
        inputVars.push_back(v);
    }
    void AddAuxVar(Variable* v)
    {
        auxVars.push_back(v);
    }
};
#endif
```

ANEXO G – Arquivo Unitary.h

```
#ifndef UNITARY
#define UNITARY

#include "Complex.h"
#include <vector>
#include <stack>
#include <stdio.h>
#include <cmath>
#include "Variable.h"

class Permutation
{
     public:
     std::vector<Variable*> vars;
     Permutation() { }
     Permutation(std::vector<Variable*> v): vars(v) {}
     void printVars()
     {
         for(auto var : vars) printf("%s ",var->name.c_str());
         printf("\n");
     }
};
class Copy
{
     public:
     Variable* var,* var2;
     Copy(Variable* v,Variable* v2):var(v),var2(v2) { }
     void printVars()
     {
         printf("%s and %s\n",var->name.c_str(),var2->name.c_str());
     }
};
class Swap
{
     public:
     int offset, length, totalsize;
     std::string altName = "";
     Swap( int o, int l, int n) : offset(o), length(l), totalsize(n) { }
     Swap( std::string name ): altName(name), offset(0), length(0), totalsize(0) {}
     void printSwap(){
         printf(" %s(%d,%d,%d)\n",altName.c_str(),offset,length,totalsize);
    }

};
class Rotation
{
     public:
     Complex rotation[2][2];
     enum class RotType { None, Identity, Not };
     RotType type = RotType::None;
     void InitIdentity()
     {
         type = RotType::Identity;
         rotation[0][0] = Complex(1,0);
         rotation[0][1] = Complex(0,0);
```

```
    rotation[1][0] = Complex(0,0);
    rotation[1][1] = Complex(1,0);
}
void InitNot()
{
    type = RotType::Not;
    rotation[0][0] = Complex(0,0);
    rotation[0][1] = Complex(1,0);
    rotation[1][0] = Complex(1,0);
    rotation[1][1] = Complex(0,0);
}
void InitConst(Complex a,Complex b)
{
    type = RotType::None;
    rotation[0][0] = a;
    rotation[0][1] = b;
    rotation[1][0] = b;
    rotation[1][1] = a;
}
bool operator ==(const Rotation& u)
{
    if(this->rotation[0][0] == u.rotation[0][0] &&
       this->rotation[0][1] == u.rotation[0][1] &&
       this->rotation[1][0] == u.rotation[1][0] &&
       this->rotation[1][1] == u.rotation[1][1])
    {
        return true;
    }
    else
        return false;
}
void printRotation()
{
    if(type == RotType::Identity)
    {
        printf("Identity Matrix\n");
        return;
    }
    if(type == RotType::Identity)
    {
        printf("Not Matrix\n");
        return;
    }
    printf("(%f + %fi, %f + %fi)(%f + %fi, %f + %fi)\n",
    rotation[0][0].r,rotation[0][0].i,rotation[0][1].r,rotation[0][1].i,
    rotation[1][0].r,rotation[1][0].i,rotation[1][1].r,rotation[1][1].i);
}
void MatrixToAngles(double* theta,double* lambda,double* phi)
{
    if(type == RotType::Identity)
    {
        *theta = 0;
        *lambda = 0;
        *phi = 0;
        return;
    }
    if(type == RotType::Not)
    {
```

```
            *theta = 3.141593;
            *lambda = 3.141593;
            *phi = 0;
            return;
        }
        //printRotation();
        *theta = 2*acos(rotation[0][0].r);
        if(*theta == 0)
        {
            if(rotation[1][1].r == -1)
            {
                *lambda = 3.141593/2;
                *phi = 3.141593/2;
            }
            else
            {
                *lambda = 0;
                *phi = 0;
            }
        }
        else
        {
            Complex div = Complex(sin(*theta/2),0);
            Complex lambdaMul = Complex(-1,0)*rotation[0][1]/div;
            *lambda = lambdaMul.r > 0 ?
            log(lambdaMul.Magnitude()) : acos( lambdaMul.r );


            Complex phiMul =  Complex(rotation[1][0]/div);
            *phi = phiMul.r > 0 ?
            log(phiMul.Magnitude()) :acos( phiMul.r );
        }
    }
};


class Unitary
{
public:
enum class UType { Rot, Swap, Copy, Perm, Measure, Null };
UType type;
union {
    Rotation* rot;
    Swap* swap;
    Copy* copy;
    Permutation* perm;
    Variable* measureVar;
};
Unitary* nextP = NULL;
Unitary* nextT = NULL;
Variable* condition = NULL;
Unitary() { type = UType::Null; }

void PlusUnitary(Unitary* ut)
{
    if(nextP == NULL)
        nextP = ut;
    else
```

```
        nextP->PlusUnitary(ut);
}
void TensorUnitary(Unitary* ut)
{
     if(nextT == NULL)
        nextT = ut;
     else
        nextT->TensorUnitary(ut);
}
void PrintUnitary(int tab = 0)
{
     for(int i = 0; i < tab;i++) printf("\t");
     printf("Unitary Operation of type ");
     if(type == UType::Rot) { printf("Rotation: ",rot);rot->printRotation();}
     if(type == UType::Swap) { printf("SwapN:"); swap->printSwap(); }
     if(type == UType::Perm) {printf("Permutation of :"); perm->printVars();
}
     if(type == UType::Copy) {printf("Copy operation of:"); copy->printVars();
}
     if(type == UType::Measure) { printf("Measure\n"); }
     if(type == UType::Null) { printf("Null operation\n"); }
     if(nextT != NULL)
     {
        nextT->PrintUnitary(tab+1);
     }
     if(nextP != NULL)
     {
        nextP->PrintUnitary(tab);
     }

}
void SetConditionVar(Variable* c)
{
     if(condition != NULL) return;

     condition = c;
     if(nextP != NULL) nextP->SetConditionVar(c);
     if(nextT != NULL) nextT->SetConditionVar(c);

}
std::string Compile(int index, std::stack<QBit*>* stk)
{
     std::string compiledUnitary = "";
     switch(type)
     {
     case UType::Rot:
        compiledUnitary.append(this->CompileRotation(index,stk));
        break;
     case UType::Swap:
        compiledUnitary.append(this->CompileSwap(stk));
        break;
     case UType::Perm:
        compiledUnitary.append(this->CompilePermutation(stk));
      break;
     case UType::Copy:
        compiledUnitary.append(this->CompileCopy(stk));
      break;
     case UType::Measure:
```

```
            compiledUnitary.append(this->CompileMeasure(stk));
         break;
        default:  compiledUnitary.append("");
        }

        if(nextT != NULL)
        {
            std::string tensorStr = nextT->Compile(index+1,stk);
            compiledUnitary.append(tensorStr);
        }
        if(nextP != NULL)
        {
            std::string sumStr = nextP->Compile(index,stk);
            compiledUnitary.append(sumStr);
        }
        return compiledUnitary;
}
protected:
std::string CompileRotation(int index,std::stack<QBit*>* stack)
{
        double t,p,l;
        rot->MatrixToAngles(&t,&l,&p);
        if(t == 0 && p == 0 && l == 0) return std::string("");
        if(rot->type == Rotation::RotType::Not)
        {
            char buffer[25];
            if(condition == NULL)
            {
                sprintf(buffer,"NOT %d\n",index);
            }
            else
            {
                sprintf(buffer,"CNOT %d %d\n",condition->GetStackIdx(stack),index);
            }
            return std::string(buffer);
        }
        char buffer[50];
        if(condition == NULL)
        {
            sprintf(buffer,"U3 %d %f %f %f\n",index,t,p,l);
        }
        else
        {
            sprintf(buffer,"CU3 %d %d %f %f %f\n",
            condition->GetStackIdx(stack),index,t,p,l);
        }
        return std::string(buffer);
}
std::string CompileSwap(std::stack<QBit*>* stack)
{
        //Generate Swap strings
        std::string swapStr;
        int s = stack->size();
        for(int i = 0; i < swap->length; i++)
        {
            char buffer[25];
            if(condition == NULL)
            {
```

```
            //Avoid swap with itself
            if(swap->offset+i != i)
                sprintf(buffer,"SWAP %d %d \n",swap->offset+i,i);
            else
            {
                sprintf(buffer,"");
            }
        }
        else
        {
            if(swap->offset+i != i &&
            condition->GetStackIdx(stack) != swap->offset+i &&
            condition->GetStackIdx(stack) != i)
                sprintf(buffer,"CSWAP %d %d %d \n",
                condition->GetStackIdx(stack),swap->offset+i,i);
            else
            {
                sprintf(buffer,"");
            }
        }
        swapStr.append(buffer);
    }
    //Execute those swaps in the qubit stack
    std::stack<QBit*> offsetStack;
    std::stack<QBit*> lenStack;
    //Destack offset
    for(int i = 0; i < swap->offset; i++)
    { offsetStack.push(stack->top()); stack->pop(); }
    //Destack length
    for(int i = 0; i < swap->length; i++)
     {lenStack.push(stack->top()); stack->pop(); }
    //Stack offset back
    for(int i = 0; i < swap->offset; i++)
    { stack->push(offsetStack.top()); offsetStack.pop(); }
    //Stack length back
    for(int i = 0; i < swap->length; i++)
    { stack->push(lenStack.top()); lenStack.pop(); }


    return swapStr;
}
std::string CompilePermutation(std::stack<QBit*>* stack)
{
    int i = 0;
    std::vector<QBit*> auxVector;
    //Builds vector as reverse stack
    while(!stack->empty()) {
        auxVector.push_back(stack->top());
        stack->pop();
    }
    int last = auxVector.size() -1;

    std::string permstr = std::string("");

    for(auto var : perm->vars)
    {
        int vecIdx = var->GetVecIdx(auxVector);
        for(int s = 0; s < var->size; s++ )
```

```
            {
                char buffer[25];
                if(i != vecIdx)
                {
                    if(condition == NULL)
                    {
                        //Avoid swap with itself
                        if(s+i != s+vecIdx) sprintf(buffer,"SWAP %d %d\n",s+i,s+vecIdx);
                        else sprintf(buffer,"");
                    }
                    else
                    {
                        if(s+i != s+vecIdx &&
                        condition->GetVecIdx(auxVector) != s + i &&
                        condition->GetVecIdx(auxVector)  != s+vecIdx)
                            sprintf(buffer,"CSWAP %d %d %d\n",
                            condition->GetVecIdx(auxVector),s+i,s+vecIdx);
                        else
                            sprintf(buffer,"");
                    }
                    permstr.append(buffer);
                    QBit* aux = auxVector.at(s+i);
                    auxVector.at(s+i) = auxVector.at(s+vecIdx);
                    auxVector.at(s+vecIdx) = aux;
                }
                else sprintf(buffer,"");
                i++;
             }
        }
        //Restore stack (reverse iterator)
        for(auto qbit = auxVector.rbegin(); qbit < auxVector.rend(); qbit++)
        {
            stack->push(*qbit);
        }

        return permstr;
}
std::string CompileCopy(std::stack<QBit*>* stack)
{
        std::string copystr = std::string("");
        for(int i = 0; i < copy->var->size; i++)
        {
            char buffer[25];
            if(condition == NULL)
            {
                sprintf(buffer,"CNOT %d %d\n"
                ,copy->var2->GetStackIdx(stack)+i,copy->var->GetStackIdx(stack)+i);
            }
            else
            {
                sprintf(buffer,"CCNOT %d %d %d\n"
                ,condition->GetStackIdx(stack),copy->var2->GetStackIdx(stack)+i
                ,copy->var->GetStackIdx(stack)+i);
            }
            copystr.append(buffer);
        }
        return copystr;
}
```

```
std::string CompileMeasure(std::stack<QBit*>* stack)
{
     char buffer[25];
      sprintf(buffer,"Measure %d\n",measureVar->GetStackIdx(stack));
     return std::string(buffer);
}
};
class IdentityRot : public Unitary
{
     public:
     IdentityRot(int len)
     {
         if(len == 0)
         {
              return;
         }
         type = UType::Rot;
         while(len != 0)
         {
             rot = new Rotation();
             rot->InitIdentity();
             if(len != 1)
             {
                 Unitary* ideRot = new Unitary();
                 ideRot->type = UType::Rot;
                 ideRot->rot = new Rotation();
                 ideRot->rot->InitIdentity();

                 if(nextT == NULL)
                 {
                     nextT = ideRot;
                 }
                 else
                 {
                     nextT->TensorUnitary(ideRot);
                 }
             }
             len--;
         }
     }
};
class ConstRot: public Unitary
{
     public:
     ConstRot(Complex a,Complex b)
     {
         type = UType::Rot;
         rot = new Rotation();
         rot->InitConst(a,b);
     }
};
class NotRot: public Unitary
{
     public:
     NotRot(int len)
     {
         type = UType::Rot;
         if(len == 0) return;
```

```
            rot = new Rotation();
            rot->InitNot();

            if(len != 1) nextT = new NotRot(len-1);
        }
};
class SwapOp : public Unitary
{
     public:
     SwapOp(int o,int l,int n)
     {
            type = UType::Swap;
            swap = new Swap(o,l,n);
     }
};
class CopyOp: public Unitary
{
     public:
     CopyOp(Variable* v,Variable* v2)
     {
            type = UType::Copy;
            copy = new Copy(v,v2);
     }
};
class PermOp: public Unitary
{
     public:
     PermOp(std::vector<Variable*> v)
     {
            type = UType::Perm;
            perm = new Permutation(v);
     }
};
class MeasureOp : public Unitary
{
     public:
     MeasureOp(Variable* v)
     {
            type = UType::Measure;
            measureVar = v;
     }
};
#endif
```

ANEXO H – Arquivo Complex.h

```
#ifndef COMPLEXQUBIT_H
#define COMPLEXQUBIT_H
#include <math.h>
class Complex
{
     public: double r, i;
     Complex(double x, double y): r(x), i(y) {}
     Complex(const Complex& c):r(c.r),i(c.i) {}
     Complex(): r(0), i(0) {}

     Complex operator +(const Complex& c)
     {
        return Complex(r+c.r,i+c.i);
     }
     Complex operator -(const Complex& c)
     {
        return Complex(r-c.r,i-c.i);
     }
     Complex operator *(const Complex& c)
     {
         // (a+bi)(c+di)=(ac-bd)+(ad+bc)i
         return Complex(r*c.r - i*c.i,r*c.i + i*c.r);
     }
     Complex operator /(const Complex& c)
     {
        // (a+bi)/(c+di) = (ac+bd)/(c*c + d*d) + i*(bc-ad)/(c*c + d*d)
         double div = c.r*c.r + c.i*c.i;
         return Complex((r*c.r + i*c.i)/div,(i*c.r-r*c.i)/div);
     }
     bool operator == (const Complex& c)
     {
         if(r == c.r && i == c.i) return true;
         return false;
     }
     double Magnitude()
     {
         return sqrt(r*r + i*i);
     }
};
class QBit
{
     public : Complex left, right;
     QBit():left(0,0), right(0,0) { }
     QBit(Complex a,Complex b) : left(a), right(b) { }
     void Normalize()
     {
         double leftM = left.Magnitude();
         double rightM = right.Magnitude();
         double div = sqrt(leftM*leftM + rightM*rightM);

         left.r /= div; left.i /= div;
         right.r /= div; right.i /= div;
     }
};
#endif
```

ANEXO I – Arquivo ——

```
def AddOperationToCircuit(Line,Circuit,QRegs,CRegs):
    operation = Line[0]
    fargs = []
    args = []
    for arg in range(1,len(Line)):
        fargs.append(float(Line[arg]))
        args.append(math.floor(float(Line[arg])))
    if operation == "U3":
        Circuit.u(fargs[1],fargs[2],fargs[3],QRegs[args[0]])
    elif operation == "CU3":
        Circuit.cu(fargs[2],fargs[3],fargs[4],0.0,QRegs[args[0]],QRegs[args[1]])
    elif operation == "SWAP":
        Circuit.swap(QRegs[args[0]],QRegs[args[1]])
    elif operation == "CSWAP":
        Circuit.cswap(QRegs[args[0]],QRegs[args[1]],QRegs[args[2]])
    elif operation == "NOT":
        Circuit.x(QRegs[args[0]])
    elif operation == "CNOT":
        Circuit.cx(QRegs[args[0]],QRegs[args[1]])
    elif operation == "CCNOT":
        Circuit.ccx(QRegs[args[0]],QRegs[args[1]],QRegs[args[2]])
    elif operation == "Measure":
        Circuit.measure(QRegs[args[0]],CRegs[args[0]])

def ReadParsedCircuit(FileName):
    file = open(FileName,'r')
    qBitNum = file.read(2)[0]
    qBitNum = int(qBitNum)
    print(str(qBitNum)+" Qbits")
    quantumRegisters = QuantumRegister(qBitNum)
    classicalRegisters = ClassicalRegister(qBitNum)
    compiledCircuit = QuantumCircuit(quantumRegisters,classicalRegisters)
    for line in file:
        lineArray = line.split()
        AddOperationToCircuit\
    (lineArray,compiledCircuit,quantumRegisters,classicalRegisters)
    return compiledCircuit

token = #Token da sua conta pessoal no site IBMQ
    IBMQ.save_account(token,overwrite=True)
    IBMQ.load_account()
    #Constrói Circuito
    QCircuit = ReadParsedCircuit('circuito.txt')
    nShots = 8192

    #Usar simulador
    simulator = Aer.get_backend('qasm_simulator')
    job = simulator.run(QCircuit,nShots)
    result = job.result()

    #Usar computador quântico
    provider = IBMQ.get_provider('ibm-q')
    qcomputer = provider.get_backend('ibmq_belem')
    job = execute(QCircuit, backend = qcomputer, shots = nShots)
    result = job.result()
```