RWTH AACHEN UNIVERSITY
Chair of Computer Science 2
Software Modeling and Verification

Master Thesis

Compilation of Quantum Programs with Control Flow Primitives in Superposition

Sascha Thiemann
Matr.-No.: 406187
Study Program: Computer Science M.Sc.
January 17, 2025

Supervisors: apl. Prof. Dr. Thomas Noll
Chair for Software Modeling and Verification
RWTH Aachen University

Prof. Dr. rer. nat. Dominique Unruh
Chair for Quantum Information Systems
RWTH Aachen University

Contents

1 Introduction

Decades ago, the field of quantum computing emerged as a theoretical combination of quantum physics and computer science. Over the years, quantum algorithms were conceived that exploit the special properties of quantum computers to gain a speedup over classical implementations that is, in parts, exponential [BeVa93]. Some of these algorithms have the potential to significantly impact the world of computer science if or when they are implemented on powerful enough quantum hardware. For example, Shor's algorithm [Shor97] provides a method for efficiently factoring prime numbers. Since many classical cryptographic encryptions rely on the complexity of factoring large primes, Shor's algorithm on powerful quantum computers may break these cryptosystems [DiCh20].

However, while the theoretical background of quantum computing is extensive and many algorithms have been proposed, the physical technology of quantum computers is not advanced enough for practical use. In turn, most quantum programming languages are used in the context of simulations, instead of practical applications, and focus on the low-level representation of quantum circuits. One such language is the OpenQASM [CBSG17] language. OpenQASM programs consist mainly of quantum bits that can be manipulated by quantum gates. Additionally, their state can be measured, and the results can be used in limited classical if-statements to control which quantum gates are applied. In contrast, there are also languages with a focus on high-level interactions, e.g., Tower [YuCa22], which contains data structures in superposition, and Silq [BBGV20], which allows for the automatic uncomputation of temporary values.

What all these languages have in common is the restriction to quantum data while using only classical control flow. The idea of control flow in superposition was first used in a functional quantum programming language proposed by Altenkirch et al. [AlGr05]. Later on, the concept of quantum control flow was defined by Ying et al. [YYF12]. Since then, only a few languages have incorporated the principle; only recently, the Quantum Control Machine, an instruction set architecture for quantum computers with quantum control flow at its core, was proposed by Yuan et al. [YVC24]. The instruction set is similar to the instructions provided by classical assembly languages, having jump and condition jump instructions. Additionally, it is specially designed for the limitations of control flow in superposition. However, the low-level design of the architecture and syntax, similar to assembly languages, combined with general restrictions on quantum programs with control flow in superposition, results in complex programs that are both hard to read and write.

In our project, we want to build upon the ideas of the quantum control machine and control flow in superposition and design a high-level language with control flow

primitives in superposition. This abstraction should be similar to the abstraction from classical assembly languages to classical high-level languages, where, e.g., classical jump instructions are abstracted to if-clauses or loops. Furthermore, the resulting language should have the same capabilities as the quantum control machine but should be more accessible, i.e., make programs easier to read and write. Besides the language design, we also plan to implement a corresponding compiler that compiles our custom language to an existing quantum language. Moreover, the compiler should consist of all the basic elements of a compiler, including the lexical, syntactic, and semantic analysis as well as code generation and optimization.

2 Background

In the following section, we introduce and discuss different concepts that are referenced in later parts of this thesis. Firstly, we give a general introduction into quantum computing with some basic background knowledge on how quantum computers work and which quantum mechanical principles are essential for them. Additionally, we discuss more specific knowledge about quantum algorithms and optimization techniques for quantum circuits. Next, we discuss quantum control flow in more detail; this includes the formal definitions and its limitations. Then, we review existing quantum programming languages in general and some specific examples. Lastly, we give an overview of the topic of compilation and the different phases of a compiler.

2.1 Quantum Computing

While computers are prevalent and important in today's society, there are many relevant problems that classical computers cannot currently and perhaps will never realistically be able to solve. Quantum computing (QC) is gaining more momentum as the technology that could solve at least some of these problems. For example, quantum algorithms like Shor's algorithm [Shor97] could provide a significant improvement for prime factorization given sufficient technology. Therefore, it is estimated to be a valuable market with many of the largest technology companies as well as governments investing billions in the research and development of quantum technology [RDB*22, Pres18]. In the following section, we take a look at the basic concepts of a quantum computer and the core principles it relies on.

Classical computers are based on simple operations executed on bits, like `and`, `or`, and `not`. These bits can either have a value of 0 or 1. Similarly, at their core, quantum computers apply simple operations, like `controlled-not` and `Hadamard`, on quantum bits (qubits). On a higher level, a classical computer executes operations on a register, consisting of multiple bits, while a quantum computer operates on quantum registers, consisting of multiple qubits. In contrast to classical bits, quantum computers use the unique properties of quantum mechanics to enable qubits to have not just one value of either 0 or 1 but a combination of both. The phenomenon, where a particle or qubit exists in a combination of both states, is called *superposition*. Additionally, quantum computers also use the idea of *entanglement* to their advantage. Two qubits are entangled when the value of one is dependent on the value of the other. The combination of superposition and entanglement enables quantum computers to solve specific problems more efficiently than classical computers [RDB*22].

Models for quantum computers can be divided into three main categories: the *analog model*, the *measurement-based model*, and the *gate-based model*. The analog model uses

smooth operations to evolve a quantum system over time such that the resulting system encodes the desired result with high probability. It is not clear whether this model allows for universal quantum computation or quantum speedup [DiCh20b]. Instead of smoothly evolving a system, the measurement-based model starts with a fixed quantum state, the cluster state. The computation is accomplished by measuring qubits of the system, possibly depending on the results of previous measurements. While there are different measurement-based models, one technique to apply gates is to leverage quantum teleportation, so-called gate teleportation [Jozs05]. The result is a bit string of the measurement results [DiCh20b, Niel06]. Lastly, the gate-based model uses a digitized, discrete set of qubits that are manipulated by a sequence of operations represented by quantum gates. The result is obtained by measuring the qubits at the end of the computation. Often, intermediate measurements are also supported. Although digital quantum computation is more sensitive to noise than analog computations, the digitization can also be used for quantum error correction [DMN13] and to mitigate the increased noise [DiCh20b]. Furthermore, because qubits are actively manipulated and not passively evolved, digital quantum computers are more flexible than analog ones [RDB*22]. Therefore, the gate-based model is the most common model, and this thesis will mainly focus on it.

2.1.1 Superposition

The first important property of quantum mechanics used by quantum computers is the idea of superposition. Qubits in superposition are often informally described as simultaneously having a value of 0 and 1 until their state is measured. However, a qubit in superposition is more formally a linear combination of its basis states. The basis states are the states where the qubit has a value of 0, written $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$, and 1, written $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$ [DiCh20a]. Furthermore, the state can be reduced to a simple vector. Therefore, a state ψ in superposition can be written as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix}.$$

The factors α and β are the amplitudes of the basis states and are complex numbers. The factors must also satisfy the condition $|\alpha|^2+|\beta|^2 = 1$. This is a result of the relation between the amplitudes and the probability to which basis state the state will collapse when measured, described in Sec. 2.1.4.

Besides $|0\rangle$ and $|1\rangle$, there exist more relevant short-hands for quantum states. For example, $|+\rangle$ and $|-\rangle$ are states in uniform superposition, i.e., both basis states are equally likely, and they are often used when discussing quantum states and transformations. They are defined as follows:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} \quad \text{and} \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}.$$

2.1.2 Entanglement

Another important quantum mechanical concept is entanglement. Simply said, two qubits are entangled when their values depend on each other. An example would be a quantum system where two qubits are in superposition and equally likely to collapse to either $|0\rangle$ or $|1\rangle$; whichever value one qubit collapses to when measured, the second one will also collapse to the same value. Additionally, changes to one of the qubits can also affect the other one. This happens independent of the locations of the two qubits [RDB*22, HHHH09].

A more formal definition for an entangled state uses the definition of a composite system. Two separate quantum systems can be represented as a single system with the tensor product of both systems. For example, the combined state $|\psi\rangle$ of the separate states $|0\rangle$ and $|1\rangle$ can be represented as:

$$|\psi\rangle = |0\rangle \otimes |1\rangle = |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

When a quantum state cannot be expressed as a tensor product of two states, the state is entangled. The previous example is a case of a maximally entangled Bell state [DiCh20a, MHH19], often denoted β_{00}, and can be expressed as the following:

$$\beta_{00} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

The entanglement of states is used by leveraging the effect of the qubits on each other to collaborate to calculate the result. Although this can be simulated on classical computers, it cannot be achieved "natively" because all classical bits are independent of each other. Moreover, quantum algorithms not using entangled states can often be simulated efficiently on classical computers [MHH19]. Therefore, entanglement is at the core of quantum computing, but it can also have unintended consequences one needs to be aware of when designing quantum algorithms.

To calculate specific functions or intermediate values, quantum algorithms may need to use additional qubits or registers whose state can, in turn, be entangled with the main data of the algorithm. If this entanglement is not resolved in time, the changes to the qubit or register can interfere with future calculations or measurements and cause the results to be invalid. This effect is called *disruptive entanglement* [YVC24]. To prevent disruptive entanglement, the concept of *uncomputation*, formalized by Bennett [Benn73], can be used. The idea is to apply computations such that the entanglement of the main data and auxiliary qubits is undone. One way to do this is to apply the inverse of applied computations in reverse order [DiCh20].

2.1.3 Quantum Gates

In gate-based quantum computers, the transformations applied to the quantum data are represented by *quantum gates*. Similar to quantum states, which can be represented by linear combinations of basis states, or vectors, quantum gates can be formulated as linear transformations of these combinations, or a matrix. Because the result of such a transformation also needs to be a valid quantum state, the transformation needs to be norm-preserving, or *unitary* [DiCh20a]. The most relevant and often used unitary gates are depicted in Tab. 2.1.

	Gates	Matrix	Ket-notation
Pauli gates	X	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\|0\rangle \mapsto \|1\rangle$ $\|1\rangle \mapsto \|0\rangle$
	Y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\|0\rangle \mapsto \|i\rangle$ $\|1\rangle \mapsto -\|i\rangle$
	Z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\|0\rangle \mapsto \|0\rangle$ $\|1\rangle \mapsto -\|1\rangle$
Hadamard gate	H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\|0\rangle \mapsto \|+\rangle$ $\|1\rangle \mapsto \|-\rangle$
Phase gate	$P(\lambda)$	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$	$\|0\rangle \mapsto \|0\rangle$ $\|1\rangle \mapsto e^{i\lambda} \cdot \|1\rangle$
Controlled-not gate	CX	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	$\|00\rangle \mapsto \|00\rangle$ $\|01\rangle \mapsto \|01\rangle$ $\|10\rangle \mapsto \|11\rangle$ $\|11\rangle \mapsto \|10\rangle$
Toffoli gate	CCX	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$	$\|000\rangle \mapsto \|000\rangle$ $\|001\rangle \mapsto \|001\rangle$ $\|010\rangle \mapsto \|010\rangle$ $\|011\rangle \mapsto \|011\rangle$ $\|100\rangle \mapsto \|100\rangle$ $\|101\rangle \mapsto \|101\rangle$ $\|110\rangle \mapsto \|111\rangle$ $\|111\rangle \mapsto \|110\rangle$

Table 2.1: A list of relevant quantum gates in matrix representation as well as functions in ket-notation.

A matrix U is unitary if it has an inverse matrix that is equal to its conjugate transpose $U^\dagger$, i.e., the following must hold:

$$UU^\dagger = I.$$

Therefore, all transformations applied to quantum states in a gate-based quantum computer must be reversible by definition. This limitation does not apply to classical computers, where non-reversible transformations, e.g., mapping an arbitrary bit to a specific value, are easily implementable.

To design a useful quantum computer or language, the set of gates should be *universal*. A set of gates is universal if any gate can be simulated by a combination of the gates from the set with arbitrary accuracy [BrBr02]. An example of a universal set of gates is the combination of the Toffoli gate together with the Hadamard gate [DiCh20a].

2.1.4 Measurement

For a quantum computer to be of any use, we need a way to read out information about its state. However, the information we can obtain from a quantum system is limited by the quantum measurement postulate. The postulate states that the only way to gain any information from a quantum system is to measure it. When measuring a quantum state, the state irreversibly collapses to one of its basis states. Furthermore, this is a probabilistic transformation, and the original state in superposition cannot be recovered from the result. Therefore, in contrast to all other transformations, measurements are neither unitary nor reversible. For a state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, the measurement collapses the state to $|0\rangle$ with a probability of $|\alpha|^2$. Correspondingly, the state will collapse to $|1\rangle$ with a probability of $|\beta|^2$ when measured [DiCh20a].

Measurement can be represented as a measurement basis set $\{M_i\}_i$ which requires the following condition:

$$\sum_i M_i^\dagger M_i = I.$$

The probability that outcome i is obtained when measuring a state $|\psi\rangle$ is equivalent to $|M_i\,|\psi\rangle|^2$. After the measurement of outcome i, the state $|\psi'\rangle$ will be equivalent to

$$\left|\psi'\right\rangle = \frac{M_i\,|\psi\rangle}{|M_i\,|\psi\rangle\,|} = \frac{M_i\,|\psi\rangle}{\sqrt{\Pr\left[\text{observe } i\right]}}.$$

Additionally, measurements in quantum programming languages can be differentiated between implicit and explicit measurements. Implicit measurements rely on two concepts. Any discarded qubit is implicitly measured, and any measurement can be delayed until the end of a circuit. In turn, instead of explicitly measuring a qubit in a quantum circuit or program, all qubits can be implicitly measured. However, some quantum communication protocols may rely on the explicit measurements of qubits. Therefore, implicit measurements are not always the best solutions, and many quantum languages include explicit measurements [Inou24].

2.1.5 Relevant Algorithms

Since quantum computers differ greatly from classical computers not only in their technology but also in the concepts they use for calculation, they cannot function without specially designed algorithms. These algorithms need to exploit the special quantum properties of qubits to achieve *quantum advantage*, i.e., a better complexity than any classical algorithm. One of the first algorithms to show its quantum advantage was the Deutsch–Josza algorithm [DeJo92]. Deutsch et al. define a problem that can be solved

in exponential time on classical computers and present a quantum algorithm that can solve the problem in polynomial time. The Bernstein-Vazirani algorithm [BeVa93] is another example with shown quantum advantage, resulting in a polynomial speed-up. However, currently, there does not exist a use case for either of the algorithms, and, therefore, they are only of limited theoretical interest [DiCh20c].

An algorithm with more potential for practical use is Shor's algorithm [Shor97]. It presents a more efficient, polynomial-time quantum implementation for the discrete logarithm, i.e., find r for a given a, x, and p such that $a^r = x \mod p$. The algorithm is of special interest because Shor also provides a reduction of prime factorization to order finding; order finding is a special case of the discrete logarithm where $x = 1$. Modern cryptography is often based on the complexity of factoring large prime numbers, e.g., the commonly used RSA cryptosystem [RSA78]. Therefore, an advanced quantum computer could break these systems with Shor's algorithm [MVDJ18]. Not only does this prospect provide a practical use case for QC, but it also results in the research field of *post-quantum cryptography* [BeLa17].

Another relevant algorithm or transformation is the quantum Fourier transform (QFT) [Copp02]. Besides being used as a subroutine in Shor's algorithm, it is also relevant for other algorithms, e.g., an addition algorithm for quantum registers [Drap00]. Similar to the discrete Fourier transform [Wino78], which operates on vectors, the QFT_{2^n} operates on the quantum equivalent of vectors, quantum registers, of size n. Registers of size n consist of n qubits. From the register, the QFT extracts the present periodic features. Then, other algorithms can use these features for their calculations.

2.1.6 Circuit Optimization

Despite the expansive theoretical foundations for QC, the current state of the art for its technology is limited. However, the technology is nearing its first milestone towards usable quantum computers with the advent of prototypes with noisy intermediate-scale quantum (NISQ) technology [BFA22]. Nevertheless, the technology is still far away from fault-tolerant quantum computers and, by definition, limited in the number of available qubits. Furthermore, the gate count of NISQ-era quantum computers is limited by the inherent noise, which is increased with each additional transformation [Pres18]. Therefore, attributes such as the gate count of a quantum algorithm are an important metric for its utility. To improve the utility of an algorithm, its quantum circuit can be optimized with different techniques and rules.

There exist many kinds of optimization techniques for quantum circuits. They are mostly concerned with optimizing the gate count of quantum circuits with the use of peephole optimizations, as described in Sec. 2.4.5. These techniques can range from general rules [GaCh11, LBZ21], which can be applied to all quantum circuits, to hardware-specific optimizations [KMO*23]. Furthermore, machine learning-based optimization frameworks for quantum circuits are also gaining popularity [FNML21, LPM*24, RLB*24].

The simplest general optimizations are so-called *null gates* [GaCh11]. They are gate combinations or gates under specific conditions that are equivalent to the identity gate

I. Therefore, any occurrence of such a null gate can be removed from the circuit. The most basic example of null gates is the double application of a self-inverse gate, i.e., a gate that is its own inverse. These include the H, X, Y, and Z gates. The identities are also depicted in Fig. 2.1. Furthermore, the same holds for any controlled version of a self-inverse gate such that the rule can also be applied to the controlled-not gate and similar ones.

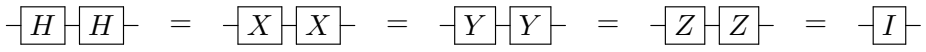

Figure 2.1: Null gates of self-inverse gates.

The second kind of null gates are gates that do not have an effect under specific conditions. For example, a controlled gate U has no effect on its qubits if we know that the control is $|0\rangle$. Similarly, the X gate does not have an effect on a qubit in the $|+\rangle$ state. Hence, the two circuits depicted in Fig. 2.2 are null gates and semantically equivalent to an identity gate.

Figure 2.2: Null gates for gates in specific conditions.

Another class of optimizations is called *control reversal*. Control reversal describes gate combination equalities based on the symmetry of the controlled-Z gate. For the controlled-Z gate, it is semantically equivalent to apply the Z gate to the second wire with a control on the first and to apply the Z gate on the first wire with a control on the second one. Based on this and together with the equalities $HZH = X$ and $HXH = Z$, a controlled-not gate, surrounded by H gates on both wires, can be represented as the reversed controlled-not gate. Both equalities are depicted in Fig. 2.3 and Fig. 2.4.

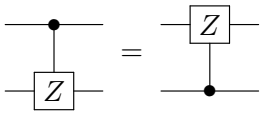

Figure 2.3: Control reversal of the controlled-Z gate.

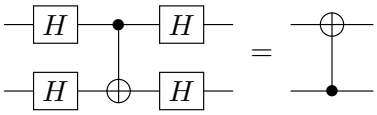

Figure 2.4: Control reversal of the controlled-not gate.

In contrast to general optimization rules, hardware-specific optimizations are mostly not concerned with the reduction of gates based on mathematically equal gate combinations; however, they exploit either the specific properties of the hardware for optimizations or replace gates with cheaper equivalents on the specific hardware. For example, a shuttling-based trapped-ion quantum computer operates by physically moving ions to segments in the hardware where operations can be applied. Since ions can

and must be freely moved, swapping qubits can easily be accomplished by physically changing the position of their hardware equivalent. On the software side, this can be achieved by removing the swap gate and swapping all following instances of both qubits [KMO*23].

2.2 Quantum Control Flow

The idea of quantum control flow was first used by Altenkirch et al. [AlGr05] when defining a functional programming language with quantum control flow elements. The language uses an if-statement in superposition, `if`$^\circ$, which is used to, e.g., define the Hadamard gate as a function *had* instead of a matrix. The *had* function takes a qubit as an input. If the qubit is true, i.e., it has a value of $|1\rangle$, the function returns a uniform superposition of true and false, where true has a negative sign. Correspondingly, for a false input, a uniform superposition with both signs positive is returned.

$$\begin{aligned} had : Q &\to Q \\ had : x &\mapsto \texttt{if}^\circ x \\ &\quad \texttt{then } \{false \mid -true\} \\ &\quad \texttt{else } \{false \mid true\} \end{aligned}$$

Quantum control flow can be divided into *quantum branching* and *iteration* [YVC24]. In the following, we will discuss both branching and iteration in superposition as well as the limitations of quantum control flow.

2.2.1 Branching

Based on the work presented by Altenkirch et al. [AlGr05], the concept of quantum control flow, more specifically quantum branching, was expanded on and formally defined by Ying et al. [YYF12]. They introduced two different types of quantum branching: quantum guarded commands and quantum choices as a special case of guarded commands. The definition of quantum guarded commands is based on Dijkstra's guarded commands [Dijk75]. Guarded commands concern the nondeterministic execution of functions based on Boolean expressions, where the nondeterminism derives from the possible overlapping of the guards. In contrast, quantum branching allows for the execution of functions based on a value in superposition. The functions are executed such that the result may be a superposition of the results of the individual functions [YVC24]. Quantum branching is, e.g., used in simulation algorithms like [BGB*18] and [LoCh19]. Furthermore, many basic concepts, such as controlled gates, can be represented as quantum branching or even single-qubit gates, as seen in the previous example of the Hadamard implementation.

The formal definition for classical guarded commands is given by:

$$\square_{i=1}^{n} b_i \to C_i$$

where C_i is a command guarded by a Boolean expression b_i. The command can only be executed if the expression is true. Similarly, quantum guarded commands map to a set of quantum programs P_i. Further, a set of qubits or quantum registers and a corresponding orthogonal basis $|i\rangle$ is given. However, the set of qubits guarding the program must be disjoint from the set of qubits used in the program. Without this condition, the resulting operation may not be unitary. For example, an X-gate that is executed if the wire it operates on is 1 always results in a value of 0; therefore, the operation can be neither reversible nor unitary. The resulting quantum guarded command is of the following form:

$$\square_{i=1}^{n} \bar{q}, |i\rangle \rightarrow P_i.$$

The quantum programs are guarded by the basis states, and the control flow results from the superposition of these basis states [YYF12].

2.2.2 Iteration

Quantum iteration can be implemented either as quantum recursion or quantum loops. While some languages implement loops based on the measurement of qubits or registers [Ying11], the concept of quantum iteration requires the body of the loop to be executed in superposition based on a guard in superposition [YYF12].

While classical iteration takes an operation and repeats it on a classical register for k iterations, quantum iteration is dependent on a value k' in superposition and, correspondingly, returns a quantum register in superposition. Moreover, it is a special case of quantum branching and heavily restricted by the limitations of quantum computers [YVC24].

2.2.3 Limitations

While quantum control flow is often based on the corresponding control flow primitives on classical computers, it is restricted by multiple limitations imposed by quantum computers. Therefore, many control flow primitives that are used in classical programs can either not be used at all or be used in a limited capacity. There are two main limitations for quantum programs. Firstly, all gate-based quantum computers need to adhere to *reversibility*. Secondly, programs need to follow the *synchronization* principle for them to return any useful results [YVC24].

Reversibility

As introduced in Sec. 2.1.3, any sequence of instructions on gate-based quantum computers, excluding measurements, is required to be reversible by definition, as they are all unitary transformations. Therefore, any quantum control flow is also required to adhere to this principle. A resulting limitation, that is not present on classical computers, is that any guards for guarded commands need to be immutable in the commands themselves. For example, if a qubit's state is flipped when its value is 0, the resulting

command will always return a value of 1. When a program returns the same result regardless of which statements were executed, the program cannot be reversible. This limitation is also inherent in the definition of quantum guarded commands, as described in Sec. 2.2.1. Moreover, control flow, as implemented in classical computers, is also not possible. At the most basic software level, modern computers use jump and conditional jump instructions to implement branching and loops. However, any classical jump instruction is inherently irreversible. Not only can a jump go to a section of code that is accessible without any jumps, but multiple jumps can also lead to the same line of code. Therefore, a reversed program cannot know which path was taken in the original program [YVC24].

A simple solution seems to be offered by the *Landauer Embedding* [Land61]. Fundamentally, the idea of the embedding is to turn a non-reversible function into a reversible one by not only returning the output but also the input of the function. For example, for a domain D and a codomain D', any non-reversible function $f : D \to D'$ can be given as a reversible function $g : D \to D' \times D$ with $g(x) = (f(x), x)$. In the case of a quantum program with, e.g., jump instructions based on the Landauer embedding, the output would be the result of the program and a complete history of which path was taken through the program. However, because the quantum data depends on the program history, they become entangled. This leads to disruptive entanglement, as described in Sec. 2.1.2, causing invalid results [YVC24].

Synchronization

As previously discussed, reversibility alone is not the only limiting factor on quantum control flow. When handling control flow, similar to the classical implementation, with a program counter in superposition, the program counter can become entangled with the data and result in disruptive entanglement leading to an invalid result. To avoid this issue, the program must not only be reversible but also adhere to the principle of *synchronization*. It states that control flow must become independent from the data. Further, because any quantum program needs to be synchronized to return any useful results, while loops dependent on a value in superposition need to be bounded by a classical value [YVC24].

2.3 Quantum Languages

With the emergence of quantum computing, many quantum languages were introduced. Most languages focus on a lower-level representation of quantum circuits. An example is the popular Open Quantum Assembly Language (OpenQASM) [CBSG17]. OpenQASM consists mainly of quantum and classical registers that can be manipulated by predefined and composite gates. Additionally, some classical control flow is possible with if-statements depending on classical bits or measurements. As its name suggests, the language is designed for low-level interactions with quantum computers and is mostly used to directly describe a quantum circuit. In Sec. 2.3.2, OpenQASM is discussed in more detail.

In contrast to the low-level circuit descriptions of OpenQASM, there are also languages with a focus on high-level interactions. One such language is Tower [YuCa22]. It does not only allow for basic qubits and registers in superposition but also abstract data structures such as lists. Another example is the language Silq [BBGV20], which allows for the automatic and safe uncomputation of registers after they have been used for, e.g., intermediate calculations. What both languages have in common is the restriction to quantum data while using only classical control flow.

Although quantum control flow was formally defined by Ying et al. [YYF12], as described in Sec. 2.2, over ten years ago, only very few languages have incorporated the principle. One example is the functional programming language proposed by Altenkirch et al. [AlGr05], where quantum branching is used to define, e.g., the Hadamard gate. Only recently was the Quantum Control Machine with quantum control flow at its core proposed by Yuan et al. [YVC24]. It presents an instruction set similar to classical assembly languages but for quantum computers and discusses the resulting limitations for the language. In the following section, we will discuss the quantum control machine in more detail.

2.3.1 Quantum Control Machine

The Quantum Control Machine (QCM), proposed by Yuan et al. [YVC24], is an instruction set architecture that does not only allow for data in superposition but also quantum control flow. The architecture is designed around the limitations of control flow in superposition.

The syntax and logic of the QCM are both heavily influenced by classical assembly languages. Similar to classical computers, the language provides a finite set of quantum registers that are all initialized to a value of zero. The instruction set of the architecture does not only provide limited gate transformations and swap operations but also more classical operations on registers, such as get-bit operations and simple arithmetical operations like addition and multiplication. However, what makes the QCM stand out are the jump instructions that enabled quantum control flow.

The gates of the architecture are limited to the X and Hadamard gate H. However, since the QCM enables quantum branching, any gate can become a controlled gate such that the X gate can easily be used in combination with quantum branching to create a Toffoli gate. Together with the Hadamard gate, the gate set is therefore universal, as described in Sec. 2.1.3.

There are three kinds of jump instructions. The first is a simple jump based on a given offset, the second is a conditional jump that performs a basic jump when a given register is 0, and, lastly, an indirect jump, which is based on the value of a given register. Although the jump instructions are based on jumps in classical computers, they are limited by the restriction of unitary gates and must adhere to *reversibility* and *synchronization* [YVC24], as described in Sec. 2.2. An overview of some QCM instructions is depicted in Tab. 2.2.

[1] After all operations, the instruction pointer is increased by the value of the branch control register.

Operation	Syntax	Semantics[1]
No-op	`nop`	Only increases instruction pointer by the branch control register.
Addition	`add` *ra rb*	Adds register *rb* to *ra*.
Multiplication	`mul` *ra rb*	Multiplies register *ra* by *rb*.
Jump	`jmp` *p*	Increases branch control register by *p*.
Conditional Jumps	`jz` *p ra*	Increases branch control register by *p* if *ra* is 0.
	`jne` *p ra rb*	Increases branch control register by *p* if *ra* is not equal to *rb*.

Table 2.2: An excerpt of the QCM instruction set with instructions used in later examples.

When quantum computers are based on unitary gates, all their operations need to be unitary and, therefore, reversible as well. This limits quantum jump instructions and prohibits them from working like their classical equivalent. However, the problem of a reversible architecture and instruction set is not unique to quantum computers but was also taken into consideration for classical architecture to, e.g., increase the energy efficiency of classical computers [AGY07, TAG12]. To enable reversible jumps, the QCM adapts the *branch control register* from the reversible Bob architecture [TAG12]. Instead of directly changing the instruction pointer of the machine, the branch control register specifies how much the instruction pointer advances after each instruction.

The branch control register can then be manipulated reversibly by, e.g., adding or subtracting from it. To jump by a given *distance*, the branch control register needs to be increased to *distance*. However, after the instruction pointer has reached the desired location, the register needs to be decreased to its original value. Otherwise, the pointer would continue to jump in larger increments, and any further jumps, i.e., modification to the branch control register, would not jump to the correct location. Since the jump instructions are defined to be reversible, the instruction set also includes a reverse jump instruction, which instead decreases the branch control register by a given offset. Therefore, a jump instruction always requires a reverse jump instruction to reset the program counter. Similarly, other operations can also be represented as the reverse operation of an existing one. For example, subtraction can be implemented as reverse addition. Further, to make the code easier to read and write, the QCM also allows for named labels, which can be used for jump instructions instead of offsets. The offset to the given label can then be computed at compile time.

An example of a classical program and the reversible equivalent can be seen in Fig. 2.5 and Fig. 2.6, respectively. Both programs calculate x^y for two registers x and y. While the first example has classical jumps that are not reversible, the second example uses reversible jump instructions and their reverse counterparts to create a reversible algorithm.

Although such a program counter addresses the issue of reversibility, it can become

```
1       add   res $1
2       add   r1  y
3  l1:  jz    l2  r1
4       mul   res x
5       radd  r1  $1
6       jmp   l1
7  l2:  nop
```

Figure 2.5: A non-reversible exponentiation algorithm.

```
1       add   res $1
2       add   r1  y
3  l1:  rjne  rl1 r1  y
4  rl2: jz    l2  r1
5       mul   res x
6       radd  r1  $1
7  rl1: jmp   l1
8  l2:  rjmp  l2
```

Figure 2.6: A reversible exponentiation algorithm.

entangled with data registers when in superposition. This can lead to disruptive entanglement where the output of the program becomes invalid [YVC24]. To prevent any disruptive entanglement of the data and control registers, QCM programs must adhere to the principle of synchronization, as described in Sec. 2.2. It requires that the control flow is separated from the data at the end of execution. However, this is not the case for the reversible example program in Fig. 2.6, which, therefore, is not a valid QCM program.

The issue that occurs in the loop of the reversible example is the *tortoise and hare* problem. Given a superposition of two different values a and b in the y register, the loop will execute a and b times, respectively. Therefore, one of the two loops will finish before the other. Since we must adhere to synchronization, the instruction pointer needs to become independent of the two values again. However, because the branch with the faster execution of the loop cannot simply wait, the other branch cannot catch up, and the instruction pointer cannot become independent of the data values. Consequently, the program does not adhere to synchronization. To prevent this issue, the program must include padding operations that are executed instead of the main loop. Furthermore, the loop also needs to be bounded by a classical value, as described in Sec. 2.2.3. This results in the algorithm that calculates $x^{\min(y,max)}$, as depicted in Fig. 2.7. Here, max is a classical bound to the number of loop iterations, as required.

2.3.2 OpenQASM Language

The Open Quantum Assembly Language (OpenQASM) 3 [CJA*22] is the successor of the OpenQASM 2 [CBSG17] language. Both languages are imperative and machine-independent quantum languages. They are low-level quantum languages and, thereby, concretely describe a quantum algorithm in the form of a circuit. OpenQASM 2 developed into a de facto standard and is often used as an intermediate language for different quantum tools [CJA*22]. OpenQASM 3 was developed to fit the changing needs of current quantum research and hardware while being mostly backwards compatible except for some uncommon cases. For example, some keywords were added or changed for the successor such that identifiers of OpenQASM 2 circuits may be invalid

```
1           add    res   $1
2           add    r1    max
3  l1:      rjne   rl1   r1   max
4  rl2:     jz     l2    r1
5  rl3:     jg     l3    r1   y
6           mul    res   x
7  rl4:     jmp    l4
8  l3:      rjmp   rl3
9           nop                    ; padding
10 l4:      rjle   rl4   r1   y
11          radd   r1    $1
12 rl1:     jmp    l1
13 l2:      rjmp   rl2
```

Figure 2.7: A synchronized, reversible exponentiation algorithm.

in the successor language. Since OpenQASM 3 is the new and improved standard, we will focus on its features in the following section.

OpenQASM 3 includes an optional version string to indicate the language in the circuit header for any top-level circuit. This is achieved by adding "`OPENQASM 3.0;`" to the beginning. Additionally, the language supports the inclusion of other source files, which can be included with the `include` keyword.

Similar to other quantum languages, OpenQASM operates on two basic data types. The first is the classical bit, while the second is the qubit. Both primitives can also be used in registers with a fixed size. Additionally, OpenQASM 3 also supports further classical data types such as angles and signed and unsigned integers. In contrast to its predecessor, where any identifiers have to start with a lowercase letter, in OpenQASM 3, identifiers can start with a range of Unicode characters with some exceptions.

The basic operations of the language can be divided into unitary and non-unitary operations. In OpenQASM 3, all unitary operations are based on the unitary $U(a, b, c)$ where a, b, c are angular parameters. While OpenQASM 2 supported a controlled-not gate natively, the successor does not and includes control modifiers instead. A control modifier can be used to turn any arbitrary unitary gate into a controlled gate with an arbitrary number of control qubits. Therefore, the formerly predefined gate CX must now be defined by the programmer or represented by an X gate with a control modifier, e.g., `ctrl @ x`. Lastly, the non-unitary operations are `measure` and `reset`. While the `measure` operation measures the state of a qubit and saves it to a classical bit, the `reset` operation discards the value of a qubit and replaces it with the $|0\rangle$ state.

The programmer can not only use the operations and modifiers provided by OpenQASM 3 but can also define custom gates. These user-defined gates are defined with an identifier for the gate and a fixed number of single-qubit arguments and angular parameters. In the body of the gate definition, the user can apply a sequence of gates to the qubit arguments with the given angular parameters. Additionally, the language also provides implicit iteration. This means that the application of a single-qubit gate to a quantum register will be interpreted as separate applications of the gate to all

qubits in the register.

```
OPENQASM 3.0;     /* Indicate language in circuit header. */

gate x a { U(pi,0,pi) a; }                /* Define x  gate. */
gate cx a, b { ctrl @ x a, b; }           /* Define cx gate. */
gate h a { U(pi/2, 0, pi) a; }            /* Define h  gate. */

qubit[2] reg;             /* Definition of quantum register. */
bit[2] res;               /* Definition of resical register. */

h reg[0];         /* Apply h gate to fist qubit in register. */
cx reg[0], reg[1];          /* Apply cx gate to the qubits. */

res[0] = measure reg[0]; /* Measure qubit and save to bit. */
res[1] = measure reg[1]; /* Measure qubit and save to bit. */
```

Figure 2.8: An OpenQASM 3 example quantum program.

In Fig. 2.8, an example circuit, written in OpenQASM 3, is depicted. The circuit takes two qubits, brings them into an entangled superposition, measures their state, and saves the result to a classical register. In the beginning of the circuit definition, the circuit header indicates the language, and the X, CX, and H gates are defined based on the predefined unitary U. Then, the quantum and classical registers, both with a size of 2, are defined. Next, the Hadamard gate H is applied to the first qubit in the quantum register, followed with the application of a controlled-not gate to both qubits. Lastly, the state of both qubits is measured and the result is saved to the classical bits.

2.4 Compilation

The execution on a computer is controlled by a program. This program is written in a specific language unique to the hardware of the computer, machine code. However, this language is often neither human-readable nor suitable for writing complex systems. Therefore, most programs are written in a more accessible language. The program can then be translated to machine code with a *compiler*.

A compiler translates a program written in a source language to a program in a target language. The compilation process can be divided into multiple steps. The first step is the *lexical analysis* to transform the source code into a sequence of tokens. Next, the syntactic structure of the code is analyzed by the *parser*. Then, the code is *semantically analyzed* to find semantic errors and infer information for the following phases. Lastly, the *code generation* step generates the code in the target language. Additionally, the compiler may perform optimizations on the code before generating the target code, or it may *optimize* the resulting target code [Oliv07, VSSD07]. In the following, we discuss the different steps of a compiler individually.

2.4.1 Lexer (Lexical Analysis)

The lexical analysis of the source program takes the character stream and groups together associated characters, producing a sequence of tokens [Oliv07]. Therefore, the step is also referred to as *tokenization* [Gref99]. The process can be divided into the *scanning* and *screening* of the character and token sequence [DeRe74].

The scanning process groups together substrings into textual elements, or tokens. In contrast to the characters and substrings, these tokens have defined meanings and may have additional attributes. For example, they may include identifiers, operators, comments, and spaces. In the case of the identifier token, an additional attribute could be the string value of the identifier. They can be specified with the help of a regular grammar or regular expression [DeRe74, VSSD07].

After being divided into a sequence of tokens, the screening step drops any characters or sequences of characters not relevant to the compilation from the program code. These may include characters such as spaces and tabs, or white space in general, and character sequences such as comments. Furthermore, it may also recognize additional special symbols, such as keywords, and map them to a designated token. For example, an identifier with a value of "while" could be mapped to the corresponding `while` token [DeRe74].

Some example regular expressions for a lexical analysis are depicted in Fig. 2.9. The code depicts regular expressions for integers, identifiers, comments, and white space in ANTLR syntax. The integer can either be an arbitrary sequence of characters between zero and nine without a leading zero or just zero with a length of at least one. Similarly, an identifier is a sequence of lower and upper case alphabetical characters, numbers, and underscores with a length of at least one and without a leading number. In contrast, a comment is any string starting with a double slash until the line break, and white space is any white space characters. Additionally, the comment and white space also define a scanning step where both are discarded.

```
1 INTEGER    : [1-9] [0-9]* | '0' ;
2
3 IDENTIFIER : [a-zA-Z_] [a-zA-Z_0-9]*;
4
5 COMMENT    : '//' ~[\r\n]*    -> skip;
6
7 SPACE      : [ \t\r\n\u000C] -> skip;
```

Figure 2.9: An example of a regular grammar for the lexical analysis.

2.4.2 Parser (Syntax Analysis)

The lexical analysis of the compiler yields a sequence of tokens with a known meaning; the structure of the program, however, is not apparent in the token sequence. For example, an operator token does not indicate what the operands are. To gain knowledge

of the structure of the program, the parser step analyzes the syntactic structure of the source program and creates a parse tree from it. The compiler can then use the tree by, e.g., walking over it to generate the target code. This step should also detect and report any syntactical errors, like a missing closing parentheses [VSSD07].

While the lexical analysis can be achieved with regular expressions, the syntactic structure of a program must be represented by, at least, a context-free grammar. Since regular languages are a subset of context-free languages, the parsing step can also perform the lexical analysis. However, there are multiple reasons why the lexical and syntax analyses are separated. Firstly, the separation of both analyses makes the compiler more modular and extensible. Furthermore, using regular expressions for the lexical analysis prevents it from being more complex than necessary with a context-free grammar. Lastly, the lexer can be more efficient when generated from regular expressions instead of a context-free grammar [VSSD07].

There are two main kinds of parsing a grammar, either top-down or bottom-up. Top-down parsing creates a parse tree based on an input sequence of tokens starting from the root and creating the nodes in a depth-first approach. It yields a leftmost derivation for the input sequence and can be implemented as a recursive-descent parser. The most common form of top-down parsing is LL-parsing, where the input is read from *l*eft to right, yielding a *l*eftmost derivation. To improve the efficiency of parsers, the context-free grammar is often restricted such that it can be parsed without backtracking with a fixed length *lookahead* onto the token sequence. Such grammars are called $LL(k)$ grammars, where k is the length of the lookahead [VSSD07, PaFi11].

In contrast, bottom-up parsing builds the parse tree from the leaves up to the root. Furthermore, instead of yielding a leftmost derivation, it produces a rightmost derivation. Similar to top-down parsing, the most common bottom-up parsers scan the input from left to right, which are therefore LR-parsers. Moreover, they can also be implemented more efficiently when restricting the grammar to a maximum lookahead. These grammars are the $LR(k)$ grammars [VSSD07, PaFi11].

An example grammar for parsing simple integer expressions is depicted in Fig. 2.10. Similar to the regular expressions in Fig. 2.9, the grammar is given in ANTLR syntax. An expression is either the sum of another expression and a term or just a term. In turn, a term is either the product of a term and a factor or just a factor. Lastly, a factor is either an expression in parentheses or an integer. Here, the definition of an integer is omitted. However, it can be seen in the previous example. The grammar is defined such that a generated parse tree inherently adheres to the order of operations.

```
1 exp    : exp '+' term    | term;
2
3 term   : term '*' factor | factor;
4
5 factor : '(' exp ')'     | INTEGER;
```

Figure 2.10: An example of a context-free grammar for parsing simple expressions.

2.4.3 Semantic Analysis

The parser analyzes the syntactic structure of a program with a context-free grammar; however, an analysis without any context is not sufficient for an analysis of non-syntactic, i.e., semantic, constraints of the program. This step is performed by the semantic analysis. The semantic analysis is used to throw semantic errors that may prevent the program from being compiled, such as the use of undefined identifiers. Further, it may also enforce constraints that prevent runtime errors, such as type checking in a strongly typed language. Additionally, the analysis step may also process and save declarations and similar information to a symbol table, which can be used in the code generation or optimization [Oliv07, SWW*88]. Moreover, the semantic analysis may not only throw errors but can also be used to infer additional information for further compilation steps. For example, besides preventing operations on operands with invalid types, the analysis may deduce which operation to apply to the operands based on their type; in the case of two integers, the analysis may infer an integer addition for the "+"-operator, while two floating point values require floating point operations [Wait74, VSSD07].

What specifically the semantic analysis does is dependent on the design of the language being analyzed. For example, a loosely typed language may have limited type checking, when compared to a strongly typed language, if any at all. Further, the implementation of the analysis can differ greatly. However, all implementations have some common elements. It requires the propagation of attributes through the syntactic structure of the program to enable the analysis. In the case of type checking, the analysis must pass on the type of a variable. Moreover, it does not only need to know the types of variables and constants, i.e., leaves in a parse tree, but also the resulting type of an expression using them. For example, an integer added to a floating point value may result in a floating point value. To infer and propagate these attributes, the parse tree may need to be traversed [Wait74, VSSD07].

2.4.4 Code Generation

After the semantic analysis of the program, which, at this stage, is in the form of a parse tree, the compiler can generate the code. Here, the compiler can either generate the target code, e.g., machine code, directly or translate the parse tree into an intermediate code. The translation of the source code to the intermediate can be thought of as the *frontend* of the compiler, with the translation of the intermediate to the target being the *backend*. While the intermediate code will need to be translated again into the target language, the use of an intermediate representation can increase the modularity and extensibility of a compiler. Additionally, it can also ease the construction of a new compiler. When creating a new compiler from a source language to a target language, the front end of an existing compiler for the source can be combined with an existing compiler for the target if both are using the same intermediate language [VSSD07, GFH82].

The most common issues when generating the target code are the evaluation order of

expressions, register and storage allocation, as well as related issues, context switches, and instruction selection [GFH82]. While these issues are critical for compilers that translate classical languages, i.e., not quantum languages, to machine code, they are mostly not relevant for the translation of quantum computers, since quantum computers do not offer the same features and abstractions that classical computers do; they have, e.g., no storage, other than the quantum registers. Therefore, we will not discuss these issues in more detail.

2.4.5 Optimization

While the lexical, syntax, and semantic analysis combined with the code generation are the essential parts of a compiler, without which it would not work, the optimization step is also important. It used to apply either machine-independent or machine-dependent optimizations. The optimizations can be applied to the parse tree, a possible intermediate representation, and the generated target code depending on the optimization itself. While the removal of unreachable code, e.g., code after a return statement, can most easily be performed on the parse tree, machine-dependent optimizations can, more appropriately, be performed on the target or intermediate code [Oliv07, VSSD07].

Two collaborating machine-independent optimizations that are often applied by compilers are *constant propagation* and *constant folding*. Constant propagation analyzes the code to find variables with constant values throughout all executions and replaces the variables in, e.g., expressions with their corresponding constant value. By itself, constant propagation may only result in marginal improvement, loading a constant literal instead of the values of a variable; however, in combination with constant folding, it can result in significant improvement. Constant folding evaluates expressions or subexpressions with constant values at compile time, resulting in fewer calculations at runtime. This can significantly increase the performance of a program, especially if large expressions or expressions in loops can be folded. Propagating constant values through the code enables more constant folding and, therefore, can improve its effectiveness [WeZa91].

Another optimization technique that can work in tandem with constant folding and propagation is *loop unrolling*. When executing a loop, each iteration needs to check the halting condition and possibly execute an increment statement, which can result in significant overhead. Furthermore, since the condition is checked before each iteration, the different executions cannot be executed in parallel. To prevent or reduce the performance overhead from these issues, the loop body can be executed multiple times and the increment statement adjusted accordingly. Further, if the number of iterations is constant, the loop can be removed entirely and replaced by the repeating loop body [HuLe99]. In this case, constant propagation and unrolling can help evaluate the halting condition such that the loop can be unrolled.

Similar to loop unrolling, *function inlining* also replaces some part of the code with an equivalent code body to reduce overhead. The inlining of a function replaces a function call with the function body. This mitigates the overhead caused by the function call. Additionally, the inlining also enables or simplifies further optimization such as

constant folding and propagation [TGS22]. However, excessive function inlining may significantly increase the code size and, therefore, have negative effects on the caching of the code and, in turn, on the performance. This effect can be decreased or mitigated by other transformations or code reductions enabled by the inlining [PeMa02].

Lastly, *peephole optimization* is concerned with optimizing inefficient code patterns by analyzing mostly the machine or intermediate code and removing or replacing them. The pattern can be replaced with a reduced number of instructions that have the same effect or instructions that are easier to execute. For example, loading a constant value of zero and executing an addition has no effect and can be removed, while a multiplication by eight can be replaced with a cheaper left shift by three. Furthermore, depending on the language operated on, peephole optimization can also implement some rudimentary constant folding; loading two constants A, B and adding them can be replaced by loading the constant $A + B$. These patterns can be saved in tables and systematically applied to the program code [McKe65, TvS82].

2.4.6 Tools

A compiler is a complex program that can not only require a lot of coding but also is also likely to include errors when written from scratch. While the earlier syntactic stages use general algorithms, writing a custom lexer and parser can require a substantial workload and is prone to errors. Therefore, there exist many tools that can either help create a lexer and parser or generate them entirely [PaFi11, ZLY17]. In the following, we briefly discuss different available compiler generation tools.

Two tools for compiler generation, often used in tandem, are the Fast Lexical Analyzer (*Flex*)[2] and GNU *Bison*[3] [DoSt99]. Flex is a lexer generator, while Bison is a general-purpose parser generator. Both generators target C and C++ code, with Bison having the experimental feature to support Java. Both tools are an extension and improvement of previous tools, *Lex* and *Yacc*, respectively. Bison implements a bottom-up parser and can parse most grammars. However, the tool is optimized for $LR(1)$ grammars, i.e., bottom-up parsing with a lookahead of a single token [ZLY17, Aaby03, DoSt99].

While Flex and Bison are separate tools for the lexing and parsing of program code, *ANTLR*[4] [PaQu95] "ANother Tool for Language Recognition" combines both purposes into a single tool. In contrast to Bison, it implements top-down parsing and can recognize any $LL(k)$ grammar with $k > 1$. Additionally, while Flex and Bison are mainly targeting C and C++, ANTLR can generate lexers and scanners in a variety of languages, including C++, Java, Python, and C#. With its newest version, ANTLR4, the lexing and parsing rules can be given in the form of a context-free grammar with the terminals of the grammar given as regular expressions. At the beginning of each grammar, the name of the grammar is given, and it is indicated whether the grammar describes a lexer, parser, or a general grammar, possibly containing both. The gram-

[2] `https://github.com/westes/flex`
[3] `https://github.com/akimd/bison`
[4] `https://github.com/antlr/antlr4`

mar rules can be either lexer or parser rules. They always begin with a rule name, followed by a colon and the different alternatives, separated by a vertical bar, and terminated with a semicolon. While parsing rule names are given in lowercase letters, lexer rule names must begin with uppercase letters. Additionally, in a grammar rule, any symbol can be given a variable name by first giving the name followed by an equals sign "=" and the corresponding symbol. Later, the variable name can be used when, e.g., traversing the parse tree to more easily access the reference to a symbol in the rule.

```
1 grammar simple_exp;
2
3 expression : INTEGER op=OPERATOR expression
4            | INTEGER;
5
6 OPERATOR   : '+'
7            | '-';
8 INTEGER    : [1-9] [0-9]* | '0' ;
```

Figure 2.11: A simple ANTLR4 grammar for expressions.

A simple ANTLR4 grammar is depicted in Fig. 2.11. Firstly, it indicates that this grammar describes a general lexer-parser combination and gives it the name "simple_exp". Then, a parsing rule called "expression" is defined where the expression is either just an integer or an integer in combination with an operator and another expression. Lastly, the operator and integer lexing rules are defined. The operator is either the "+" or "-" character, and the integer is a regular expression for an arbitrary sequence of number characters without a leading zero. Lastly, in the expression rule, the `OPERATOR` symbol is given the variable name `op`, which can be used to reference the operator later on.

3 Concept

After the discussion of the background of quantum computing in general, quantum control flow and programming language, and the compilation of programs, in this chapter, we present a concept for the quantum programming language "Luie" with control flow primitives, based on the ideas of the Quantum Control Machine. After an informal description, we define both the syntax of the language and translation functions from the source language to the target language OpenQASM. Next, we present the different optimizations that are applied by the compiler. Additionally, we give a theoretical overview of the graphical abstraction that is used to optimize the program. Lastly, we discuss the command line interface of the compiler, its options and their corresponding behavior.

3.1 Language Overview

In the following section, we discuss the different concepts that the proposed language provides for writing quantum programs. Furthermore, we discuss their behavior and some use cases for them. Additionally, we explain why some features of other languages were not incorporated and the reasons for some special behaviors.

The idea for our proposed language is to provide a high-level language with the capabilities of the Quantum Control Machine (QCM). For this, we want to remove low-level concepts, such as jump instructions, and add high-level ones, such as code blocks with variable contexts and additional data types. Additionally, since jump instructions in superposition are removed, we need to add other control flow statements so that the language is as expressive as the QCM. For this, we introduce two basic control flow statements: the loop statement, which is unrolled at compile time, and the quantum if- and else-statements. Based on these ideas, we propose the loop unrolled, if-else, programming language "Luie".

Firstly, we discuss code blocks and their corresponding variable contexts, or scopes. Then, we introduce the data types that are available in the language. Next, the basic operations as well as the more complex control flow statements are listed and explained. This is followed by an overview of the possible expressions and the available operations and built-in functions. Lastly, we discuss the composite gates of the language.

3.1.1 Blocks and Scopes

Similar to many other languages, Luie uses code blocks and corresponding scopes. Both the code blocks and scopes are used to structure the code and enable the reuse of identifiers in different contexts. Code blocks and scopes are hierarchically structured;

the parent of a code block is the block that directly contains this scope, while ancestor blocks are all blocks that contain the block. In contrast, the descendants of a block are all blocks that are directly and indirectly contained in the block. Furthermore, two blocks are independent if one is neither an ancestor nor a descendant of the other. Since each scope corresponds to a code block, the same terms can be used for them.

Each Luie program is contained in the main code block. This main code block can contain arbitrarily many other code blocks. In turn, these code blocks can also contain any number of nested blocks. However, the main code block can only exist once and is the ancestor of all others. Therefore, it also cannot depend on any other block. The main block not only differs from the others in terms of hierarchy, but it is also the only block that can contain composite gate declarations. The composite gates are declared at the top of the main code block and are followed by other declarations or statements.

Similar to the main code block, all other code blocks also consist of declarations and statements. A new code block is defined either in the declaration of a composite gate or in the body of a control flow primitive; this includes the if-statement, else-statement, and loop statement. Additionally, each code block has a unique scope.

Scopes represent the variable context of a code block. In a given scope, the program can access all variables previously defined in this scope and all its ancestors; however, it cannot access the variables of any descendants. In contrast, two independent scopes do not have access to the variables defined in the other scope. Therefore, two independent if-statements can define a variable with the same name in their scope.

3.1.2 Data Types

Luie is mainly focused on being a quantum language with quantum control flow; this is also reflected in its data types. The language operates mostly on two types of quantum data, registers and qubits. Both behave as described in Sec. 2.1, where a register represents an array of qubits with a fixed length. Both registers and qubits are declared similarly. First, the `qubit` keyword is used to indicate the declaration. In the case of a register, the size of the register is given in brackets. Lastly, the identifier for the register or qubit is specified. For data types in most classical languages, the variables need to be explicitly initialized. In quantum computers, however, qubits can only be initialized to $|0\rangle$. Therefore, Luie does not require or allow for an initialization value for qubits and registers but always defaults to $|0\rangle$ for all qubits and register entries.

The second class of data types in Luie are classical data types. They can be divided into constant values, named constants, and loop iterators. In contrast to most classical languages with similar data types, they are not referenced in the compiled program; they are only used at compile time and, therefore, must all have a constant value or a known behavior. For example, while the value of an integer constant is constant, the value of a loop iterator changes with each iteration but is constant in each. Constant values are the most simple data types and represent any value given directly in the program code.

Similar to constant values, named constants, often just called constants, also have

a constant value that is saved as an expression when they are declared and evaluated when the code is generated. However, they can not only be integers but also unsigned integers and floating point values. Furthermore, they can be given by an expression when declared and referenced as often as needed in the context they were declared in. Constants are declared with the `const` keyword and need to be given an identifier as well as the type of the constant. The type is either the `int`, `uint`, or `double` keyword. Lastly, an expression for the value of the constant is given.

The loop iterator data type is declared when creating a loop and is also only accessible in the loop body. Its value corresponds to the current loop iteration, e.g., for the first iteration zero and the last one $n - 1$, where n is the number of iterations. Loop iterators are usually not implemented as a special data type but by incrementing, e.g., an integer; however, Luie does not allow mutable, classical data types and, therefore, cannot increment an integer to iterate over a loop. In turn, it requires a special data type whose value is constant for each iteration.

3.1.3 Basic Operations

Any language needs some basic operations to manipulate its data types. While these operations may be addition or multiplication on classical computers, quantum computers manipulate their data with gates. Luie provides multiple predefined gates with which the qubits and registers can be manipulated. The syntax for applying gates to qubits is similar to other quantum languages; the name of the gate is given, and all arguments are listed next, separated with commas.

The first kind of gate set natively available consists of basic single-qubit gates; these include the X, Y, and Z gates as well as the Hadamard gate H. The logical extension of single-qubit gates is the second gate set, the multi-qubit gates; they include the controlled-not gate CX and the Toffoli gate CCX. These gates are not required for the overall gate set to be universal because they can easily be simulated with the control flow statements available in the language. However, they are included because most programmers are used to them; internally, they are translated to controlled X gates. Finally, the parameterized gates are the last gate set. In contrast to the basic single- and multi-qubit gates, they are not constant in their behavior but depend on one or multiple parameters. These parameters are given as expressions and are evaluated at compile time; they are given in parentheses after the gate and separated by commas. Currently, the only parameterized gate available is the phase gate $P(\lambda)$. The respective behavior and mathematical definition of all predefined gates in Luie are discussed in Sec. 2.1.3. Since the language provides both the Toffoli and Hadamard gates, the overall gate set is universal. Furthermore, any gates that are not predefined can easily be implemented as composite gates; they are described in Sec. 3.1.7.

However, one feature not provided for gate applications is implicit iteration. Implicit iteration allows gates that operate on a single qubit to be applied to an entire register instead by giving the register as an argument. Since Luie provides loop statements with which a gate can explicitly be applied to an entire register with additional control over the application, implicit iteration is not implemented in the language.

3.1.4 Measurements

For a quantum circuit to be of any use, measurements are required. However, there are many options for implementing measurements. Our language only implements implicit measurements and does not allow the explicit measurement of a qubit at an arbitrary point in the circuit. While explicit measurement may allow for more flexibility and manual optimization of qubit usage, both approaches can implement the same set of algorithms. Furthermore, our language focuses mainly on control flow statements, so we decided on a simple approach for the measurement of qubits. Additionally, explicit measurement statements in the language would require further classical data types to store the result of the measurement; this would increase the overall complexity of the language and require the user to add additional declarations for each explicit measurement.

In our language, the measurements are implicitly added at the end of each qubit wire. Thus, the compiler declares both a bit for each qubit and a classical register for each quantum register. Next, each qubit and quantum register is measured, and the result of the measurement is saved to the corresponding bit or classical register.

3.1.5 Control Flow

The main focus of the proposed language is quantum control flow. Therefore, Luie provides two different control flow statements: a loop statement with a fixed number of iterations and an if-statement operating on data in superposition.

The loop statement can be used to iterate over a code block a fixed number of times. While the number of iterations needs to be known at compile time, it can depend on constants known at compile time, such as the size of a register. However, it can, e.g., not depend on the measurement of a qubit or the value of a qubit in superposition. This relates back to the principle of synchronization, as described in Sec. 2.2.3. Additionally, the body of the loop can depend on the iteration index, i.e., the current value of the iterator. Therefore, the loop statement can, e.g., iterate over a register or part of it and apply a series of gates to each qubit. Furthermore, the loop body can also include all statements and declarations available for each code block. In turn, the loop statement can contain nested loops, or parts of its body can be guarded by an if-statement. However, the target language of the compiler and quantum computers in general do not have a concept of local variables. Therefore, while declarations can be performed, a declaration in a loop body is translated to a “global” declaration of a qubit for each iteration. This can significantly increase the qubit count of the program. To optimize the current implementation, a single qubit could be declared that is reset or measured at the end of each iteration, thereby reducing the required number of qubits. However, this optimization is currently not implemented.

In contrast to loop statements, the if-statements do not depend on constant data known at compile time; their execution, however, depends on the value of a qubit in superposition. Further, the guard of the statement can only be a single qubit, not

a register, and cannot include boolean expressions. The body of the statement can contain any other statements, including further if- and loop statements, as well as declarations. When compiled, each gate application inside the body is guarded by the given qubit and translated to a controlled version of the gate where the guard is the control. However, the declaration is not dependent on the value of the guard. Therefore, similar to declarations inside loop statements, the declaration is always performed independent of the guard's value. In turn, the declaration of qubits or registers inside an if-statement can increase the qubit count. However, since it is only performed once and not for each iteration, the gate count is not affected to the same extent as the declaration in loop statements. Additionally, besides the main code block, the if-statement also offers an optional else case. In this case, the translated statements of the else-block are controlled by the negated guard.

3.1.6 Expressions

The language allows for complex expressions that are evaluated at compile time. These expressions can be used to access specific indices of a register or define the range of an iterator used by a loop statement. Besides the typical operations like addition and multiplication, the language also implements different functions. However, the targeted quantum computers should not need to and, in some cases, cannot evaluate these expressions; their values need to be not only constant but also evaluated at compile time. Furthermore, they can neither depend on the measurement nor the value of a qubit.

The first kind of operations are the basic operations; they include addition and subtraction, multiplication and division, and the negation. The grammar of the language is designed such that the usual operator precedences are respected. Additionally, parentheses can be used to adjust this order. The operations operate either on an integer value or an identifier. If an integer value is given, it can easily be parsed to an integer and used with the operations. However, the identifier can either be a constant or an iterator. For either, the compiler looks up the current value in the symbol table and evaluates the remaining expression.

Besides basic operations, the language provides more complex functions that can be used for specific calculations. Firstly, the `sizeof` function takes a register as an input and returns the size of the given register. This can be useful for creating a loop statement that iterates over the size of a register. Furthermore, it can be used to create a register based on the size of another. Next, the `power` function takes two expressions as arguments, evaluates them, and raises the first argument's value to the value of the second argument. Lastly, similar to the `power` function, the `min` and `max` functions both take two expressions as arguments and return the minimum and maximum value of the evaluated expressions, respectively.

Luie provides two different expressions for creating ranges. Since they create ranges, or more specifically loop iterators for a given range, and the other operations only operate on basic numeric types, e.g., integers and floating point values, they are not grouped together with the others but implemented separately. The first way to define a

range is to give the start and end values separated by two dots, e.g., `1..9` for iterating from one to nine. In this case, both the start and end values are included in the iteration. However, this expression can only take integers for the start and end values, not expressions; this limitation is imposed such that the expression remains readable. The other operation for creating ranges is the `range` keyword followed by the start and end values in parenthesis and separated by a comma, e.g., `range`$(1, 9)$. For this expression, the arguments can also be given as expressions. Furthermore, the `range` expression also provides a shorthand, where only the length *len* of the range is given. In this case, it will always start from zero and iterate to $len - 1$. This is most useful for creating ranges to iterate over registers.

Lastly, in contrast to many other languages, the access of a register is not implemented as an expression. A register access, i.e., a qubit, is not a classical data type and cannot be used in any context other than as a gate argument or as an if-statement guard. Therefore, it is implemented as a separate grammar rule and not as an expression.

3.1.7 Composite Gates

Similar to OpenQASM, Luie also allows for the declaration of composite gates. They can be used to declare a custom gate that applies the gate combination specified in the gate body. This can be particularly useful for gate combinations often used in the code to reduce redundancy and, furthermore, improve the readability of the code. For example, the swap gate can be defined on two qubits where three controlled-not gates are applied such that the values of both qubits are swapped. In this case, not only are three gates reduced to only one, but the code clearly indicated that the values are swapped without prior knowledge of the effect of the controlled-not gates.

While the concept of a composite gate is also available in OpenQASM, Luie expands on the possibilities provided by OpenQASM's implementation. Firstly, the gate declaration does not only allow for simple gate application statements but also for the control flow statements provided by Luie. This includes the loop statements and if-statements in superposition; only qubit and register declarations are prohibited in the body of gate declarations. In contrast to composite gates in OpenQASM, not only qubits are allowed as arguments but also registers. Therefore, registers can be given as arguments, and the loop statement can iterate over them, resulting in gates that can depend on the length of a register and do not need to be reimplemented for each different size. However, in contrast to OpenQASM, the language currently does not allow for parameterized composite gates.

Two examples of composite gates are depicted in Fig. 3.1. The first gate is a simple definition for the commonly used, and often predefined, `swap` gate. Below, a gate definition for the quantum Fourier transform is given. It not only references the previous swap gate but also uses both if- and loop statements. Firstly, the gate iterates over all qubits in the given register *reg* with size n. In each iteration, the gate first applies the Hadamard gate to the qubits and, again, iterates over all remaining qubits in the register and applies the phase gate depending on the value of the qubit in

```
1  // Swaps the values of two qubits
2  gate swap(a, b) do
3      cx a, b;
4      cx b, a;
5      cx a, b;
6  end
7
8  // Performs a discrete Fourier transform on a register of qubits
9  gate qft(reg) do
10     for i in range(sizeof(reg)) do
11         h reg[i];
12         for j in range(sizeof(reg) - (i + 1)) do
13             qif reg[j + (i + 1)] do
14                 p(1/(power(2, (j + 1)))) reg[i];
15             end
16         end
17     end
18     for j in range(sizeof(reg) / 2) do
19         swap reg[j], reg[sizeof(reg) - (j + 1)];
20     end
21 end
```

Figure 3.1: A Luie gate definition for the Quantum Fourier Transform.

superposition. Additionally, the parameter for the phase gate is calculated based on the offset of both indices. After iterating through the entire register and applying the corresponding gate, the gate swaps each qubit at index i with the qubit at index $n-i$ for $0 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor$.

3.2 Syntax

The syntax of our programming language *Luie* is defined by the context-free grammar CFG_{Luie}. It consists of a set of non-terminals V_{Luie}, terminals Σ_{Luie}, and grammar rules R_{Luie}, as well as the start symbol prg_{Luie}. The rule n represents a natural number, while id represents an identifier. The grammar rules R_{Luie} are discussed in the following.

$$
\begin{aligned}
CFG_{Luie} &= (V_{Luie}, \Sigma_{Luie}, R_{Luie}, prg_{Luie}) \\
V_{Luie} &= \{exp, rExp, gate, qArg, stm, \\
&\qquad dcl, gDcl, blk, t, prg_{Luie}\} \\
\Sigma_{Luie} &= \{\texttt{..}, \texttt{range}, \texttt{(}, \texttt{)}, \texttt{+}, \texttt{-}, \\
&\qquad n, id, \texttt{[}, \texttt{]}, \dots\} \qquad \texttt{where } n \in \mathbb{N}_0, id \in Identifier
\end{aligned}
$$

However, the following definitions, including the translation functions in Sec. 3.3, exclude some minor language features for simplicity; these are the application of param-

eterized gates and the different types of named constants, such as unsigned integers and doubles.

The grammar contains two different kinds of expression rules, the arithmetic expression rule *exp* and the range expression rule *rExp*.

$$\textit{Expression} : exp ::= n \mid id \mid exp_1 \texttt{ + } exp_2 \mid exp_1 \texttt{ - } exp_2 \mid exp_1 \texttt{ * } exp_2 \mid \dots$$
$$\textit{RangeExpression} : rExp ::= n_1..n_2 \mid \texttt{range(}exp\texttt{)} \mid \texttt{range(}exp_1, exp_2\texttt{)}$$

Here, the first range expression variant with two dots only allows for natural numbers, including zero. Since we want to avoid overly complex range expressions for this variant, we do not allow for expressions and use the number symbol instead, which does not allow for negative integers. However, if ranges with negative start or end indices are required, the last expression variant can be used.

One grammar rule that is used in the context of gate applications is the qubit argument rule, *qArg*. It is used for the arguments in gate applications and allows for either a qubit, specified by an identifier, or a register access, specified by an identifier and expression. Additionally, we define a set of predefined gates, *ConstGates*, that contains all gate identifiers with predefined semantics; it is used in the translation functions to differentiate between predefined and composite gates.

$$\textit{QubitArgument} : qArg ::= id \mid id[exp]$$
$$\textit{ConstGates} = \{\texttt{h}, \texttt{x}, \texttt{y}, \texttt{z}, \texttt{cx}, \texttt{ccx}\}$$

Next, the declaration rules *dcl* and *gDcl* are used to declare qubits, registers, or named constants and gate declarations, respectively. Each requires an identifier given by the *id* rule. In the case of the constant and register declaration, the value and size are given by an expression. Additionally, the gate declaration requires a list of identifiers for the arguments and the code block of the composite gate.

$$\begin{aligned}\textit{Declaration} : dcl ::= {} & \texttt{const } id \texttt{ = } exp\texttt{;} \mid \\ & \texttt{qubit } id\texttt{;} \mid \\ & \texttt{qubit[}exp\texttt{] } id\texttt{;}\end{aligned}$$
$$\textit{GateDeclaration} : gDcl ::= \texttt{gate } id \texttt{ (}id_1, \dots, id_n\texttt{) do } blk \texttt{ end}$$

The statement rule *stm* can either be an if-statement, with an optional else-block; a loop statement; a gate application statement; or a skip statement. The qubit argument rule is used for both the if-statement, to specify the control qubit, and the gate application statement, to specify the arguments. Additionally, the range expression is used to specify the range of the loop statement. The block rule is used for the body of the if-statement, the optional else-block, and the loop statement.

$$\begin{aligned}\textit{Statement} : stm ::= {} & \texttt{qif } qArg \texttt{ do } blk \texttt{ end} \mid \\ & \texttt{qif } qArg \texttt{ do } blk_1 \texttt{ else } blk_2 \texttt{ end} \mid \\ & \texttt{for } id \texttt{ in } rExp \texttt{ do } blk \texttt{ end} \mid \\ & id\ qArg_1, \dots, qArg_n\texttt{;} \mid \\ & \texttt{skip;}\end{aligned}$$

The last grammar rules are the block rule *blk*, which specifies a possibly empty list of translatables; the translatable rule t, which can either be a statement or declaration; and the program rule prg_{Luie}, which consists of an optional list of gate declarations and a code block.

$$\begin{aligned} Block &: blk ::= t_1 \dots t_n \mid \epsilon \\ Translatable &: t ::= stm \mid dcl \\ Program &: prg_{Luie} ::= gDcl_1 \dots gDcl_n \; blk \mid blk \end{aligned}$$

Since we define some auxiliary functions that operate on OpenQASM code, a reduced grammar for the OpenQASM programming language is given by CFG_{QASM}. It consists of a set of non-terminals V_{QASM}, terminals Σ_{QASM}, and grammar rules R_{QASM}, as well as the start symbol prg_{QASM}. Some of the terminals and non-terminals in the grammar are the same as in the source code grammar CFG_{Luie}. In these cases, the definitions of the grammar rules are also the same; all other grammar rules are discussed in the following.

$$\begin{aligned} CFG_{QASM} &= (V_{QASM}, \Sigma_{QASM}, R_{QASM}, prg_{QASM}) \\ V_{QASM} &= \{qArg, gateApp, qubitDcl, qStm, \\ &\quad\; qasm, prg_{QASM}\} \\ \Sigma_{QASM} &= \{\texttt{ctrl}, \texttt{negctrl}, \texttt{(}, \texttt{)}, n, id, \\ &\quad\; \texttt{x}, \texttt{y}, \texttt{z}, \texttt{cx}, \texttt{ccx}, \texttt{[}, \texttt{]}, \dots\} \qquad \texttt{where } n \in \mathbb{N}_0, id \in Identifier \end{aligned}$$

A qubit or register is declared by the qubit declaration rule *qubitDcl*. It requires an identifier for the symbol and, in the case of a register, a size in the form of a constant natural number.

$$\begin{aligned} QubitDeclaration : qubitDcl ::= \; & \texttt{qubit } id\texttt{;} \mid \\ & \texttt{qubit[}n\texttt{] } id\texttt{;} \end{aligned}$$

A gate is applied with the gate application rule *gateApp*. There are four different kinds of gate applications: one with no control, two with either arbitrarily many positive or negative control qubits, and one with both arbitrarily positive and negative control qubits. The number of control qubits is given as a natural number.

$$\begin{aligned} GateApplication : gateApp ::= \; & id\ qArg_1, ..., qArg_n\texttt{;} \mid \\ & \texttt{ctrl(}n\texttt{) @ } id\ qArg_1, ..., qArg_{n'}\texttt{;} \mid \\ & \texttt{negctrl(}n\texttt{) @ } id\ qArg_1, ..., qArg_{n'}\texttt{;} \mid \\ & \texttt{ctrl(}n_1\texttt{) @ negctrl(}n_2\texttt{) @ } id\ qArg_1, ..., qArg_{n'}\texttt{;} \end{aligned}$$

Lastly, in our reduced grammar, an OpenQASM program consists of a header and

a list of statements that can be either a gate application or gate declaration.

$$
\begin{aligned}
QASMStatement : qStm &::= gateApp \mid qubitDcl \mid \epsilon \\
QASM : qasm &::= qStm_1 \dots qStm_n \\
QASMProgram : prg_{QASM} &::= \texttt{OPENQASM 3.0;} \\
&\quad \texttt{include "stdgates.inc";} \\
&\quad qasm
\end{aligned}
$$

3.3 Translation

In the following, we discuss a formal translation of the source code, given in the form of a Luie program prg_{Luie}, to the target code, in the form of an OpenQASM program prg_{QASM}. An example translation process is given in Sec. 4.4.4.

An important part of the translation is the symbol table. It is used to propagate symbol information throughout the translation. A symbol table is a function that maps an identifier to the information of the corresponding symbol.

$$
\begin{aligned}
SymbolTable := \ \{st \mid st : Identifier \dashrightarrow & (\{\texttt{const}\} \times \mathbb{Q}) \\
& \cup (\{\texttt{qubit}\} \times \mathbb{N} \times Identifier) \\
& \cup (\{\texttt{arg}\} \times QubitArgument) \\
& \cup (\{\texttt{gate}\} \times Block \times Identifier^{+})\}
\end{aligned}
$$

The symbol information, contained in the symbol tables, varies depending on the four different kinds of symbols. The first is the constant symbol, which is identified by the `const` keyword, together with the value of the constant symbol given by a real number. Next, the `qubit` keyword identifies the symbol information of a register; it contains the size of the register as a natural number, where a size of one indicates a qubit, and the unique identifier for the register. Next, the argument symbol information saves the qubit argument to which a composite gate argument maps; it is identified by the `arg` keyword. Lastly, identified by the `gate` keyword, the gate symbol information saves the code block of a composite gate as well as a list of argument identifiers with at least one element.

At the root of the translation is the translation function $trans$. It maps a source code program prg_{Luie} to the corresponding OpenQASM program prg_{QASM}. Firstly, the header information for the program is added. Next, the initial symbol table st_ϵ, with no mappings, is updated with the gate declaration information. Lastly, the code block is translated.

$$
\begin{aligned}
trans : Program &\dashrightarrow QASMProgam \\
trans(gDcl_1 \dots gDcl_n \ blk) = &\ \texttt{OPENQASM 3.0;} \\
&\ \texttt{include "stdgates.inc";} \\
&\ bt(blk, update(update(update(st_\epsilon, gDcl_1), ...), gDcl_n))
\end{aligned}
$$

To translate code blocks, the block translation function *bt* is used. It maps a block *blk* and a symbol table *st* to the OpenQASM translation of the code block. Since a block consists of a list of statements and declarations, they are translated individually. However, the translation of a declaration may adjust the symbol table. Therefore, the translation of a translatable, i.e., either a statement of declaration, returns not just the translation but also a, potentially updated, symbol table. This symbol table is used for the next translation. Additionally, if the block is empty, the function simply returns an empty result.

$$
\begin{aligned}
bt &: Block \times SymbolTable \dashrightarrow QASM \\
bt(t_1 \dots t_n, st_1) &= tr_1 \quad \text{where } (tr_1, st_2) = tt(t_1, st_1) \\
&\quad\; tr_2 \quad \text{where } (tr_2, st_3) = tt(t_2, st_2) \\
&\quad\; \dots \\
&\quad\; tr_{n-1} \quad \text{where } (tr_{n-1}, st_n) = tt(t_{n-1}, st_{n-1}) \\
&\quad\; tr_n \quad \text{where } (tr_n, \_) = tt(t_n, st_n) \\
bt(\epsilon, st) &= \epsilon
\end{aligned}
$$

The translatable translation function *tt* can translate both declarations and statements. Since declarations may adjust the symbol table, it returns not just the translation but also a symbol table. To translate the statements and declarations, it calls the corresponding translation functions, *ct* and *dt*, respectively.

$$
\begin{aligned}
tt &: Translatable \times SymbolTable \dashrightarrow QASM \times SymbolTable \\
tt(t, st) &= \begin{cases} dt(t, st) & \text{if } t \in Declarations \\ (ct(t, st), st) & \text{otherwise} \end{cases}
\end{aligned}
$$

The declaration translation *dt* returns a possible translation of the given declaration and an updated symbol table. In the case of constant declarations, the symbols are only used at compile time, and, therefore, only the symbol table is updated, and an empty result is returned. In contrast, qubit and register declarations need to be specified in the target program. In turn, they are translated. While the syntax is quite similar in both languages, the translation needs to ensure the uniqueness of identifiers and evaluate the expression that gives the size of the register. A unique identifier is

generated when updating the symbol table with a qubit or register identifier.

$$
\begin{aligned}
dt &: Declaration \times SymbolTable \dashrightarrow QASM \times SymbolTable \\
dt(\underbrace{\texttt{qubit}\ id\texttt{;}}_{decl}, st) &= (\texttt{qubit}\ uid\texttt{;}, st') \\
&\quad \text{where } st' = update(decl, st) \text{ and } st'[id] = (\texttt{qubit}, 1, uid) \\
dt(\underbrace{\texttt{qubit[}exp\texttt{]}\ \texttt{q;}}_{decl}, st) &= (\texttt{qubit[}index\texttt{]}\ uid\texttt{;}, st') \\
&\quad \text{where } st' = update(decl, st) \text{ and } st'[id] = (\texttt{qubit}, n, uid) \\
&\quad \text{and } index = at(exp, st) \text{ and } index > 0 \\
dt(\underbrace{\texttt{const}\ id\ \texttt{=}\ exp\texttt{;}}_{decl}, st) &= (\epsilon, update(decl, st))
\end{aligned}
$$

The update function *update* is used to update the symbol table with symbol information from a declaration. In turn, it maps a declaration and symbol table to a symbol table. Additionally, all updates to the symbol table are only performed if the identifier from the declaration neither has a corresponding mapping already in the symbol table nor is an identifier corresponding to a constant gate.

$$update : Declaration \times SymbolTable \dashrightarrow SymbolTable$$

In the case of a constant declaration, its identifier maps to the `const` keyword and the evaluation of the expression given in the declaration. For the gate declaration, the symbol table maps the identifier to the `gate` keyword, the block of the gate declaration, and a list of identifiers that represent the arguments to the gate.

$$
\begin{aligned}
update(\texttt{const}\ id\ \texttt{=}\ exp\texttt{;}, st) &= st[id \mapsto (\texttt{const}, at(exp, st))] \\
update(\texttt{gate}\ id\texttt{(}id_1, \ldots, id_n\texttt{)}\ \texttt{do}\ blk\ \texttt{end}, st) &= st[id \mapsto (\texttt{gate}, blk, id_1, \ldots, id_n)]
\end{aligned}
$$

In the case of the qubit and register declaration, the update function is more complex. Since our language allows variables in independent scopes to have the same identifiers, we need to ensure the uniqueness of identifiers in the target code. For this, a unique identifier *uid* is generated. The rest of the update is similar to the previous cases, and symbol information, such as the size and unique identifier of a qubit of register, is saved to the symbol table.

$$
\begin{aligned}
update(\texttt{qubit}\ id\texttt{;}, st) &= st[id \mapsto (\texttt{qubit}, 1, \texttt{id\_}uid)] \\
&\quad \text{where } uid \text{ is unique identifier} \\
update(\texttt{qubit[}exp\texttt{]}\ id\texttt{;}, st) &= st[id \mapsto (\texttt{qubit}, at(exp, st), \texttt{id\_}uid)] \\
&\quad \text{where } uid \text{ is unique identifier}
\end{aligned}
$$

The arithmetic translation *at* is used to evaluate expressions in the source code to constant real values. While a constant value just evaluates to itself and an identifier to

its constant value, the operations evaluate to the operation applied to the evaluation of the subexpressions.

$$
\begin{aligned}
at &: Expression \times SymbolTable \dashrightarrow \mathbb{Q} \\
at(n, st) &= n \\
at(id, st) &= val \quad \text{if } st[id] = (\texttt{const}, val) \\
at(exp_1 + exp_2, st) &= at(exp_1, st) + at(exp_2, st) \\
at(exp_1 - exp_2, st) &= at(exp_1, st) - at(exp_2, st) \\
at(exp_1 * exp_2, st) &= at(exp_1, st) * at(exp_2, st) \\
at(exp_1 / exp_2, st) &= at(exp_1, st) / at(exp_2, st) \\
at((exp), st) &= at(exp, st) \\
at(-exp, st) &= -at(exp, st) \\
at(\texttt{sizeof(}id\texttt{)}) &= n && \text{if } st[id] = (\texttt{qubit}, n, uid) \\
at(\texttt{sizeof(}id\texttt{)}) &= at(\texttt{sizeof(}qArg\texttt{)}) && \text{if } st[id] = (\texttt{arg}, qArg)
\end{aligned}
$$

The statements are translated with the statement translation function ct. It maps a statement and symbol table to the corresponding translation. Firstly, the skip statement is translated to an empty result.

$$
\begin{aligned}
ct &: Statement \times SymbolTable \dashrightarrow QASM \\
ct(\texttt{skip;}, st) &= \epsilon
\end{aligned}
$$

Next, the translation of a gate application is divided into the application of a constant and composite gate. In the case of a constant gate application, the same gate can be used in the translation, and only the qubit arguments need to be translated to their translated counterparts. For this, the qubit translation function qt is used.

$$
\begin{aligned}
ct(id\ qArg_1, \ldots, qArg_n\texttt{;}, st) = id\ qt(qArg_1, st), \ldots, qt(qArg_2, st); \\
\text{if } id \in ConstGates
\end{aligned}
$$

To translate the application of a composite gate, the corresponding code block is translated. Additionally, an empty symbol table is used in the translation and initialized with mappings from the argument identifiers of the gate symbol to the qubit translation of the given qubit arguments.

$$
\begin{aligned}
ct(id\ qArg_1, \ldots, qArg_n\texttt{;}, st) = bt(blk, st_\epsilon[id_1 \mapsto (\texttt{arg}, q_1), \ldots, id_n \mapsto (\texttt{arg}, q_n)]) \\
\text{if } id \notin ConstGates \\
\text{where } q_i = qt(qArg_i, st), i \in [1, \ldots n] \\
\text{and } st[id] = (\texttt{gate}, blk, id_1, \ldots, id_n)
\end{aligned}
$$

The qubit translation function qt is used to translate the identifier of a gate argument or control qubit, in the case of if-statements, to the corresponding unique identifier

regardless of whether the qubit argument is a qubit, register access, or argument identifier of a composite gate.

$$
\begin{aligned}
qt : QubitArgument \times SymbolTable \dashrightarrow QubitArgument \\
qt(id, st) &= uid && \text{if } st[id] = (\texttt{qubit}, 1, uid) \\
qt(id[exp], st) &= uid\texttt{[}\underbrace{at(exp, st)}_{n}\texttt{]} && \text{if } st[id] = (\texttt{qubit}, m, uid) \text{ and } m > n \\
qt(id, st) &= qArg && \text{if } st[id] = (\texttt{arg}, qArg) \\
qt(id[exp], st) &= uid\texttt{[}at(exp, st)\texttt{]} && \text{if } st[id] = (\texttt{arg}, qArg) \text{ and } qArg = uid
\end{aligned}
$$

The next statement translation case is the loop statement. It is translated by evaluating the given range expression with the range expression evaluation function rt to get the range $(start, end)$. Next, the loop body is translated $end - start$ times, and each time the symbol table is updated such that the loop iterator identifier maps to the current iteration value.

$$
\begin{aligned}
ct(\texttt{for } id \texttt{ in } rExp \texttt{ do } blk \texttt{ end}, st) = \; & bt(blk, st[id \mapsto (\texttt{const}, start)]) \\
& bt(blk, st[id \mapsto (\texttt{const}, start + 1)]) \\
& \dots \\
& bt(blk, st[id \mapsto (\texttt{const}, end - 1)]) \\
& bt(blk, st[id \mapsto (\texttt{const}, end)]) \\
& \text{where } (start, end) = rt(rExp, st)
\end{aligned}
$$

The range expression evaluation function simply differentiates between the three different possibilities of defining a range and returns the range as a tuple of whole numbers.

$$
\begin{aligned}
rt : rExp \times SymbolTable \to \mathbb{Z} \times \mathbb{Z} \\
rt(n_1..n_2, st) &= (n_1, n_2) \\
rt(\texttt{range(}exp\texttt{)}, st) &= (0, at(exp, st) - 1) \\
rt(\texttt{range(}exp_1, exp_2\texttt{)}, st) &= (\lfloor at(exp_1, st) \rfloor, \lfloor at(exp_2, st) \rfloor)
\end{aligned}
$$

The last translation is the translation of if-statements. The if-statement is translated by adding the given qubit argument as a control qubit to all gate applications in the translated code. To achieve this, the code block is translated, and the *control* function, together with the translated qubit argument, is applied to the translation. In the case of the optional else-block, the *nControl* function is used to add the qubit argument as a negative control to the translated gate applications.

$$
\begin{aligned}
ct(\texttt{qif } qArg \texttt{ do } blk \texttt{ end}, st) &= control(qt(qArg, st), kt(blk, st)) \\
ct(\texttt{qif } qArg \texttt{ do } blk_1 \texttt{ else } blk_2 \texttt{ end}, st) &= control(qt(qArg, st), kt(blk_1, st)) \\
& \quad nControl(qt(qArg, st), kt(blk_2, st))
\end{aligned}
$$

The *control* function maps a qubit argument $qArg$ together with QASM code to a controlled version of the given code where $qArg$ is a control qubit for all gate applications

in the program. In the case of a list of quantum statements, the function is applied to each statement separately. If the statement is a qubit declaration, the declaration is simply returned without changes.

$$
\begin{aligned}
control &: QubitArgument \times QASM \to QASM \\
control(qArg, qStm_1 \dots qStm_n) &= control(qArg, qStm_1) \\
&\quad \dots \\
&\quad control(qArg, qStm_n) \\
control(qArg, qubitDcl) &= qubitDcl
\end{aligned}
$$

If a gate application is given, the qubit argument is prepended to the gate arguments, and a control modifier is either added or adjusted to the new number of control qubits.

$$
\begin{aligned}
control(qArg, & & & id\ qArg_1, \dots, qArg_{n'};) = \\
& \texttt{ctrl(1)} & & \texttt{@}\ id\ qArg, qArg_1, \dots, qArg_{n'}; \\
control(qArg, & & \texttt{negctrl}(n) & \texttt{@}\ id\ qArg_1, \dots, qArg_{n'};) = \\
& \texttt{ctrl(1)} & \texttt{@ negctrl}(n) & \texttt{@}\ id\ qArg, qArg_1, \dots, qArg_{n'}; \\
control(qArg, & \texttt{ctrl}(n) & & \texttt{@}\ id\ qArg_1, \dots, qArg_{n'};) = \\
& \texttt{ctrl}(n+1) & & \texttt{@}\ id\ qArg, qArg_1, \dots, qArg_{n'}; \\
control(qArg, & \texttt{ctrl}(n_1) & \texttt{@ negctrl}(n_2) & \texttt{@}\ id\ qArg_1, \dots, qArg_{n'};) = \\
& \texttt{ctrl}(n_1+1) & \texttt{@ negctrl}(n_2) & \texttt{@}\ id\ qArg, qArg_1, \dots, qArg_{n'};
\end{aligned}
$$

Correspondingly, the *nControl* function exhibits the same behavior, but the qubit argument $gArg$ represents a negated control qubit; its definition can be found in Appendix A. Since we place the negative control as the second modifier, the *nControl* function cannot simply prepend the qubit argument $qArg$. However, it is inserted after the first i arguments, where i is the number of positive control qubits given by the `ctrl` modifier.

3.4 Optimization

Our compiler does not only translate the source language Luie to the target language OpenQASM but can also apply optimizations to the program. Our programming language contains multiple language constructs that cannot be directly translated to quantum computers. For example, as quantum computers can only apply gates and have no constructs for loops or jump instructions, the loop statement cannot be translated to an equivalent construct. Therefore, it needs to be reduced to available concepts; in this case, these are only gate statements. The reduction from the loop statements to only basic constructs mirrors code optimizations applied by classical compilers. We call these optimizations applied by the translation, instead of the optimization, *inherent optimizations*. The following, we discuss both the inherent optimizations as well as the optimizations applied by the optimization stage of the compiler.

The first kind of inherent optimizations are constant propagation and constant folding. While the target language does allow for expressions, they need to be constant. Furthermore, some language features that depend on expressions have no equivalent in the target language. In turn, they, and their corresponding expressions, need to be evaluated at compile time. This is the case for, e.g., the control flow statements. Therefore, any variable is always constant, and its value is propagated to be used in the evaluation of expressions. Further, each expression is evaluated at compile time such that constant propagation and constant folding are inherently applied.

Secondly, loop unrolling is always applied to all loop statements. Quantum computers in general cannot provide the ability to iterate over gates since they operate on static circuits. While newer versions of OpenQASM provide loop statements, we decided against their use to allow for more flexible circuit optimizations. Additionally, a reduced set of instructions, namely register declarations and gate applications, allows for easier extension of the compiler to support other target languages. Therefore, to allow for loop statements in our language, the loop is unrolled entirely at compile time.

The last inherent optimization is function inlining. Our language provides the ability to define custom gates that consist of an arbitrary combination of gate applications and, possibly, control flow statements. While OpenQASM has a similar functionality, the composite gates OpenQASM provides are more restrictive; for example, they do not allow for registers as arguments to the gate. Furthermore, in contrast to classical computers, quantum computers do not natively support function calls or related concepts. Therefore, each time a composite gate is used, the compiler inlines the gate body at the location where the gate is applied.

3.4.1 Optimization Rules

After the code is translated, the compiler can perform additional peephole optimizations; these are not inherent to the translation of the program and, therefore, are optional. The peephole optimizations can be applied to the internal representation of the source language, and most hardware-independent optimizations are usually applied to the intermediate representation instead of the target language. However, the peephole optimization rules only operate on sequences of gate applications such that control flow statements or composite gates may only hinder, not aid, the effectiveness of the optimization. The overall performance of the optimization may be increased by optimizing composite gates before inlining their code so that a gate that is called ten times only needs to be optimized once; nevertheless, inlining the gates before applying the rules may enable more optimizations and, thereby, increase the effectiveness at the cost of performance. Therefore, we apply the optimization rules after the code is translated. Additionally, the user can not only specify whether optimizations are applied but also which to apply when using the compiler. In the following section, we discuss the different peephole optimization rules. Additionally, while some optimizations are referred to by the terms commonly used in literature, as described in Sec. 2.1.6, the others without any naming conventions are given descriptive names.

The first kind of optimization rule is the *null gate* optimization; it describes sequences of gate applications such that the resulting behavior is equivalent to applying the identity gate. In the case of classical computers, an example is the sequential execution of two negation operations. In contrast, a quantum example is the application of two successive Hadamard gates. While these optimizations can easily be performed by the programmer themselves for a simple list of instructions, the manual optimizations increase in complexity when using composite gates and control flow statements. Moreover, the programmer cannot remove two null gates by hand that are contained in two different successive composite gates. Therefore, the optimization rules can not only help to reduce the workload of the programmer but also apply optimization rules that cannot be implemented without major changes to the program.

Next, the *peeping control* optimization rule also belongs to the null gate rules. However, its implementation requires some additional evaluations, and, therefore, we separate it from the other null gate rules. The peeping control rule can remove a controlled gate from the circuit if the value of the control wire is $|0\rangle$ at the position of the gate. To estimate the value of the control wire, the implementation needs to iterate over all previous gates on the wire. Therefore, while it is still a null gate, its implementation differs greatly from the other null gates. Furthermore, we implement an additional optimization that removes the control from the gate if the value of the control wire is known to be $|1\rangle$. While the rule does not reduce the gate count of the circuit, it may enable further optimizations on the control wire. For example, two X gates on the control wire can be separated by a controlled-not gate. If the value of the control is $|1\rangle$ when the controlled-not gate is applied, the control can be removed, and the now successive X gates can be removed with a null gate optimization. Since this optimization does not remove the gate from the circuit, it is not a null gate optimization.

Another optimization rule that the compiler implements is the *Hadamard reduction* rule. It implements the matrix equivalences for both the X and Z gates being surrounded by Hadamard gates, $HXH = Z$ and $HZH = X$. Thereby, the rule reduces the gate count of the circuit if the optimization rule is applied. However, the optimization rule does not remove the gate combination but replaces it with another; therefore, the rule is not a null gate optimization. Similar to the null gate optimizations, the rules themselves are not hard to apply by hand. However, in combination with the more complex statements available in the language, the application of the rule is not trivial, and the compiler can optimize parts of the circuit that would have required major changes to optimize them manually.

Lastly, the *control reversal* optimization rule optimizes a controlled-not gate that is surrounded by Hadamard gates on both the control and target wires. When applied, the four Hadamard gates are removed, and the control and target qubits of the gate are switched. Therefore, the gate count is reduced by four gates. As described in Sec. 2.1.6, the optimization is based on two Hadamard reductions and the control reversal of the controlled-Z gate. However, the reversal of the controlled-Z gate does not have any direct gain when applied; it can only enable other optimizations. Furthermore, the application of the controlled-Z reversal may also disable other optimizations. Therefore, an optimization algorithm using the rule would need to either

test both possibilities or estimate the value of the application and, in turn, increase the complexity of the optimization algorithm significantly. Because of this, our optimizations do not include the controlled-Z reversal but implement only the special use case of the rule, the control reversal optimization rule.

3.4.2 Circuit Graph

While it is possible to directly apply optimizations to the program code or internal representation of the code, this approach can be tedious and error-prone. For example, the easiest approach would be to iterate over the code and search for code sequences with more efficient but equivalent alternatives, similar to the peephole optimization patterns on classical computers presented in Sec. 2.4.5. However, two consecutive gates operating on a single wire may be separated by multiple gate applications on different wires in the programmatic description. Therefore, many simple optimization rules may not be applied when using a simplistic algorithm. A more complex approach would be to subdivide the program into lists of gate applications where the wires being operated on by each list are disjunct. While this approach can result in the application of more optimization rules, it will also miss possible applications and already requires a complex implementation. Furthermore, improving on this method only increases its complexity and, in turn, makes it more prone to errors and generally tedious to work with and debug. Therefore, the language does not directly apply the optimizations to the program but uses a circuit graph description, based on the graph described by Kreppel et al. [KMO*23], to apply the optimizations.

The circuit graph C is a graphical description of a quantum circuit; it is an acyclic and directed graph. Furthermore, it is an extension of the classical graph definition. Therefore, besides the set of nodes V and edges E, it includes a set of qubits Q associated with the graph. Furthermore, it contains a function Q_V that maps input and output nodes to their corresponding qubits, as well as a function Q_E that maps edges to their qubits. Additionally, the set of nodes V consists of the set of input nodes I, output nodes O, and gate nodes G.

$$
\begin{aligned}
C &= (V, E, Q, Q_E, Q_V) \\
V &= \underbrace{I}_{\text{Input Nodes}} \cup \underbrace{O}_{\text{Output Nodes}} \cup \underbrace{G}_{\text{Gate Nodes}} \\
E &\subseteq \{(x, y) \mid x, y \in V \wedge x \neq y\} \\
Q_V &: I \cup O \to Q \\
Q_E &: E \to Q
\end{aligned}
$$

For each qubit in the circuit, there exists both an input node and an output node, i.e., the function Q_V is surjective. Additionally, each input-output node pair is assigned exactly one qubit. Furthermore, input nodes do not have an incoming edge, while output nodes do not have an outgoing edge. Moreover, each input node has exactly one outgoing edge, while each output node has exactly one incoming edge; the qubits

assigned to the outgoing or incoming edges are the same as the qubits assigned to the corresponding input or output nodes.

$$\begin{aligned}
\forall q \in Q : &(\exists_{=1} i \in I \text{ such that } Q_V(i) = q) \wedge \\
&(\exists_{=1} o \in O \text{ such that } Q_V(o) = q) \\
\forall i \in I : &\nexists v \in V \text{ such that } (v, i) \in E \\
&(\exists_{=1} v \in (O \cup G) \text{ such that } (i, v) \in E \wedge \\
&(\exists_{=1} q \in Q \text{ such that } Q_V(i) = q \wedge \\
&Q_E((i, v)) = q)) \\
\forall o \in O : &\nexists v \in V \text{ such that } (o, v) \in E \\
&(\exists_{=1} v \in (I \cup G) \text{ such that } (v, o) \in E \wedge \\
&(\exists_{=1} q \in Q \text{ such that } Q_V(o) = q \wedge \\
&Q_E((v, o)) = q))
\end{aligned}$$

Besides the input and output nodes, all other nodes represent gates in the circuit. For all gate nodes, the number of incoming edges is equivalent to the number of outgoing edges. Additionally, the number of incoming or outgoing edges is the same as the number of arguments for the gate. Importantly, in this case, any qubits controlling the application of the gate, e.g., the first qubit in a controlled-not gate, also count toward the number of arguments.

$$\forall g \in G : |g| = 2n \quad \text{where } n \text{ Number of arguments for gate of } g$$

Furthermore, for all gate nodes, each incoming edge has a corresponding outgoing edge with the same assigned qubit.

$$\begin{aligned}
\forall v \in G : &(\exists v' : (v', v) \in E) \implies \\
&(\exists q, v'' : Q_E((v', v)) = q \wedge (v, v'') \in E \wedge Q_E((v, v'')) = q)
\end{aligned}$$

Therefore, for each qubit, there exists one path from its input node to its output node such that all gates that are applied to it are visited in order of application.

Graph Construction

When using the circuit graph to optimize a quantum program, the first step is to systematically construct the graph from the program. The creation of the graph starts with the input and output nodes for each qubit in the circuit. For each declaration of a qubit, an input and output node pair is created. If a register is declared, a pair is created separately for each qubit in the register; as a result, for a register with size n, n pairs are created in total. Next, the gate applications in the program can be iterated. For each gate application, a corresponding gate node is created. To insert this node into the graph, each qubit argument requires an incoming edge to this node and a corresponding outgoing edge from the node. Additionally, the incoming node

must come from the gate that was previously applied to the qubit or, if no gate was applied beforehand, the input node. Similarly, the outgoing edge must lead to either the next applied gate or the corresponding output. Therefore, for each qubit, the edge coming into the output node can be diverted to the gate node, and an outgoing edge, from the gate to the output node, can be created. Repeating this step for all gate applications in the program results in the corresponding circuit graph.

An example of a simple, unoptimized circuit graph is depicted in Fig. 3.2. For simplicity, all applied gates are depicted inside of the corresponding node in the circuit graph. The circuit consists of three qubits, q_0, q_1, and q_2. Their input and output nodes are depicted on the left and right of the graph, respectively, and are labeled with the corresponding qubit. Firstly, an X gate is applied to the first qubit q_0. Then, a controlled-not gate is applied to the first two qubits, q_0 and q_1, where q_0 is the control qubit. Simultaneously, two Hadamard gates H are applied to the third qubit q_2. Lastly, another X gate is applied to the first qubit q_0 while another controlled-not gate is applied to the second and third qubits q_1 and q_2. In this case, the second qubit q_1 is the control qubit.

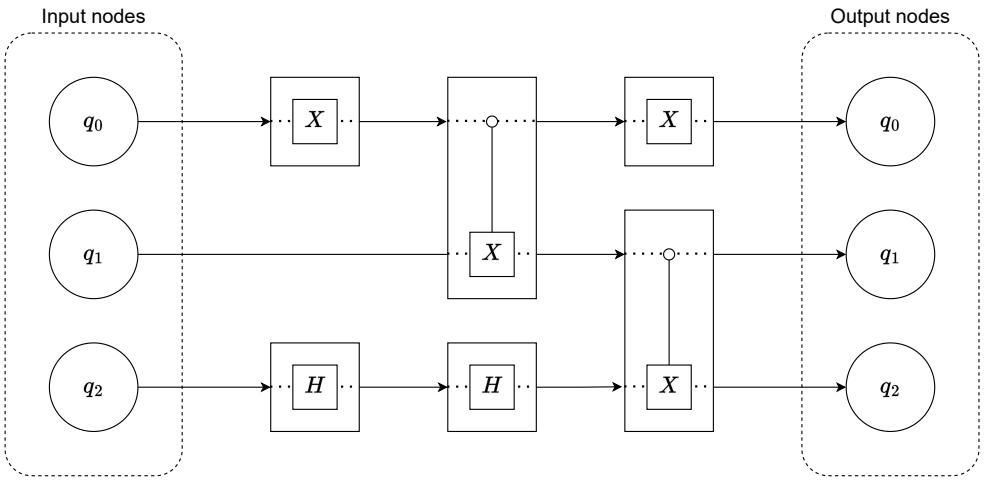

Figure 3.2: An example of a simple, unoptimized circuit graph.

Graph Optimization

The next step in the optimization process is the application of optimization rules. In this case, these are peephole optimizations. To optimize the graph, subgraphs are systematically iterated for each qubit $q \in Q$ by walking along the wire path of q. A wire path is a path that starts at the input node of a qubit q, ends in the corresponding output node, and follows the edges corresponding to the qubit q. For each node *start* in the wire path, subpath p starting at *start* and following the wire path up to a maximum length *max* are iterated. This *max* length depends on the maximum number of nodes that are affected by an optimization rule. Each subgraph p is checked for an optimized

alternative based on a list of optimization rules R. If one is found, the subgraph is replaced with it. However, a single iteration over the entire graph may not find all optimizations, as the application of optimizations may enable further optimizations. For example, after removing a gate combination, two previously separated Hadamard gates may now represent a null gate combination, which can, in turn, also be removed. Therefore, the process needs to be repeated until no more optimizations can be applied. This is the case if the process is repeated without applying any optimizations. The optimization algorithm, depicted in Alg. 1, shows how the subpaths are iterated.

```
Data: Circuit Graph C = (V, E, Q, Q_E, Q_V), List of optimization rules R,
      Maximal rule length max
repeat ← true;
while repeat = true do
    repeat ← false;
    foreach q ∈ Q do                                  # Iterate through all qubits
        start ← i   where Q_V(i) = q ;                          # I input nodes
        start ← v   where Q_E((start, v));              # Next node on wire path
        while start ∉ O do                                      # O output nodes
            last ← start;
            p ← [start] ;                    # Create path that starts with start
            for i ← 0 to max do                 # Iterate paths up to max length
                foreach r ∈ R do             # Check all rules for applicability
                    if r can be applied to p then
                        apply r to p;
                        repeat ← true;
                        break;
                    end
                end
                if ∄ v : Q_E((last, v)) = q then          # Existence of next node
                    break;
                end
                last ← v   where Q_E((last, v)) = q);
                append last to p;
            end
            start ← v   where Q_E((start, v)) = q;   # Next node on wire path
        end
    end
end
```

Algorithm 1: The pseudocode for the algorithm used to optimize a circuit graph.

Let us consider an example optimization process of the circuit graph depicted in Fig. 3.2. Firstly, subpaths for the wire path of the first qubit q_0 are iterated. Here, the subpath of note is the path with the first two gate nodes X and CX. Since the

X node is the child of the input node, we know that the value of the qubit will be $|1\rangle$ after the application. Therefore, we know that the controlled-not gate will always apply the X gate to the second qubit q_1, and we can replace the CX gate node with a simple X node that is only visited by the second wire path. For simplicity, we assume that the optimizations are only applied after all subpaths are iterated. In turn, there is no further optimization that can be applied to the second qubit in this round of the optimizations. However, in our implementation, the optimizations are applied while iterating over the circuit. Lastly, on the third qubit wire, there are two consecutive Hadamard gates that can be removed. Overall, the first round of optimizations results in the circuit graph depicted in Fig. 3.3.

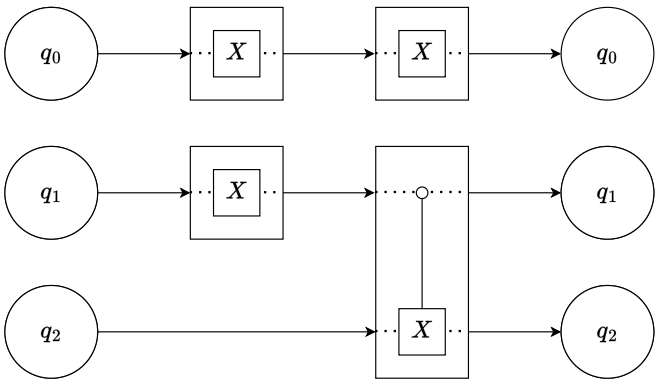

Figure 3.3: The circuit graph after the first optimization step.

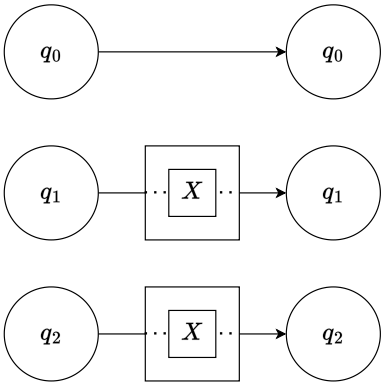

Figure 3.4: The completely optimized graph.

In the next optimization round, the two X gates on the first qubit wire are applied consecutively. Therefore, they can both be removed from the circuit. Finally, the combination of an X gate child of an input node followed by a controlled-not gate can, again, be optimized such that the CX gate is replaced with a simple X gate on the third qubit wire. The result is a circuit where the first qubit remains unchanged and only an X gate is applied to the second and third qubit. This result is also depicted in Fig. 3.4.

Graph Translation

After all possible optimization rules were applied and no others were enabled in turn, the only remaining step is to translate the circuit graph back to a programmatic description. The qubits for the circuit are easily declared by iterating over all input or output nodes. Additionally, the qubit count can be reduced by leaving out all unused qubits, i.e., qubits to which no gates are applied. This is the case if the wire path only consists of the input and output nodes. In the case of the optimized circuit, depicted in Fig. 3.4, the first qubit q_0 can be skipped when adding the declarations to the translated program.

For the gate applications, the only requirement is that, if a gate node is a descendent

of another, its translated gate application statement must come after the statement of the ancestor node. Therefore, for most circuit graphs, there are multiple possible programs describing the circuit correctly. For example, when translating the optimized graph described above, the order of the X gate applications to both the second and third qubit is irrelevant, such that both possibilities, i.e., either applying the gate to q_1 or q_2 first, are valid translations of the program. Furthermore, while iterating over all input or output nodes and adding the corresponding qubit and register declarations to the beginning of the program is the easiest approach, the order and placement of the declarations are also arbitrary as long as they are declared before their first use.

3.5 Command Line Interface

The command line interface of the compiler enables the programmer to interact with the compiler and specify specific behavior. There are currently four different parameters that can be specified: the input file, output file, optimizations to be applied, and the verbosity of the compilation.

The first, and most important, parameter is the input file. Since without a specified input file the compiler has no program to compile, the parameter is mandatory. It can be given by either "`-i`" or "`--input`" followed by the path to the input file. Similarly, the output file is specified with another parameter; it can be specified with "`-o`" or "`--output`" followed by the path to the output file. However, this parameter is not required, and the default behavior is to create a file with the name "output.qasm" in the current directory.

Next, the optimization parameter can be used to specify which optimizations are applied to the code. Here, the possibilities are either none, the null gate, peeping control, Hadamard reduction, or control reversal optimization rules. Each optimization has a specific keyword that can be passed as a parameter to indicate the optimization. Additionally, multiple different optimizations can be applied by listing them separated by a plus sign. For example, to apply both the null gate and peeping control optimizations, the correct parameter would be "`nullgate+peepingcontrol`". The optimization parameter is specified by either passing "`-O`", in this case an uppercase "O" to differentiate it from the output, or "`--optimization`". When the optimization is not explicitly specified, the default behavior is no optimization. While the user can specify which optimizations are applied, they cannot specify how many optimization iterations are performed; the optimization algorithm always iterates until no further optimizations are found.

Another parameter that can be based on the compiler is the verbose parameter; it toggles the verbose mode of the compiler on. When this mode is active, the compiler prints info, warning, and error messages out to the user. In comparison, only specific error messages and warnings are displayed to the programmer when the compiler is executed without the verbose mode. In contrast to the other parameters, the verbose mode does not require any other arguments and can be activated by passing either "`-v`" or "`--verbose`" to the compiler. If the parameter is not passed, the compiler is

executed normally.

Besides the parameters that can be passed to the compiler to modify its behavior, it also provides a help text. The help text can be accessed by passing the help parameter to the compiler with either "`-h`" or "`--help`". Additionally, it is displayed when an invalid or unknown parameter was given by the programmer together with a message that indicates that the given parameter is invalid. The help text itself gives a short description of the compiler and lists all parameters that can be passed. Furthermore, each parameter is listed with a short description of its behavior.

An example for the usage of the CLI is depicted in Fig. 3.5. The first parameter is the input parameter; it indicates that the compiler should compile the Luie program called "program.luie" in the current directory. Next, the output file is specified. After the compilation is complete, the compiled code should be located in the build directory of the current directory and be called "program.qasm". Following the specification for the input and output files, the optimizations parameter indicates which optimizations are to be applied to the program. In this case, the compiler applies both the null gate and peeping control optimization gates. Lastly, since the verbose parameter was not specified, the compiler will only print essential errors and warnings and no other informational messages.

```
1 ./LUIECompiler --input "./program.luie"
2                --output "./build/program.qasm"
3                --optimization nullgate+peepingcontrol
```

Figure 3.5: A command line interface example.

4 Implementation

After the high-level discussion of the programming language and compiler, as well as the formal definitions for both the syntax and translations, in the following chapter, we discuss the concrete implementations of the different stages of the compiler. Firstly, we present the implementation of some general classes and concepts in the compiler as well as the compiler class, which calls and connects all other stages. Next, we discuss the lexical and syntactic analysis of the program code; more specifically, we present the grammar implementation of the language and explain its structure. After the lexical and syntactic analysis, the different parts of the semantic analysis are discussed. Then, the implementation of the code generation and its parts are presented. This is followed by a discussion of the implementation of the optimization.

4.1 Compiler

The compiler consists of multiple classes and stages that work together to compile and optimize the given program. In the following implementation chapter we use specific language when talking about different compilation stages. Any step that is executed in our compiler happens at compile time. The first steps are the analysis, including the lexical, syntactic, and semantic analysis. These are followed by the code generation. Any computations performed in this step are executed at generation time. Lastly, the code is optimized. Despite the discrete stages, there are multiple overarching classes and commonly used structures that do not belong to a certain stage. In the following section, we discuss the implementation of general functionalities and introduce the commonly used structures.

When the compiler program is started, the main function calls the Command Line Interface class to parse the arguments to the compiler data class. The compiler data is used to enable and control the compilation process; it contains all required information, such as the path to the input and output files. Furthermore, it specifies which optimizations are to be applied to the compiled program. Lastly, it indicates whether the compilation process is to be executed verbosely.

After the compiler data is parsed, the main function passes it to the compile function of the compiler class. This is a static class containing general compilation and printing functionalities as well as corresponding properties. We implement it as a static class so that all parts of the program can easily access its functions and properties. The first property of the compiler is the `Printer` function; its default value is the native console printing function of C#, but it can be set to any arbitrary function that takes a string as an input and does not return anything. While this property is not changed in a normal compilation process, it is used to check the console output of the compiler

in the test cases. Secondly, the verbose property indicates whether the compiler prints not only the errors to the user but also the warnings and informational logs.

To interact with the printer property of the compiler, there are a variety of different printing and logging functions; in the end, all use a single print function that uses the printer property to display a given message. While the printing functions, used for printing errors and warnings, call the print function directly, the logging functions have an additional check so that they only print if the compiler is executed verbosely. Lastly, the logging functions also use compiler services of the runtime to allow for special injections. All of the logging functions have the optional member name and line number arguments that are annotated with a special attribute. In turn, if the optional arguments are not set, the compiler injects the name of the caller for this function as well as the line number where it was called. This can be helpful when trying to debug issues with the compiler.

The compile function handles the entire remaining compilation process. First, it retrieves the program code from the given input path. For the reading and saving of files, a separate `IOHandler` class exists that has some basic logic for reading and writing files. Then, the parse tree is created and passed to the semantic analysis of the compiler. Here, all errors and warnings are printed to the user. If any errors are thrown, the compilation is aborted. Next, the code is generated from the parse tree. If the compilation resulted in an error, again, the compilation is aborted. After the code is generated, it is optimized based on the optimizations given in the compiler data. Lastly, the resulting program is written to the file specified by the output path.

4.1.1 Command Line Interface

As the interface between the programmer and the compiler itself, the command line interface (CLI) is an essential part of the compiler. Its purpose is to interact with the programmer and create the compiler data, which specifies the behavior of the compiler. To achieve this, the CLI consists of two different parts. The first are the attributes that are used to annotate the compiler data class, and the second part is the CLI handler; it parses the input arguments and creates the compiler data from them. Additionally, it prints the help text to the console if needed.

An attribute is a C# class that can be used to annotate fields and properties of another class; together with reflection, it can be used to create a modular and easily extendable compiler data class with parameters and descriptions for each compiler data property. Reflection allows programs to get information on types of loaded assemblies. In our case, we are interested in the information on classes, more specifically information on properties of the compiler data class. We create custom attributes with a class that inherits from the `Attribute` class and contains the required information about the properties we need. With reflection, our program can get a list of all properties with specific attributes and use their information to create, e.g., the help text of the CLI. In turn, we have two custom attributes in our program. The first is the `CLIParameterAttribute`, which specifies both the short and long name of an attribute corresponding to the compiler data property. For example, in the case of the

input path property, the short name is a lower case “i” and the long name is “input”. The second attribute is the `CLIDescriptionAttribute`; it contains a description for the property it is applied to. Then, this description can be displayed in the help text. In the case of the input path property, the description describes that the parameter describes the path to the input file. The code for the input path property example is depicted in Fig. 4.1.

```
1 [CLIParameter('i', "input")]
2 [CLIDescription("Path to the input file.")]
3 public string InputPath { get; set; } = string.Empty;
```

Figure 4.1: The input path property declaration with its parameter and description attribute.

To parse the command line arguments and create the compiler data from it, the command line interface class is used; it consists of functions to parse the arguments and print the help text to the console. Since the syntax for the CLI argument input is very basic, the command line parsing itself is basic. The input string is split at each space and given to the function as a string array. First, the function retrieves the CLI parameter attributes for all compiler data properties. Then, it iterates over the string array and, for each array entry, checks whether the parameter attribute matches the string; for example, in the case of the input path property, the string would have to be either “-i” or “–input”. If this is the case, the corresponding parsing function, depending on the matched argument, is called with the string array and a reference to the current index. A reference to the index is used to allow for further changes to the index, depending on the number of possible arguments for a parameter. For example, the verbose parameter will always return true and not change the index, while a path parameter will increment the index and return the next string of the array. If no attribute can be matched or an argument exception is thrown in the process, the compiler will notify the user of the invalid or missing argument and print the help text to clarify the compiler options.

The help text is created based on the attributes of the properties in the compiler data class. Firstly, the function retrieves both the parameter and description attributes. Since the compiler data class does not include the help parameter, as it is only used in the CLI context, the help text manually prints its information. Then, the parameters are iterated. For each parameter, a matching description is searched. If no description can be found, the text defaults to a message indicating that none is available. Lastly, both the parameter and corresponding description are printed to the console.

4.1.2 Symbols

The symbols are a major part that is used in most stages of the compilation process, mainly the semantic analysis and code generation; they are used to store all necessary information on data type, composite gates, and similar objects in the language.

The basis is an abstract symbol class that consists of an identifier and error context property. The identifier uniquely identifies a symbol in the scope it is used while the error context saves information about the symbol and its declaration environment to be used for possible error messages. In total there are eight different symbols that are derived from the class. In the following, we discuss these symbols and how they are used in the compiler. The hierarchy of the symbol classes is depicted in an UML diagram in Fig. 4.2.

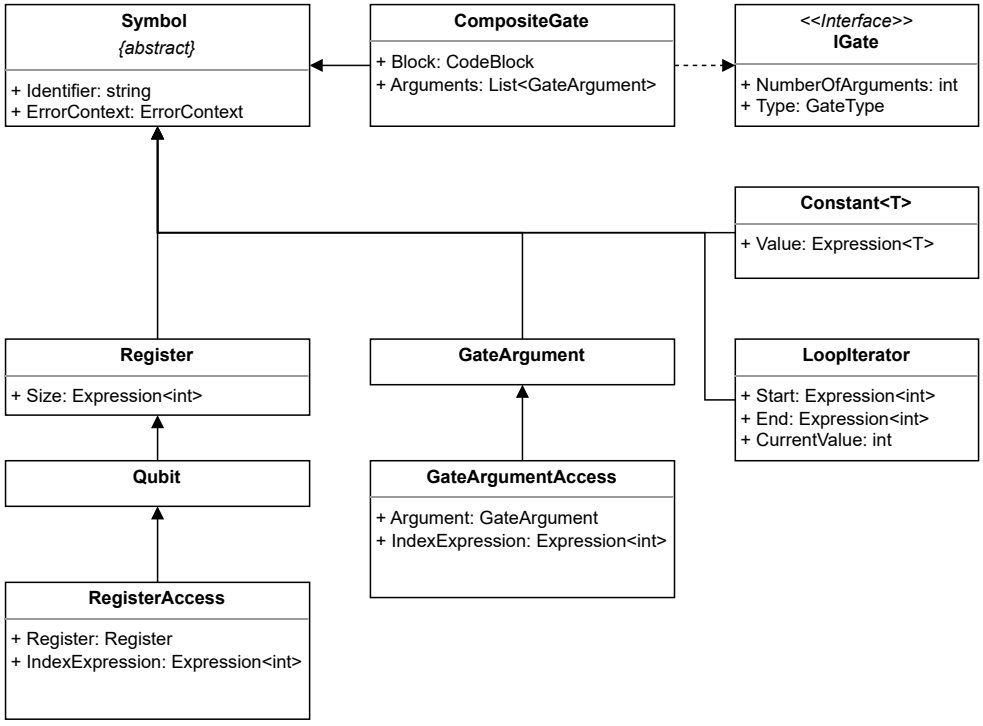

Figure 4.2: A diagram showing the hierarchy of symbol classes.

There are three different symbols for quantum data types. The first kind is the register. Furthermore, it is the basis of all other quantum data type symbols. Besides the inherited properties, it consists of an expression that represents the size of the register. The expression specifying the size of the register may not be evaluable when its symbols are created because the value of some identifiers may be known only at generation time. For example, the body of a loop statement is unrolled at generation time, and the value of the loop iterator depends on this iteration. In turn, the value of the iterator identifier may not be known when the register symbol is created. Therefore, instead of saving a constant value, the symbol specifies the expression for its size. Then, when the code is generated and all values are known, the expression can be evaluated.

Next, the qubit symbol inherits from the register. While physically the qubit is the basic element and a register consists of qubits, in our case, it is useful to assume that

a qubit is a special case of a register where the size is one so that the qubit can inherit the register symbol properties and functions. Furthermore, instead of differentiating between a qubit and register declaration, a register declaration is sufficient for this hierarchy, only requiring a differentiation when printing the code. In turn, all registers with size one are inherently optimized to be qubits in the target code. Besides setting the size to one, the qubit symbol has no additional attributes.

The last quantum data type symbol is the register access. This symbol is necessary because the access of a register is not implemented as an expression. Since a register access cannot be used in any expression and only in the context of a gate application or if-statement, instead of implementing special quantum expressions that can only be used in limited cases, we create another symbol. The register access symbol inherits from the qubit symbol, as it can be used whenever a qubit symbol can be used. For example, in the case of a gate application statement, a list of qubits, to which the gate is applied, is saved. In turn, since each register access is a qubit, no additional list or differentiation is required for the gate application, only a virtual function to return the translated qubit code. The symbol also contains an integer expression that specifies which index is accessed and the register that is accessed. Similar to the size expression of the register symbol, the expression is saved since it may not be evaluable when the symbol is created.

There are two symbols for classical data; these are the constant symbol and the loop iterator. The constant symbol represents a named constant of differing types. To allow for different variable types, the symbol has a generic parameter `T`. Additionally, the generic parameter is restricted to implementing the number interface, indicating that it is a numeric type. Furthermore, as with the other symbols where a property may depend on a value that is not known at the time of creation, the constant symbol value is given by an expression. This expression is of the type `T`.

Secondly, the loop iterator symbol is the other symbol used to represent classical data; however, it is specifically designed for the loop statement of our language and its special properties. It is used to unroll the loop body and propagate the value of the current iteration of the loop. To achieve this, the symbol contains the start and end indices as integer expressions; they can be evaluated when the code is being generated. Furthermore, the symbol contains a current value property that contains the value of the current iteration. When the code for the loop body is unrolled by being iterated, any reference to the iterator symbol will evaluate to the current value.

The last three symbols are all related to the composite gates. Firstly, there is the composite gate symbol itself. It is created when a composite gate is declared and, later on, is used whenever it is referenced in a gate application statement. To allow for a predefined gate, an interface is declared that represents the important attributes of a gate; these are the number of arguments to the gate and the gate type. The gate type differentiates between the different predefined gates, such as the Hadamard or X gate, and the composite gates. This gate interface is implemented by the composite gate. Besides the properties required by the gate interface, the composite gate symbol consists of a block property and an arguments property. The block property contains the code block that is the body of the composite gate; it is used to inline the statements

of the gate whenever it is applied. Secondly, the arguments are a list of gate argument symbols that represent the arguments that can be passed to the composite gate.

The second kind of composite gate-related symbols are the above-mentioned gate argument and gate argument access symbols. Both represent an argument to a composite gate and serve as a placeholder for the symbol of the argument that is passed when the gate is applied. When inlining, the placeholder, i.e., the argument symbol, is mapped to the given argument. Furthermore, while the composite gates themselves can only operate on the previously discussed quantum data type symbols, the programmer does not specify the type of the argument, which can be both a qubit or quantum register, when declaring the composite gate. Therefore, we introduce a symbol that will ignore type checking until the composite gate is called and the arguments are specified. Then, if an argument is of an invalid type for a specific use, a type error is thrown. Similar to the register access, an additional gate argument access symbol is required because an access cannot be represented as an expression; it inherits from the gate argument symbol, has a reference to the gate argument that is accessed, and an integer expression that evaluates to the access index at generation time.

4.1.3 Symbol Table

The symbol table is a data structure that saves all symbol information and offers functions for adding new symbols and retrieving symbol information. For example, a new symbol can be added with the `AddSymbol` function, and symbol information can be obtained based on the identifier with the `GetSymbolInfo` function.

To allow for different symbol contexts, the scope data structure is used. Scopes are used to enable the declaration of different variables with the same identifier in independent scopes, i.e., two scopes where neither is the other's ancestor nor descendent. The scope class contains an identifier dictionary that maps a string, in this case representing the identifier, to the corresponding symbol. Because of the quantum if-statement, a scope can also be guarded by a qubit. Therefore, a scope has an optional guard symbol that either references the symbol that guards the scope and all descendants or is null. Lastly, the scope contains a reference to the code block it belongs to. While the parse tree is traversed, each statement is added to the code block of the current scope. Later, the code block can be passed along in the generation to, e.g., create the loop statement belonging to the scope.

The scopes are saved in the symbol on a stack. Each time a new code block is entered when traversing the parse tree, a new scope is pushed onto the scope stack. Similarly, each time a block is exited, the scope stack is popped to remove the latest scope from the data structure. To simplify the interactions with the scope stack, the symbol table contains both a current scope and a current identifier map property; both reference the top-most scope on the stack and the identifier map of the current scope, respectively.

While the scope data structure holds a reference to the guard symbol, the symbol is known before the creation of the current scope because first the if-statement is traversed, then the code block. Therefore, the class interacting with the symbol table

would need to save the current identifier and pass it to the push scope function. To avoid this additional complexity, the symbol table holds a guard stack that can be interacted with by using the additional `PushGuard` and `PopGuard` functions. In turn, the symbol table can use the scope stack to pass the current guard to a newly created scope. Additionally, guards and scopes are abstracted when using the symbol table such that the class traversing the parse tree only needs to push and pop the correct information and is not required to save the current guard information.

Lastly, the symbol table contains a property that generates a unique identifier. To achieve this, it contains a private integer field `_uniqueId` that is initialized with a value of zero. Each time the unique identifier property is retrieved, the id is incremented. The identifier is just the id with a "id_" prefix. The property is used when generating the OpenQASM code, as it does not allow for different variables with the same identifier. While this is a very simplistic approach, these resulting identifiers are always unique, as long as the same symbol table is used, and the resulting identifiers are predictable. This predictability is especially helpful when creating test cases for the translation of source code, as we can simply give the expected target code.

4.1.4 Error Handling

Error handling is an essential part of both the semantic analysis and code generation phase of the compiler; without clear and informative error messages, debugging compilation errors is both challenging and tedious. Therefore, our compiler supports a number of different errors and exceptions, mainly differentiating between two types of errors with different severities. The first type is the *warning*. A warning from the compiler can indicate issues in the source code that may cause unintended behavior. However, the issue itself does not prevent the compilation of the program and is simply an indication that there may be something wrong. In contrast, the *critical error* is caused by a flaw in the source program that prevents the correct compilation and will result in the abortion of the compilation. In the following, we will discuss the error handling of our compiler in general, different warnings and critical errors the compiler may raise, their corresponding causes, and implementations.

Any compilation error caused by the user is represented by the abstract compilation error class, which contains three different properties. The first property is the error type; it indicates the severity of the error, as described above. Secondly, the description property should hold a description of the error that can be used to inform the programmer what issue occurred. Lastly, the error context saves relevant information about the source code properties of the error; it is implemented as a `struct` that contains both the line and column index of the source code location of the error. The error context can be created either from the line and column directly, the token where the error occurred, or the parsing rule with which the error is associated. In many cases, the error context can be created when the error occurs. For example, if an undefined identifier error is found in the declaration analysis, the parser rule context is given, and the error can easily be created from it. However, in other cases, errors may be thrown at generation time, and the corresponding parser rule context may no

longer be known. Therefore, all symbols also contain the error context corresponding to their declaration.

While the abstract compilation error class contains general properties that are required for all errors, each error may contain additional properties that are required to give a clear and informative description of the error. Additionally, since many errors occur in reference to a specific identifier, the compiler uses an additional abstract identifier error that contains a string property that holds the identifier. In the following, we will go through the warnings and errors. All of their classes, including their respective properties, are depicted in Fig. 4.3.

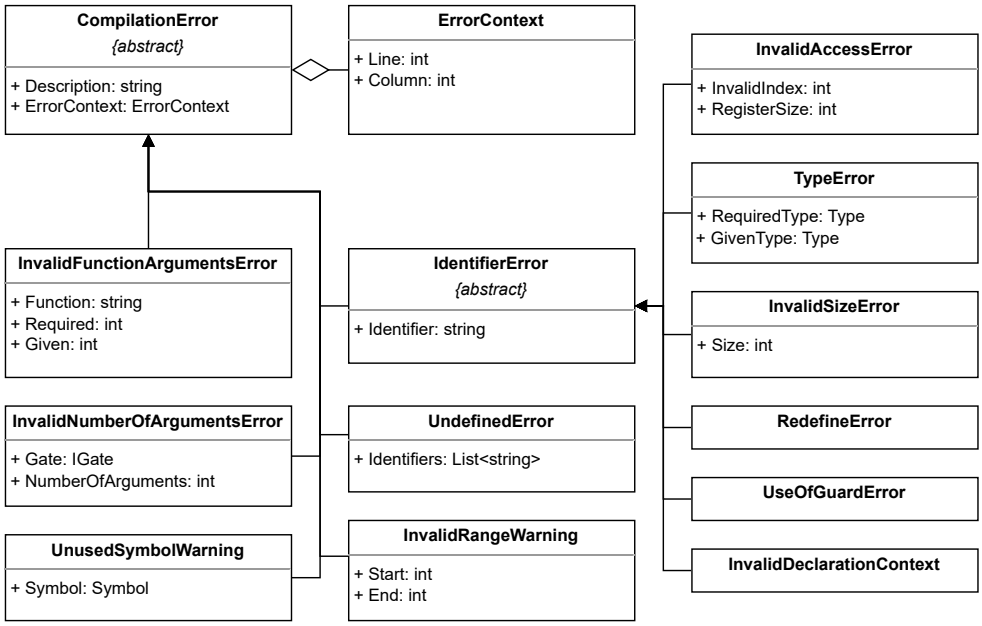

Figure 4.3: A diagram showing the hierarchy of error classes.

The compiler can throw two different kinds of warnings. The first is the invalid range warning; it can occur in the context of loop statements. The loop statement iterates over a range that is defined by the user. It can be given as either a size n and iterate from zero to $n-1$ or a start and end index, i_{Start} and i_{End}, respectively, and iterate from the start to the end. However, the range iterator is designed to only increase. Therefore, a range where $i_{Start} \geq i_{End}$ is invalid. Since the loop statement is unrolled at compile time, a range with a size less than or equal to zero can just be ignored. However, the user may not intend this behavior. Therefore, the compiler warns the user that the range is invalid. The class for the warning contains the start and end indices for the invalid range.

Secondly, the unused symbol warning is raised when a symbol, e.g., a register or composite gate, is defined in the source code but never used. This symbol is stored in the corresponding error class. The unused symbol does not have any negative effect on the compilation, and the optimization step can easily remove, e.g., an unused register.

Therefore, this is only a warning, and the program can be compiled. However, an unused symbol may indicate that the wrong symbol was used somewhere else or part of the program is no longer used. Hence, the user is warned of the unused symbol, and unintended behavior may be prevented.

The first critical error is the invalid access error. It occurs when a register is accessed at an invalid index i, i.e., i is either smaller than zero or larger than $size - 1$, where $size$ is the size of the register. While this error could easily be ignored and would cause no issue when compiling the program, the resulting code would be an invalid circuit description. The invalid access error saves the identifier that is accessed, inheriting from the identifier error, as well as the invalid access index.

Secondly, the invalid number of arguments error for gates and functions is caused when the number of arguments given to either a gate or function does not correspond to the number of required arguments. For example, the Hadamard gate always expects one argument, while the controlled-not gate requires two. Similarly, the `sizeof` function operates on only one argument. The compiler cannot proceed when given too few arguments; for the opposite case, while dropping any leftover arguments is possible, it would result in unexpected behavior. Therefore, the compiler reports an error for both cases and aborts the compilation. In the case of the invalid function argument error, the function as well as the number of required and given arguments are saved; for the gate error, the given number of arguments and the gate interface object, containing both the gate type and number of required arguments, are saved.

Another error is the invalid size error. It occurs when a register is declared with an invalid size. A size is invalid if it is less than or equal to zero. A register with no entries cannot be used for anything and likely indicates an issue in the program, while a register with a negative amount of entries is impossible. Therefore the compilation is aborted and the error is thrown. In the corresponding error class, which inherits from the identifier error, the invalid size is saved as well as the identifier used in the declaration.

The next two errors are concerned with the declaration of variables in a given context; they are the undeclared and already-declared errors. An undeclared error is raised when a variable is used in a context where it is not defined. In this case, the symbol table does not have a symbol stored for the given identifier, and the compiler cannot continue. In contrast, the already-declared error occurs when a variable is declared in a context where the same variable identifier has already been assigned to a different symbol. While the compiler could overwrite the previous declaration, this can easily lead to unexpected behavior, and, in turn, we do not allow a declaration in the same scope to be overwritten. While the redefine error saves the identifier that is redefined, inheriting from the identifier error class, the undefined error saves a list of possible undefined identifiers. A list of identifiers is saved because, in our implementation, an expression returns a list of undefined identifiers, which is used to create the error.

The type error is thrown when a variable is used in a function or gate but is not the required type. Languages with loose typing may be able to convert some types to the required type by, for example, parsing the integer value of a string. However, this can not only result in unexpected behavior and hard-to-debug errors in the code, but, in

the case of a quantum language, it may also require the conversion between classical and quantum data, which is not easily achievable. In this case, the error class inherits from the identifier error class and contains the required and given type as well as the identifier of the symbol.

Next, another error is the use-of-guard error; it occurs when a qubit is referenced in a context that is guarded by itself. While the compiler can easily translate any such occurrence, they result in an invalid circuit description. As described in Sec. 2.2.1, a gate that operates on and is controlled by the same qubit cannot be reversible. Therefore, the compiler prevents the generation of an invalid circuit and aborts the compilation. The error class only contains the corresponding identifier and, in turn, inherits from the identifier error.

Finally, the last error is the invalid declaration error. Since composite gates are only meant to operate on the registers and qubits given as arguments, our language does not allow for register or qubit declaration in the context of a composite gate. Any register or qubit declaration inside the scope of a composite gate declaration will result in an invalid declaration context error. The error class inherits from the identifier error class and, in turn, contains the identifier of the corresponding invalid declaration.

One advantage of a semantic analysis separate from the code generation is that, if an error is found, the semantic analysis can continue. For example, if an undefined identifier error occurs while generating code, the code generation can no longer continue as the identifier cannot be mapped to a symbol, and, therefore, the information for the code generation is incomplete. However, a semantic analysis can continue as it does not need any more information about a symbol than its existence. In turn, while errors in the code generation phase use typical error handling with exceptions that abort the tree traversal, the semantic analysis uses a custom error handler. Mainly, this error handler object consists of a list of compilation errors and a report function that takes an error and adds it to the list. Additionally, the handler also contains properties that indicate whether it contains critical errors and two lists that return only the critical errors or warnings, respectively. After the semantic analysis, the compiler will iterate over the errors in the handler and print them as either errors or warnings, depending on their error type.

Besides the code generation exception, there exists the internal exception. This exception is used in all stages of the compiler. The code generation exception wraps compilation errors in the code generation stage where the user is at fault. However, there may be cases where the compiler enters an error state for which the user is not responsible, e.g., an unexpected null reference. For these cases, the internal exception is used.

4.2 Lexical and Syntactic Analysis

The lexical and syntactic analysis steps of the compiler are not implemented by hand, but ANTLR4 is used to generate both the lexer and parser. In Sec. 2.4.6, a short introduction is given into the generation tool. The tool does not only generate the

lexer and parser for the language but also some data structures, like the parse tree, and abstract classes that are used for the following compiler steps. In the following, we discuss the ANTLR grammar of Luie and give an overview of the different classes that are generated by ANTLR.

4.2.1 Grammar

The entry point for the grammar is a parsing rule called `parse`. It states that the program consists of a main code block followed by the end-of-file lexer token `EOF`. The main code block starts with arbitrarily many gate declarations, or none at all, and is followed by any number of declarations and statements in any order, or none at all. Similarly, a "normal" code block also consists of any number of declarations and statements; however, it does not allow for gate declaration. The grammar for the entry point, main, and general code block is depicted in Fig. 4.4.

```
1 parse     : mainblock EOF;
2
3 mainblock : gateDeclaration* (declaration | statement)*;
4
5 block     : (declaration | statement)*;
```

Figure 4.4: The basic structure of parsing rules for Luie.

While syntactically it would be equivalent to define the main block to consist of gate declarations and a code block, the code generation implementation requires the main code block to directly hold the declarations and statements. Internally, each code block object holds a list of declarations and statements that, in turn, can hold references to other code blocks. However, a block itself cannot hold a reference to another block. Furthermore, the code generation is only called on the main code block, which then recursively calls the code generation for each declaration and statement. Therefore, the main code block needs to contain the declarations and statements directly; otherwise, the code generation will always generate an empty program.

There are three kinds of declaration rules in the grammar. The first is the declaration rule for qubits and registers. It starts with the register keyword lexer token, followed by an optional expression in brackets giving the size. If no size is given, a qubit is created; otherwise, a register with the given size is created. The identifier for the qubit or register is the last part of the rule.

Next, the constant declaration rule is used to declare constants that can be used at compile time, e.g., for the size of registers or to give the number of loop iterations. Similar to register and qubit declaration rules, the rule starts with a constant keyword token. Then, the identifier is given and separated from the type of the constant with a colon. Lastly, the expression for the value of the constant is given after an equality sign.

The final declaration rule is used for composite gates. Again, it starts with a corre-

```
declaration          : registerDeclaration | constDeclaration;

registerDeclaration : REGISTERKEYWORD
                      ('[' size=expression ']')?
                      id=IDENTIFIER ';';

constDeclaration     : CONSTANTKEYWORD id=IDENTIFIER
                      ':' type=TYPE '='
                      exp=expression';';

gateDeclaration      : GATEKEYWORD id=IDENTIFIER
                      '(' param=gateArgument ')'
                      DO block END;

gateArgument         : IDENTIFIER (',' IDENTIFIER)*;
```

Figure 4.5: The parsing rules for declarations in Luie.

sponding keyword and is followed by the identifier of the gate. Next, the gate argument rule is given in parentheses. The gate argument rule matches any number of identifiers, separated by commas, requiring at least one identifier. Lastly, the gate declaration rule ends with a code block rule. All declaration rules are given in Fig. 4.5.

The grammar contains four different types of statements. The first is the gate application statement, followed by two control flow statements and, lastly, the skip statement. The gate application statement starts with the gate rule. A gate can either be a constant or predefined gate, such as the Hadamard or X gates; a parameterized gate, such as $P(\lambda)$; or a composite gate given by its identifier. In the case of the parameterized gate, the parameter is given by an expression. Following the gate rule, the register rules specify to which qubits or registers the gate is applied. Here, at least one argument is required, but there can be arbitrarily many arguments, each separated by a comma. A register or qubit can either be given by the corresponding identifier or as a register access. In this case, the index to be accessed is given by an expression. The grammar rules related to the gate application statement are also depicted in Fig. 4.6.

Next, the first control flow statement is the if-statement. It starts with the `if` keyword, which is followed by the register rule. This register is the qubit that controls the execution of the body. After the control bit is specified, the `ifStat` rule gives the code block that is controlled by the control bit. This is followed by an optional else-block; it starts with the `else` keyword and the corresponding code block. Both the main and else code blocks are introduced by the `do` keyword. Lastly, the if-statement is ended with the `end` keyword.

The second and last control flow statement is the loop statement. Similar to other statements and declarations, it starts with the corresponding `for` keyword. Next, the identifier for the loop iterator is given, followed by the `in` keyword. Then, the range for the loop statement is specified. There are three different possibilities to give the

```
1  statement : gateApp       | qifStatement
2            | forstatement | SKIPSTAT ';';
3
4  gateApp   : gate register (',' register)* ';';
5
6  gate      : type=CONSTGATE | id=IDENTIFIER
7            | paramGate=PARAMGATE
8              '(' param=expression ')';
9
10 register  : IDENTIFIER
11           | IDENTIFIER '[' index=expression ']';
```

Figure 4.6: The parsing rules for gate statements in Luie.

range. Firstly, the simplest syntax for a range is the start and end indices separated by two dots. The other two options both start with the `range` keyword followed by arguments in parentheses. The first option only gives the length of the range as an expression, while the second specifies both the start and end indices as expressions. Lastly, the body of the loop is a code block surrounded by the `do` and `end` keywords. The grammar rules for both control flow statements are depicted in Fig. 4.7.

An expression mainly consists of three different grammar rules. The first rule is the expression rule itself. It matches the operands with the lowest precedence. In our case, these are both the addition and subtraction operations. The left-hand side of the addition and subtraction is also an expression. In contrast, the right-hand side specifies a term rule. Alternatively, the expression can also be neither an addition nor a subtraction but just a term.

Similar to the expression rule, the term represents two operations. In this case, they are the multiplication and division operations with neither the highest nor the lowest precedence. While the left-hand side of both operations matches the term rule again, the right-hand side specifies a factor. Again, the term can also be neither a multiplication nor a division but a factor instead.

Lastly, the factor rule includes all operations with the highest precedence. The first

```
1  qifStatement : IF register ifStat elseStat? END;
2
3  ifStat       : DO block ;
4  elseStat     : ELSE DO block;
5
6  forstatement : FOR IDENTIFIER IN range DO block END;
7  range        : start=INTEGER '..' end=INTEGER
8               | RANGE '('length=expression')'
9               | RANGE '('start=expression ','end=expression')';
```

Figure 4.7: The parsing rules for the control flow statements in Luie.

```
1  expression : left=expression op='+' right=term
2             | left=expression op='-' right=term | term;
3
4  term       : left=term op='*' right=factor
5             | left=term op='/' right=factor     | factor;
6
7  factor     :'(' exp=expression ')' | func=function
8             | identifier=IDENTIFIER | value=INTEGER
9             | op='-' factor;
10
11 function   : func=FUNCTION '(' param=funcParam ')';
12 funcParam  : IDENTIFIER (',' IDENTIFIER)*
13            | expression (',' expression)*;
```

Figure 4.8: The parsing rules for an expression in Luie.

option is an expression in parentheses. The second possibility is a negated factor. Finally, the factor can either be the result of a function, the value of an identifier, or an integer value itself. While the value of an identifier is just given by the identifier itself and the integer value is specified by giving the integer, the result of a function is given by the function rule. The function rule starts with the name of the function. An example of a function name is `sizeof` or `power`. The name of the function is followed by the function parameters in parentheses. The function parameters are either a list of identifiers or expressions separated by commas. Both require at least one identifier or expression but allow for arbitrarily many. The additional case of a list of identifiers is added since registers are not allowed in expressions but are used in the `sizeof` function. However, this can lead to issues when, e.g., the `min` function is given a list of constants. These problems are addressed in the implementation for the creation of function objects where, in some cases, a list of identifier expressions is created from a list of identifiers. All expression grammar rules as well as the function-related rules are given in Fig. 4.8. Furthermore, all terminal symbols, such as the different functions and the regular expressions for identifiers and integers, are given in Sec. B of the appendix. Note that the discussed grammar is semantically equivalent to the grammar used in the implementation; however, some of the formatting and naming was adjusted for the thesis.

4.2.2 Data Structures and Classes

Based on the grammar discussed in the previous section, the ANTLR tool generates different classes and files that can be used for the lexical and syntactic analysis of the source program. The tool generates token files as well as the necessary classes for the analyses; these classes inherit from abstract base classes such as `Lexer` or `Parser` that are contained in the ANTLR runtime package. Additionally, the tool can also generate optional classes that can be used for the semantic analysis of the program or for code generation; these classes are the `Listener` and `Visitor`.

After the lexical and syntactic analysis yields the parse tree of the source code, both the listener and visitor classes can be used to traverse the tree. They implement a list of virtual functions without any effect. Then, these can be overridden and extended by custom functionality. The visitor class has a generic type parameter `Result` and implements a visit function that returns `Result` for each grammar rule. For example, in the case of our grammar, as discussed in Sec. 4.2.1, the visitor class implements, among other things, a `VisitParse` and `VisitMainblock` function. Additionally, for each grammar rule, there exists a context that provides information about the rule, such as its line and column in the source code. Furthermore, the context can be used to get information about the terminal in a rule. For example, in the case of the expression rule, the context can be used to learn which operation, either addition or subtraction, was given. The parse tree can be traversed by implementing the different visit functions and, in their implementation, having them call their child rules.

In contrast, the listener class does not have a generic type parameter, and, in turn, its functions do not return anything either. However, instead of implementing a visit function for each grammar rule, the listener implements both an enter and an exit function for each rule. Furthermore, the functions corresponding to the children in the parse tree are not explicitly, but the tree is traversed by a walker class, and the enter and exit functions are called in order of the traversal. Therefore, the `EnterParse` function is the first function that will be called, while `ExitParse` is the last. Similar to the visitor functions, both the enter and exit functions have the context of the corresponding grammar rule as an argument. Both our semantic analysis and code generation implementation create custom listener classes instead of visitors. We use listeners as they offer an implementation better suited for our use case and an overall simple implementation, as they only require passive listening functions. In contrast, the visitor needs to actively implement the tree traversal. Furthermore, visitors require a unified return type; while for most rules, this could be a translatable, this type would not fit the creation of expressions.

4.3 Semantic Analysis

The second step in our compilation process is the semantic analysis of the input program. The analysis checks for any semantic errors in the code that can be detected before the next compilation step, the code generation. Our implementation of the semantic analysis differentiates between a declaration analysis and type checking. The declaration analysis is concerned with checking for errors related to the declaration of variables. For example, when a variable is used, it checks whether it was declared previously, or when a variable is declared, the analysis ensures that the identifier is not in use in the current scope. In contrast, the type checking ensures that the type of a variable is consistent with the use of the variable.

Both analyses are implemented as separate `Listeners`, as described in Sec. 4.2.2. This is done to separate both classes and give them concrete purposes while making the overall structure of the compiler more modular. In turn, since both analyses

are implemented separately, they currently each require a traversal of the parse tree, making the semantic analysis more inefficient. However, this can easily be addressed by writing a `Listener` that wraps both analyses, traverses the parse tree only once, and calls the corresponding function for each wrapped analysis.

4.3.1 Declaration Analysis

The declaration analysis reports any errors that are caused by the use of identifiers in invalid contexts; these are the use of an undeclared variable, the declaration of a variable in a context where it is already defined, and the use of a qubit in a code block that is guarded by the same qubit. A code block is guarded by a qubit if it is contained, directly or indirectly, in an if- or else-statement with the qubit as the control, i.e., either the code block or an ancestor code block is the block of the statement. Additionally, it also reports a warning when a variable is declared but never used.

For its analysis, the class creates a symbol table and adds any symbol that is declared. Further, it executes all other functions required for a valid analysis with the symbol table, like pushing scopes onto and popping them from the stack when required in the parse tree traversal. Before adding a symbol to the table, i.e., every time the parse tree traversal exits a declaration, the `Listener` checks whether the identifier is already declared in the current context; if this is the case, the corresponding error is reported. Similarly, each time an identifier is referenced outside of a declaration, the analysis ensures that the variable is defined; otherwise, it reports the corresponding error as well. Lastly, whenever a register rule is encountered in the traversal, the analysis not only checks that the variable is declared but also whether the current code block is guarded by the referenced register. If this is the case, the analysis reports an error. While the register parsing rule is only used for gate applications and if-statements and not for function calls such as `sizeof`, only performing the check for the register rule cannot result in an invalid use of an identifier. This is the case because all function calls result in constant values and only refer to properties of a register, not the register itself; therefore, the compiled program will not have a reference to the register in the controlled context.

While the symbol table suffices for reporting the previous errors, it does not track the usage of symbols. Therefore, it cannot be used for the unused symbol warning. For this analysis, the class contains a dictionary that maps a symbol to the number of references. Each time a symbol is added to the symbol table, the corresponding entry in the dictionary is initialized to zero. Then, for every reference to the symbol, its usage counter is incremented. At the end of the tree traversal, specifically when exiting the main code block, the analysis iterates over all entries in the dictionary and reports a warning whenever the usage counter is still zero. However, the identifier consisting of only the underscore character "_" is exempt so that it can be used as a throwaway variable, similar to its behavior in other programming languages.

4.3.2 Type Checking

Type checking is used to ensure that any use of a symbol occurs in a valid context for this symbol. For example, while a qubit symbol can be used as the argument for a gate application, it does not represent a classical numerical value and, therefore, cannot be used in the context of a factor. Similar to the declaration analysis, a symbol table is used, and all symbols that are declared while traversing the tree are added to it. Additionally, the scopes are also pushed and popped according to the traversal to allow differently typed variables in independent scopes.

To check the use of a symbol, each grammar rule where an identifier can be used checks its type. For this, each function gets the identifier string and retrieves the corresponding symbol information from the symbol table. In the case of the register rule, only register symbols are allowed. However, if an index is given, i.e., a register is accessed, the given register symbol cannot be a qubit.

Next, identifiers in the factor rule can only be either a loop iterator or a constant symbol because a classical numeric value is required in expressions. In turn, register symbols, e.g., are not valid. In contrast, the `sizeof` function does only allow for register symbols. Since the `sizeof` function is the only one that takes identifiers as arguments and not expressions, it is the only one that requires type checking.

While identifiers in expressions can only reference classical data, gate applications require quantum data types as arguments. In this case, all symbols given as arguments are checked. If they are not a register, a type error is reported. Furthermore, while composite gate allow register are arguments, this is not the case for the predefined gate. Therefore, for predefined gates, all symbols that are not qubit are reported. Additionally, after the arguments were checked, it is ensured that the number of given arguments is equal to the number of required arguments. Besides the predefined gate, a composite gate can also be applied. In turn, they are referenced with their identifier. Therefore, in the gate grammar rule, only composite gate symbols are valid.

Similar to the application of predefined gates, the if-statement requires either a qubit or register access symbol. If they are not given, again, a type error is reported. In contrast, the range rule is not checked for type error but for an invalid range; this error is reported if the start index is greater than or equal to the end index, i.e., the size of the range would either be zero or negative.

Lastly, in all cases where a quantum data type is required, if the given symbol is a gate argument, the type checking is skipped and any issues are reported when the code is generated. This is done because composite gates do not require the type of an argument. Therefore, when traversing a composite gate code block, the compiler cannot know whether the argument is a register, qubit, or register access; the type checking is postponed to the code generation, when the types of the arguments are known.

4.4 Code Generation

To generate the target code, first, the parse tree is traversed and the source code is translated to an in-memory representation consisting of objects corresponding to different language concepts. These concepts can be divided into three different kinds: statements, declarations, and code blocks. All of the objects implement the `ITranslatable` interface; it requires a translation function that translates the source code representation to the target code representation. In turn, they are often referred to as translatables. The target code representation is a collection of objects that describe an OpenQASM program; they can be translated directly to the textual OpenQASM code. In the following, we discuss the generation of the source code representation from the parse tree, the translation from the source code to the target code representation, and any other utilities that are used in the process.

4.4.1 Source Code Representation

Similar to the implementation of the semantic analysis, the parse tree is traversed with another custom listener. However, in contrast to the semantic analysis, the code generation listener does not directly interact with a symbol table but uses a separate code generation handler.

The code generation handler facilitates the creation of the source code representations when traversing the parse tree. Firstly, it contains a main code block; this code block is initiated as an empty code block without a parent when the handler is created. Furthermore, it will hold, directly or indirectly, the references to all other source code objects. The second important property is the symbol table. The handler implements different methods for interacting with this table. For example, it contains methods for both pushing and popping scopes as well as guards. Additionally, the handler implements unique functions for each symbol that can be added to the table, with a protected general function. This is done because some symbols need additional logic when they are added to the symbol table. For example, when a register symbol is added to the table, a register declaration is also added to the current code block.

In the code generation listener, similar to the semantic analysis, the scopes and guards are pushed and popped based on the tree traversal. Furthermore, the symbols are also added according to the declarations in the source code. Except for the main code block, all translatables, i.e., code blocks, statements, and declarations, belong to exactly one code block. In turn, whenever one is reached in the traversal of the parse tree, the translatable is created and added to the corresponding code block. This code block is always identified as the current code block. Whenever a code block is entered, the current code block is updated to the newly created one. In turn, when a block is exited, the current is set to the parent block of the current one.

The first example of a translatable is the register declaration; as described previously, whenever a register symbol is added, the corresponding declaration is added to the current code block. In contrast, statements require more information. For each statement, the relevant symbol information is read from the symbol table based on the

given identifier. Furthermore, any additional information is retrieved from the code generation handler. Then, both the symbol and additional information are used to create the statement object. For example, a loop statement requires both the loop iterator symbol and the code block object of its loop body. In turn, since the code block needs to have been traversed, the loop statement is created in the exit function of the loop rule. However, as the code block already existed, the listener class needs to save the last popped scope and retrieve the code block, corresponding to the loop body, from it. The loop iterator can easily be read from the symbol table based on the given identifier. While the type checking ensures the correct type of the symbol in most cases, a code generation exception may be thrown if the symbol in the table is not a loop iterator. Finally, the loop statement is created and added to the code generation handler and, in turn, to the current code block. The creation of the other statements works similarly; however, since the predefined and composite gates are implemented as separate translatable statements, the gate applications need to differentiate between the two and create the corresponding statement.

The most basic translatable is the code block. It represents a block of code in the source program and, in turn, contains all statements and declarations contained in the block; they are saved in a list of translatables. Next, the code block object also contains a reference to its parent code block. If the code block is the main block of the program, the parent is set to null. While each variable has a unique identifier, this identifier is only unique in the context of its declaration, and an independent context may contain a declaration for the same identifier. Therefore, the identifiers used in the source code may not be usable in the target code. To solve this issue, each declaration is assigned a new unique identifier. The code block contains a dictionary that maps all its declarations to the corresponding unique identifiers. Besides these three properties, the code block implements some functions to add translatables to the list and some utility functions to, e.g., retrieve the unique identifiers for declarations.

Another type of translatable is the declaration. While there are multiple different declarations, e.g., register and composite gate declarations, in our source language, most concepts are not translated to a corresponding declaration in the target language; in turn, there currently exists only one possible declaration. However, for extensibility, the compiler contains an abstract declaration class; it includes a symbol property with a reference to the symbol that is declared by the object. The register declaration inherits from the abstract class. The third group of translatables is statements. Each statement inherits from an abstract statement class. Each statement contains an error context that corresponds to the source code location of the statement. In total, there are six different statements.

The first two are the different gate application statements, the composite gate statement and the gate application statement. Since the implementation of the translation differs so greatly, they are implemented as two different statements. Both contain a gate property; however, while the gate property of the composite gate statement is a composite gate, the gate application contains a predefined gate in its property. Furthermore, they both contain an arguments property. In this case, while the gate application statement simply saves a list of symbols, the composite gate contains a

dictionary that maps the argument symbols used in the composite gate to the symbols given as arguments.

Next, three statements are related to the control flow of the program. Firstly, the loop statement contains both the loop iterator symbol and the body of the loop in the form of a code block. Similarly, the quantum if-statement also contains a reference to its code block. Additionally, the statement includes a guard property that holds the symbol guarding the execution of the code block. Lastly, the else-statement simply inherits from the if-statement and contains no additional properties, only differing in the translation implementation.

Lastly, in contrast to most other statements, the skip statement does not contain any additional properties. All the different translatables, including the code block, statements, and declarations, are depicted in Fig. 4.9.

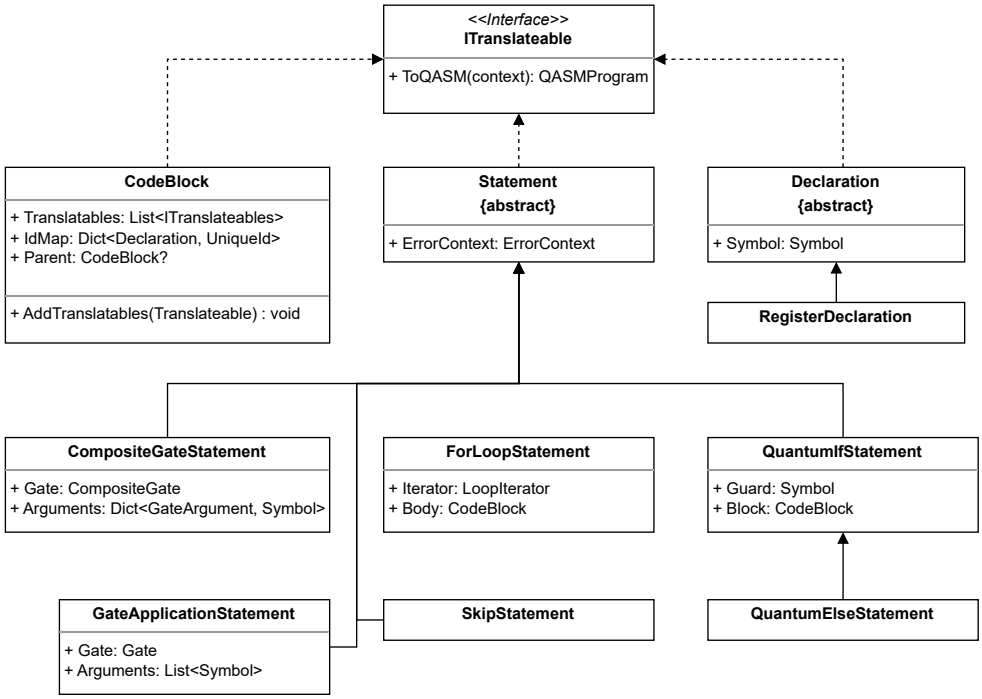

Figure 4.9: A diagram showing the hierarchy of translatable classes.

4.4.2 Translation

To create the target code representation that can be translated to the final program, the translatables, i.e., code blocks, declarations, and statements, need to be translated. For this, all translatables implement a translation function that takes the code generation context and returns the target code representation in the form of a `QASMProgram` object.

The code generation context contains all additional, relevant information for the code generation that is not already contained in the translatable objects themselves;

this may include information that is not known at the time the translatables are created and instead injected when they are translated. The first property is the current block, which holds a reference to the code block for which the code is generated. Since all statements must be contained in a block, there must also always exist a current code block. The block, e.g., contains a dictionary that maps declarations to their new unique identifiers; this dictionary is filled while translating the declarations. Therefore, this information must be injected at generation time with the help of the code generation context.

Similarly, the code generation context holds the symbol table that is generated while creating the source code representation. However, this symbol table may be changed while the code is translated. For example, composite gates are inlined where they are applied. However, the translation of the composite gate requires an empty symbol table, as they do not have access to any symbols outside the scope of the composite gate. In turn, the translation of the composite gate cannot use the symbol table of its current block but must create a new one. The symbol table itself is used to retrieve symbol information for, e.g., the evaluation of expressions where it may be used to get the value of a constant symbol.

Lastly, the context contains a dictionary that maps an argument to a symbol. This is used for the translation of the composite gates. Since there may exist nested composite gates, i.e., a composite gate that calls another composite gate, the dictionary contained in each composite gate statement is not sufficient for the translation alone. In turn, for each composite gate translation, the corresponding mappings are added to the dictionary.

The translation of any source code representation will always start with a code block translation, as it is always the root of a program. However, our target code representation does not contain a code block equivalent. In turn, instead of translating the code block directly, a new program object is created, and each translatable, contained in the code block, is translated individually and appended to the program object. Then, the result is returned.

In contrast, the register declaration has an equivalent in the target language. In this case, this is the qubit declaration code. However, as previously described, the identifier that is declared may not be unique. In turn, the symbol table is used to create an identifier that is unique in the resulting program. For this, a static integer is used and incremented. Furthermore, this identifier is added to the dictionary of the current block so that any reference to the declaration can be mapped to the newly created unique identifier. While the size of a register declaration is given in the form of an expression, the target code object requires a constant integer value. Therefore, the size expression is also evaluated before the code object is created.

Of the statements, the skip statement is the easiest to translate; it represents an empty operation and, therefore, is translated by returning an empty program object. Similarly, the gate application statement has a simple translation. Since the target language also contains gate application statements, the translation simply needs to convert the gate and arguments to the target code representation. This can be accomplished with some simple helper functions and lookups. In contrast, the composite gate

statement does not have any target code equivalent and is, instead, inlined. In turn, the composite gate translation simply calls the translation for the body code block and injects some argument mappings. Then, the translation of the body is returned.

The translation of the loop statements is, in principle, similar to the translation of a composite gate; it also contains a code block that is translated. However, the code block is not inlined once but needs to be translated multiple times, each time with a different value for the iterator. To achieve this, both the start index and end index expressions are evaluated. Next, the current value of the iterator is set to the start index, and a for loop is executed until the current value is equal to or larger than the end value. For each iteration, the code block is translated and added to a program. At the end, the program consists of the unrolled loop statement and is returned.

Lastly, the translations of the if- and else-statements are mostly identical. In our target language, OpenQASM, the desired behavior for both the if- and else-statements can be accomplished by adding a control and negative control to each gate application statement in the code block, respectively. For this, the program object contains a function to add a guard. In this function, all code objects are iterated, and the guard is added to each gate application code. For the else-statement, a negated guard is added. Therefore, the translation of the if- and else-statements consists of translating the code block and adding the guard symbol as a possibly negated guard to it.

4.4.3 Target Code Representation

The target code is translated from the source code representation in the form of a `QASMProgram` object. This program object functions as a wrapper for the translated gate applications and declarations and, in turn, holds a list of `Code` objects that represent these translations. Additionally, it provides functions to add and optimize the program or print the final target code string.

Each code object inherits from the abstract code class, which implements two abstract functions. The first is a `ToCode` function, which converts the object to code by returning the text string of the code object. Secondly, the class implements a function that checks the semantic equivalence of a given code to the code object. In this case, if two code objects result in the same behavior of the program, they are semantically equal. For example, two gate applications are equal if they apply two semantically equivalent gates and the arguments and guards are semantically equivalent. The semantic equivalence is used in the optimization rules to check whether different rules are applicable. In the following, we will discuss the different code classes and their implementation.

The main two code classes are the gate application code and qubit declaration code. The gate application code represents a gate application and, in turn, contains the guards for the gate, an object that represents the gate, and a list of qubit arguments to the gate. All these different properties have their own code classes, guard codes, gate codes, and qubit codes, respectively. For the code conversion, the gate application adds the control and negative control depending on the number of positive, i.e., not-negated, guards and negative guards, respectively. Next, the gate is converted and

added. If they exist, both the positive and negative guards are converted, joined by commas, and added to the code. Similarly, at the end, the arguments are translated and added, separated by commas, to the code string. To check the equivalence of two gate applications, first, the arguments are checked. Since the order of arguments matters, both argument lists are iterated simultaneously, and the equivalence of each argument pair is checked. In contrast, the order of guards does not matter; in turn, it suffices that, for each guard in one list, there exists a guard in the other that is semantically equal. Lastly, the equivalence of the gates is checked. Additionally, when a gate code is created, some extra checks and transformations are performed. Firstly, a check is performed that qubits used as guards are disjunct from the qubits used as arguments to the gate. As for the transformations, any use of a controlled-not or Toffoli gate is replaced with an X gate, and the control qubits are added to the guards; while the circuit description remains equivalent, any transformations in the following optimization stage do not require additional cases to find different kinds of controlled-not gates.

The guard code holds a reference to the qubit code that guards the gate as well as a boolean value that indicates whether the guard is negated or not. Two guard codes are semantically equivalent if their corresponding qubits are and their negation status matches. The code conversion just references the conversion of the qubit. Next, the gate code holds the gate type, the code conversion prints the gate based on the type, and the semantic equivalence compares the gate type. Additionally, there exists a parameterized gate that inherits from the gate code class. In this case, the object holds an additional parameter property, and functions are adjusted accordingly. Lastly, the qubit code class holds a reference to the register symbol it is associated with for error handling and the unique identifier of the qubit in the target code. The code conversion simply returns the unique identifier, and the semantic equivalence compares the identifiers. Additionally, there exists a register access code that inherits from the qubit code. It contains an extra index property. In the semantic equivalence function, the base function is called, i.e., the equivalence function of the qubit code, and the indices are compared. Furthermore, the code conversion returns an access to the identifier based on the index property.

The declaration code is an abstract class that represents any declaration in the OpenQASM program. It contains a unique identifier property for the variable it declares. Additionally, with the identifier, the class can also implement the semantic equivalence check where the identifiers are compared. The qubit declaration code inherits from the class and contains an additional size property, corresponding to the size of the register. Depending on the size of the declaration, either a single qubit is declared in the conversion, if the size is one, or a quantum register for the given size is declared.

While the gate application and qubit declaration code, as well as the related classes, are sufficient to translate the source code, the resulting program does not contain any measurements. Since our language only supports implicit measurements, these are added after the translation and optimization of the code. More specifically, the program class contains a function that prints the final program with the required

```
gate c_h_reg(control, reg) do
    qif control do
        for i in range(sizeof(reg)) do
            h reg[i];
        end
    end
end

qubit c;
qubit[3] a;
c_h_reg c, a;
```

Figure 4.10: An example Luie program to show the code generation process.

OpenQASM header and adds measurements for each qubit and register in the circuit; when the compiler writes the final program to the specified output file, this function is used. To represent the measurements in the target code, there exist both a bit declaration and measurement code class. Similar to the qubit declaration, bit declarations contain a size property and, depending on that size, either a single bit or classical register is declared. In contrast, the measurement class contains two unique identifiers for the measurement target and the storage variable. While the semantic equivalence compares both identifiers, the code conversion returns the OpenQASM statement based on the two identifiers.

4.4.4 Example Process

After the discussion of the different steps and parts of the code generation, we want to take a look at an example. Our example program is depicted in Fig. 4.10. First, a composite gate `c_h_reg` is declared. It takes two arguments, a control qubit and a register `reg`. In the composite gate, a quantum if-statement is used to apply the following statements based on the value of the control qubit. Next, a loop statement iterates over the size of the register argument `reg`. To each qubit in the register, the Hadamard gate H is applied. After the composite gate is declared, a qubit `c` and a register `a` are declared. The register is given a size of three qubits. Lastly, the composite gate is called with the `c` as the control bit and `a` as the `reg` argument.

The first step in the code generation process is the creation of the source code representation. The representation of the code example is depicted in Fig. 4.11. The root of the source code representation is the main block. While this block is depicted in the diagram, for simplicity, all other code blocks are omitted. The main block contains three different translatables: the declarations and one statement. The two declarations are qubit `c` and register `a`, declared in the code example. They are followed by a composite gate statement. This gate contains both a dictionary that maps the gate argument symbols to the given arguments and a reference gate code block, represented by an arrow in the diagram. The body of the composite gate contains an if-statement where `control` is the control qubit. In turn, the if-statement contains a loop statement

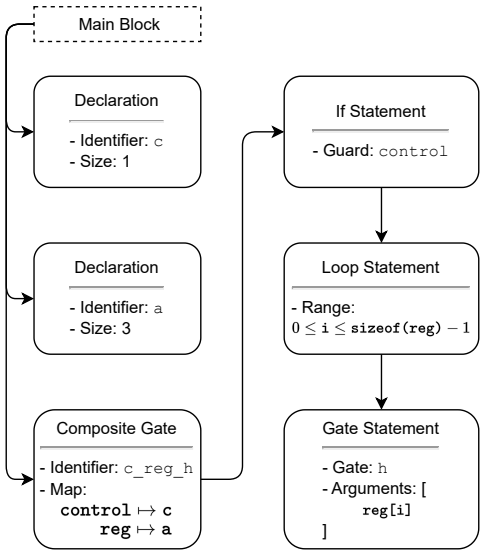

Figure 4.11: The source code representation of the example program.

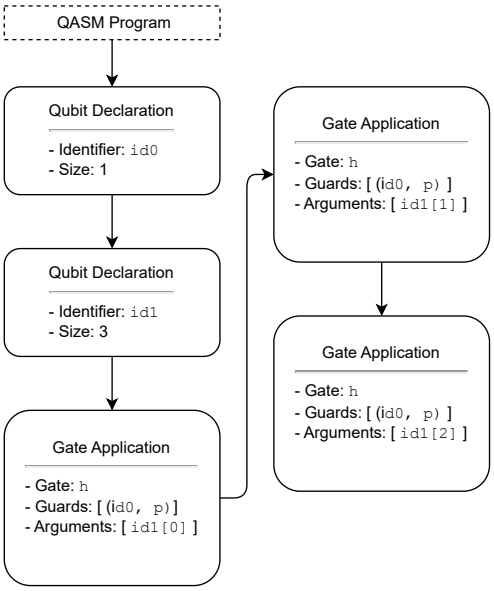

Figure 4.12: The target code representation of the example program.

that iterates over the range from zero to `sizeof(reg)` − 1. Although the start index is given by a constant value, both the start and end of the range are given as expressions and are evaluated when the loop statement is translated. Lastly, the loop statement contains a gate statement that applies the Hadamard gate to the `reg[i]` qubit.

Next, the source code representation is translated to the target code representation. The corresponding translation, without the measurement statements added later in the compiler, is depicted in Fig. 4.12. The two declarations are easily translated to their corresponding qubit declarations. However, to ensure the uniqueness of each identifier, they are assigned new identifiers; `c` becomes `id0` and `a` becomes `id1`. The next three statements, the composite gate, if-, and gate statement, have no target code equivalent. First, the composite gate translation maps the argument symbols used in the gate body to the newly declared qubits. In this case, `control` is mapped to `id0` and `reg` to `id1`. Then, the body of the gate is translated and inlined. To translate the if-statement, the loop statement needs to be translated first. Here, the loop body is translated for each value in the range of the loop. For this, the start and end expressions are evaluated to zero and two, and, in turn, the loop body is translated three times. For each translation, the new value of the loop iterator `i` is propagated. The loop body itself only consists of a gate application, where the Hadamard gate is applied to the qubit `id1[i]` where `i` changes depending on the iteration. When the loop statement is translated to the three gate applications, the if-statement adds its guard symbol `id0` to the guard list of each gate application. The result is a sequence of three gate applications, guarded by `id0` and applying the Hadamard gate to each

```
OPENQASM 3.0;
include "stdgates.inc";
qubit id0;
qubit[3] id1;
ctrl(1) @ h id0, id1[0];
ctrl(1) @ h id0, id1[1];
ctrl(1) @ h id0, id1[2];
bit id0_measurement;
measure id0 -> id0_measurement;
bit[3] id1_measurement;
measure id1 -> id1_measurement;
```

Figure 4.13: The OpenQASM translation of the example Luie program.

qubit in the `id1` register.

Lastly, the remaining step is the conversion from the target code representation to the compiled program code. The final program code, including the language header and measurements, is depicted in Fig. 4.13. The program object is translated by iterating through all its code objects and calling their translation function `ToCode`. In the case of the declaration, a qubit is declared if the size is one, while a register is declared for any other size. Since the uniqueness of identifiers is ensured when translating to the target code representation, the identifier of the code object is used for the translation to text. The translation of the gate application adds a control or negative control modifier to the translation, as well as the corresponding control qubits, to the translation depending on the number of positive and negative control qubits. In the example, all gate applications contain only one positive guard. In turn, the resulting statements contain only the control modifier `ctrl(1) @` with the corresponding qubit as the first argument to the controlled gate. Lastly, a function is called that adds the header and measurement statements to the program.

4.4.5 Expressions

Expressions are used in multiple source code representation objects as a placeholder for a constant value that can only be calculated when the object is translated to the target code. In turn, they need to be created together with the source code object. When such an object is created, the expression subtree of the parse tree is traversed and an expression object is created. The initial function to create the expression differentiates between the different possible cases; for example, in the initial case, either an addition, subtraction, or a term, which can be either a multiplication, division, or a factor, is given. In the case of an addition or subtraction, the two subexpressions are created with a recursive function call, and a combined expression is created. In contrast, when the expression is a term, the corresponding expression creation function is called. Both the expression creation function for the term and factor operate similarly to the function of the expression itself. In turn, the result is an expression object.

The expression object inherits from an abstract expression class that can be either

a binary or unary operation expression, an abstract function expression, an identifier expression, or a constant expression. The expression class takes a generic type parameter with the condition that this type parameter is a number, or more specifically implements the number interface `INumber`. The type parameter is typically propagated to all subexpressions such that all elements of an expression are of the same type. The type parameter is set when an expression is created and if inferred from the context. For example, in the case of a constant declaration, the given type is used, while for range expressions, the integer type is used.

A binary operation expression is any expression with an operator that operates on two subexpressions. In our case, these are addition, subtraction, multiplication, and division. The expression is evaluated by evaluating the subexpressions and applying the corresponding operation. Similarly, the unary operator expression is an expression with an operator that operates on only one subexpression; the only unary operation is the negation. Again, the expression is evaluated by evaluating the subexpressions and applying the corresponding operation.

Next, the function expression class is an abstract class from which the different concrete function expression objects are derived. The different possible functions are `sizeof`, `power`, `min`, and `max`. Firstly, the `sizeof` expression function takes an identifier and the symbol table as an input and looks up the register information in the table. When evaluated, it returns the size of this register. In contrast, the `power` function takes two expressions as arguments, evaluates both, and raises the result of the first to the power of the value of the second. Lastly, both the minimum and maximum functions take at least one expression up to arbitrarily many as arguments, evaluate all of them, and return the minimum or maximum value, respectively.

The last two expressions are the identifier and constant expressions. Both differ from all but one expression, the `sizeof` function expression, in that they are the leaves of the expression tree and do not contain any subexpressions. Similar to the `sizeof` function expression, the identifier expression contains the symbol of either a constant or loop iterator; when it is created, it retrieves the respective information for the symbol table, and, when it is evaluated, the result is the value of the symbol. In the case of the loop iterator, the current value is used. If the type of the identifier does not correspond to the type of the expression, the value is cast. In contrast, the constant expression simply contains a value of the type given as the type parameter; when evaluated, it returns the value.

4.5 Optimization

The implementation of the compiler does not only translate the custom language to OpenQASM 3 but also allows for optimizations on the translated circuit. To apply the optimizations to the translated circuit, the circuit description is used to build a circuit graph, as described in Sec. 3.4.2. Next, the graph is iterated, and the subgraphs are checked for applicable optimization rules. If a rule is applicable, the rule is applied. The process of iterating over the entire graph is repeated for as long as rules were

applied in the previous iteration over the graph. When the optimization of the circuit is completed, the graph is translated back to a programmatic description of the circuit, and the result is returned.

The optimization itself is handled by the optimization handler. It contains a reference to the program, which is optimized, and a function to optimize the program with an optimization type as the argument. The optimization type is an enum where each value is a flag for the corresponding optimization. For example, no optimization `None` has a value of 0, null gate optimizations `NullGate` have a value of 1, and the Hadamard reduction optimization `HReduction` has a value of 4. In turn, when the optimization function is given a value of 5 or `NullGate | HReduction`, only the null gate and Hadamard reduction optimizations are applied. The optimization function retrieves the list of rules to apply and creates the circuit graph from the program. Next, a single optimization round is performed. If optimizations were applied, another round is performed; otherwise, the optimization process is complete. Then, any unused qubits, i.e., qubits to which not gates are applied, are removed from the program. This is the case if the successor of the input node of a qubit is the corresponding output node. Lastly, the circuit graph is translated back to a program and returned.

In the following, we will discuss the implementation of the different steps in the optimization process. This includes the circuit graph in general, the construction of the graph based on the program, and the translation of the graph back to a circuit. Lastly, we discuss the implementation of the optimization rules.

4.5.1 Circuit Graph

The basis of the circuit graph implementation is a list of interfaces for basic graph elements, i.e., an edge, node, graph, and path interface. They stipulate the basic properties and functions the elements require. For example, while the edge interface contains a start and end node, the graph interface contains a list of nodes and edges as well as functions to add and remove nodes and edges. Based on these interfaces, basic edge, graph, node, and path classes are implemented. They implement the different basic functionalities required by the interfaces. Lastly, these basic elements are extended into circuit graph elements; these are circuit nodes, edges, a wire path, and, finally, the circuit graph. Additionally, the circuit node is extended further to separate classes for input, output, and gate nodes.

The circuit graph is constructed from a `QASMProgram` input. First, all qubit and register declarations are selected from the program; for each qubit declaration, a graph qubit is created and the corresponding input and output nodes are added to the graph. In the case of a register declaration, the same creation is repeated for each qubit in the register. Similarly, all gate application statements are selected from the program; for each statement, a gate node is created and inserted into the graph by inserting the node at the end of the wire path for each qubit argument to the gate. The result is the complete circuit graph.

When translating the circuit graph back into a programmatic description, often, multiple translations are valid. For example, given a circuit with two qubits q_0 and q_1,

with a Hadamard gate H being applied to both, in the program, it does not matter whether the gate is applied to q_0 or q_1 first. The only requirement for the order of gate applications is that, if a gate G_0 is applied to a qubit before another gate G_1 on the same wire, the statement corresponding to G_0 must come before the statement of G_1 in the resulting program. For our translation, we implement an eager approach where a single wire path is translated as far as possible. If a gate is reached for which other gates need to be translated first, the translation switches to the wire path that needs to be translated first.

First, a dictionary *curr*, which maps from qubits to nodes, is created where, for each qubit, a current node is initialized to the successor of the input node. Additionally, the first qubit is set as the current qubit q. Next, the main algorithm is repeated for as long as a qubit exists where the current node is not the output node. In the main body, the current node g of the current qubit q is selected. If g is an output node, the current qubit is set to a qubit whose current node is not an output node. If qubit q' exists in the arguments to the gate of g, where the corresponding current node g' is not equal to g, the current qubit is set to g' and the translation continues. If this is not the case, the translation of g is added to the program, and the current nodes of all qubit arguments are set to the corresponding successor nodes. The pseudocode for the algorithm is depicted in Alg. 2.

Data: Circuit Graph $C = (V, E, Q, Q_E, Q_V)$, $V = I \cup O \cup G$
Result: *prog* as the translation of the circuit graph C
$prog \leftarrow []$;
$curr \leftarrow [q_{init} \mapsto n \mid q_{init} \in Q, n \in G \cup O, \exists\, i \in I : Q_V(i) = q_{init}, (i, n) \in E]$;
$q \leftarrow$ first element in Q;
while $\exists\, q' \in Q : curr[q'] \notin O$ **do**
  $g \leftarrow curr[q]$;
  if $g \in O$ **then**
    $q \leftarrow$ first element in $\{q_{rem} \mid q_{rem} \in Q, curr[q_{rem}] \notin O\}$;
    continue;
  end
  if $\exists\, q' \in Q, q'$ *argument to gate of* $g, curr[q'] \neq g$ **then**
    $q \leftarrow q'$;
    continue;
  end
  add translation of g to *prog*;
  foreach q_g *argument of* g **do**
    $curr \leftarrow curr[q_g \mapsto n_{succ} \mid n_{succ} \in G \cup O, Q_E((g, n_{succ})) = q_g]$;
  end
end

Algorithm 2: The pseudocode for the algorithm used to translate a circuit graph to a program.

4.5.2 Optimization Rules

When the circuit graph is checked for optimizations, as described in Sec. 3.4.2, a list of rules is given, and, for each subgraph, all rules are checked for applicability. These rules all implement the `IRule` interface. It describes that each rule contains a maximum rule depth property and a function to check the applicability of the rule for a wire subpath as well as a function to apply the rule. The interface is implemented by an abstract optimization rule class from which all rules inherit and which contains some additional auxiliary functions. In total, the compiler contains four different rule classes, corresponding to the optimizations discussed in Sec. 3.4.1; these are the *control reversal*, *Hadamard reduction*, *null gate*, and *peeping control* rules.

The control reversal optimization is applicable when a controlled-not gate is preceded and succeeded by a Hadamard gate on both wires. However, our optimization algorithm will only check subgraphs that belong to a single wire. Therefore, instead of checking a given path for a matching condition, the rule operates on a maximum length of one gate node and checks whether the given node is a controlled-not gate. If this is the case, the rule is applicable when all successors and predecessors are Hadamard gate nodes. Lastly, the rule is applied by removing all surrounding Hadamard gates and switching the control and target qubit of the gate.

The Hadamard reduction rule is applicable when an X or Z gate has a Hadamard gate as a predecessor and successor node. In turn, the maximum length of the rule is three. Additionally, the rule saves a gate type property for the gate, which is surrounded, as well as a property for the gate type to which the rule reduces. For both cases, i.e., X or Z gates, the class contains static instances with the corresponding properties. The rule is applicable when both the start and end nodes of the path are Hadamard gates and the middle node is the sandwiched node. Additionally, we allow the rule to be applicable for controlled variants of the rule. As long as the guards of all gates are mutually inclusive, the arguments to all three gates are equivalent, and the sequence of gates is not interrupted on any wire, the rule is applicable. To apply the rule, the Hadamard gates are removed from the circuit, and the middle node's gate type is set to the reduction gate.

The null gate rule removes gate combinations that are equivalent to an identity operation. It contains an array of gate types, representing the null gate combination. In turn, the maximum rule length is the length of this array. Similar to the Hadamard reduction, the different possible null gate rules are created as static instances of the class with the corresponding combination. The rule is applicable if the gates in the path correspond to the null gate combination. Additionally, the rule is extended by allowing for controlled variants of the null combination with the same conditions imposed by the extension of the Hadamard reduction rule.

Lastly, the peeping control rule removes a controlled gate if one of the control qubits has a value of $|0\rangle$ or removes the control qubit if its value is known to be $|1\rangle$. Similar to the control reversal rule, the rule cannot operate on the wire paths alone. Instead, the rule searches for a controlled gate where the qubit, corresponding to the path, is a control qubit. In turn, the maximum length for the rule is one. If the path consists of

a controlled gate and the path qubit is one of the controls, the rule tries to evaluate the current value of the control qubit. For this, it initiates the state to false and iterates over all predecessor gate nodes on the wire. For an X or Y gate, the state is flipped, while a Z gate does not affect the value. For any other gate or controlled versions of the gates, the state cannot be evaluated. The rule is applicable whenever the state can be evaluated. When applied and the state is false, the controlled gate is removed. In contrast, if the state is true, the current qubit will be removed as a guard from the controlled gate.

5 Evaluation

In the evaluation chapter, we evaluate our implementation of the proposed compiler. The evaluation consists of two aspects. Firstly, we evaluate the optimizations performed by the compiler. In the second part, we evaluate the execution time and performance of the different compilation stages.

To evaluate the compiler, a program to compile is required. For this, we use the quantum ripple-carry adder proposed by Cuccaro et al. [CDKM04]. The quantum adder works similarly to its classical equivalent. For the arguments, it takes both a carry input qubit *cin* and an output qubit *cout* as well as two quantum registers a and b. When applied, the adder adds the first register a to the second b while taking the carry input qubit *cin* into account. If the result for the addition is larger than the register b, the carry output qubit *cout* is used. More specifically, the algorithm adds $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$ together in place. After the computation, the resulting bit string $s = s_n \dots s_0$ is mostly contained in the b register, where b is now equal to $s_{n-1} \dots s_0$, and s_n is saved in the *cout*.

To implement the ripple-carry adder, two auxiliary gates are used; these are the majority gate `MAJ` and the "unmajority and add" gate `UMA`. The `MAJ` gate computes the majority of three bits c_i, b_i, and a_i. Together with the `UMA` gate, the result is the string bit s_i in the b_i register entry while the carry qubit and a register are unchanged. The combined semantics and the intermediate values of the `MAJ` and `UMA` gates are depicted in Fig. 5.1. Their concrete implementations are given in Appendix C.1.

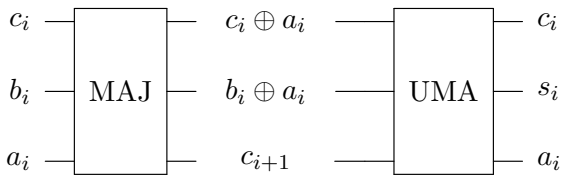

Figure 5.1: The combined semantics and the intermediate values of the `MAJ` and `UMA` gates.

To compute the addition of the entire register, the `MAJ` gate is first applied to *cin* and the first elements of the b and a registers. Since the intermediate carry bit is saved in the a register, the `MAJ` gate is applied to each element pair of the b and a registers while the previous element of the a register is used as the carry bit for the gate. After all `MAJ` gates are applied, the carry bit in the a register is written to *cout*. Next, the `UMA` gates are applied in reversed order, similar to the `MAJ` gates. In the end, the a register has the same value as before, the b register contains the result of the addition,

and *cout* possibly contains an additional carry bit. The complete implementation of the adder is depicted in Fig. 5.2.

```
1  gate adder(a, b, carryIn, carryOut) do
2      majority carryIn, b[0], a[0];
3      for i in range(sizeof(a) - 1) do
4          majority a[i], b[i + 1], a[i + 1];  end
5
6      qif a[sizeof(a) - 1] do x carryOut; end
7
8      for i in range(sizeof(a) - 1) do
9          unmaj a[(sizeof(a) - 2) - i], b[(sizeof(a) - 1) - i], a
               [(sizeof(a) - 1) - i];
10     end
11     unmaj carryIn, b[0], a[0];
12 end
```

Figure 5.2: A Luie implementation of a quantum ripple-carry adder circuit.

5.1 Optimization

The optimizations, which are applied to the quantum circuit, are the first aspect we want to evaluate. For this, we apply our optimizations to variants of the same algorithm, the quantum ripple-carry adder. Firstly, we analyze the optimizations to the adder when given classical inputs. Next, we evaluate the optimizations of the adder for inputs in superposition with different register sizes. Lastly, we compare our optimizations to optimizations that are applied by the default Qiskit transpilation process of quantum circuits.

In our first example, we use the inputs $a = |1\rangle$ and $b = |15\rangle$, with both the input and output carry qubit having a value of $|0\rangle$. Furthermore, both input registers have a size of four qubits and, in turn, have $2^4 = 16$ possible classical values. After the adder is applied, the a register and carry input, per definition, remain at their initial values of $|1\rangle$ and $|0\rangle$, respectively. In contrast, the b register now has a value of $|0\rangle$ and the carry output a value of $|1\rangle$, indicating that the result of the addition is $|16\rangle$. The quantum circuit corresponding to the example before optimization rules are applied is depicted in Fig. 5.3.

Since the circuit only consists of X, controlled-not, and Toffoli gates, neither the Hadamard reductions nor control reversal optimization rules can be applied to the circuit. However, both the peeping control and null gate optimizations can be applied. Whenever a qubit wire contains a known value of $|0\rangle$, the controlled gates can be removed. For the opposite case, the control can be removed and the resulting X gate can be removed when in combination with another X gate. In turn, the circuit can be optimized such that the resulting one only contains gates that initialize the result; only two X gates remain. While the first gate flips the first qubit of the a register,

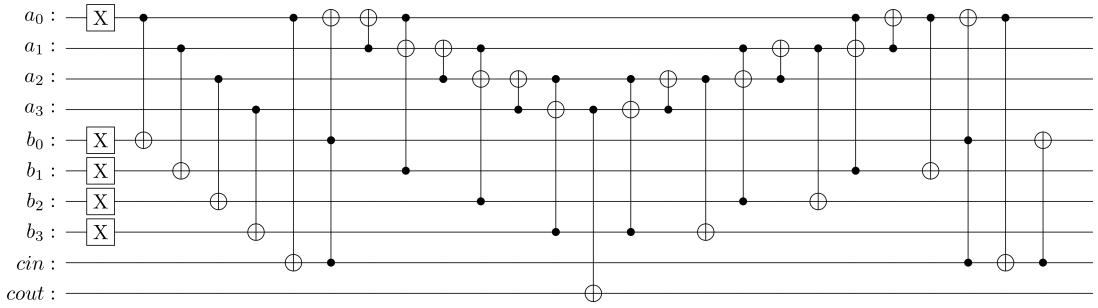

Figure 5.3: An unoptimized circuit of a quantum ripple-carry adder.

initializing it to $|1\rangle$, the second flips the carry output qubit, indicating a result of $|16\rangle$. The OpenQASM code for the optimized circuit is depicted in App. C.2.

In the second example, we input a value in superposition. Now, register a contains a value of $\frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$ and register b contains a value of $|4\rangle$. Again, both carry qubits are initialized to $|0\rangle$. After the adder gate is applied, the carry qubits and register a remain as initialized. However, the b register now has a value of $\frac{1}{\sqrt{2}}(|4\rangle + |7\rangle)$. Since the input is now no longer a classical value, the peeping control optimization can only be applied in a few cases. In turn, only twelve gates are removed. The optimized quantum program is depicted in App. C.3. In other cases, even fewer optimizations are applied; e.g., with an input of $b = |15\rangle$, only four gates are removed while four Toffoli gates are optimized to controlled-not gates. While the optimizations are not as effective for inputs in superposition, they can optimize significant parts of the program. For example, when we repeat both examples but increase the size of the registers to 64 qubits, the optimized circuit contains the same amount of gates. In these cases, if qubits are not entangled by the data in superposition, all gate applications to their wires, except for the initializations, can be removed.

We also compared the optimizations applied by our compiler to the default optimizations that are applied by the Qiskit transpilation process. Qiskit[1] [JTK*24] is a software development kit for building, simulating, and transpiling quantum circuits. Additionally, the kit can interpret OpenQASM programs and build quantum circuits from them. When transpiling quantum circuits with Qiskit, i.e., transforming to a specific domain such as another basis gate set, the kit can perform optimizations to the circuit based on an optimization level. For the comparison, we used the quantum ripple-carry adder implementation provided by the OpenQASM repository[2]. The code for the program is depicted in Fig. 5.4. The program can then be loaded with Qiskit and built into a quantum circuit. To apply the default optimizations of Qiskit, we transpile the circuit with the highest optimization level to the base gate set our compiler translates to, i.e., $\{X, Y, Z, CX, CCX, H\}$. While our compiler translates to arbitrary controlled gates, they are not used in this example. Furthermore, even the Y, Z, and Hadamard gates are redundant as they are not used.

[1] `https://github.com/Qiskit/qiskit/`

[2] `https://github.com/openqasm/openqasm/blob/main/examples/adder.qasm`

```
include "stdgates.inc";
gate majority a, b, c {
    cx c, b;
    cx c, a;
    ccx a, b, c;
}
gate unmaj a, b, c {
    ccx a, b, c;
    cx c, a;
    cx a, b;
}
// Given qubits cin, cout
// Registers a, b; n size of a and b
majority cin, b[0], a[0];
for uint i in [0: n - 2] { majority a[i], b[i + 1], a[i + 1]; }
cx a[3], cout;
for uint i in [n - 2: -1: 0] { unmaj a[i],b[i+1],a[i+1]; }
unmaj cin, b[0], a[0];
```

Figure 5.4: An OpenQASM 3 implementation of a quantum ripple-carry adder circuit.

However, while the Qiskit transpilation can apply many optimizations such as null gate optimizations, the default rule set does not contain a rule similar to our peeping control optimization rule. In turn, no controlled gates can be optimized in the circuit. Furthermore, since the null gate optimizations can only be applied after peeping control rules were applied, the transpilation process does not apply any optimizations to the circuit. Since Qiskit targets lower-level transpilation to specific hardware or quantum devices and not high-level optimizations, its focus is on optimizations different from ours. In contrast, our compiler has no transpilation capabilities and cannot target any specific hardware. In turn, the optimizations and overall capabilities of Qiskit and our compiler are complementary and can be used in tandem.

5.2 Execution Time

The second aspect we want to evaluate is the execution time of different compilation stages. Since our compiler does not implement the lexical and syntactic analysis itself but uses ANTLR to generate the lexer and parser, our evaluation focuses on the semantic analysis, code generation, and, finally, optimization. To analyze the performance, we compiled the program with the ripple-carry adder with an input of $a = \frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$ and $b = |15\rangle$. In this example, the complexity of the compiler circuit can be adjusted by changing the register sizes for a and b. For our evaluation, we used the powers of two from 16 to 1024. For each register size n, we compiled the program ten times and took the averages of the execution times for each compilation stage. The results are depicted in Fig. 5.1.

The first stage is the semantic analysis. In this stage, the source code is analyzed

Register Size n	Execution Time of Stages in ms		
	Semantic Analysis	Code Generation	Optimization
16	28.3	45.4	117
32	27.5	43.8	215.4
64	27.3	47.8	711.6
128	26.3	50.4	2292.4
256	26.2	59.7	10755.7
512	25.8	74.9	60204.7
1024	26.1	109.1	405376.6

Table 5.1: The execution times compiling a quantum ripple-carry adder with different register sizes.

for semantic errors, e.g., undeclared variables or type errors. Since the analysis is performed on the source code, the execution time does not change significantly with a change in register size, as the source code size remains unchanged. The execution times range from 25.8 ms to 28.3 ms with an average of 26.79 ms.

Next, the code generation stage generates the target code from the source code. In our example program, the source code size stays constant. Both the `MAJ` and `UMA` gates are not dependent on the register size. However, the adder gate iterates over the size of the registers with a loop statement twice. Since the loop statements are unrolled at compile time, the generated code size increases linearly with an increase in register size, and, in turn, the execution time should increase likewise. This trend is not clear for small register sizes between 16 and 64. However, for larger register sizes, a linear trend is detectable.

Lastly, the optimization stage applies optimization rules, if possible, to the generated quantum circuit and, thereby, reduces its complexity. The optimization algorithm iterates over the circuit by going through all qubits. From there, it iterates over all gates on the qubit wire. In the case that all gates operate on all qubits, the resulting circuit iteration has a complexity of $n_g \cdot n_q$ where n_g and n_q are the number of gates and qubits, respectively. Additionally, the circuit iteration is repeated as often as optimizations can be found. In the case that all gates can be optimized and are only optimized one gate at a time, the overall complexity of the optimization is $n_g^2 \cdot n_q$. Since the number of gates increases linearly with the register size in our example, the worst-case complexity for its optimization is $\mathcal{O}(n^3)$, where n is the register size. While the worst case would be a cubic increase in execution time when increasing the register size, the data shows an approximate quadratic increase, starting at 117 ms on average for a register size of 16 and going up to about 6 minutes and 45 seconds for a register size of 1024.

6 Conclusion and Future Work

In the following chapter, we conclude our thesis. First, we give a short summary of the background, presented concept, final implementation, and, lastly, evaluation of this project. Then, we discuss possible improvements to the project and future work.

6.1 Summary

For this thesis, we designed a quantum programming language with control flow primitives in superposition and implemented a corresponding compiler. This compiler covers all the typical steps, including the lexical, syntactic, and semantic analysis of the code as well as the code generation and optimization. The result is a translated OpenQASM quantum circuit description.

In the beginning, we discussed some related work and background and the topic of quantum control flow and quantum computing in general; we presented the concept of quantum control flow and its limitations. Next, we introduced the Quantum Control Machine, an instruction set architecture focused on quantum control flow with instructions similar to classical assembly language. Additionally, we discussed other low- and high-level quantum programming languages, such as the Open Quantum Assembly (OpenQASM) language. Lastly, we gave a short introduction into the compilation of programs, including the different steps and some tools to build compilers or specific compilation stages.

On the basis of the background knowledge, we discussed the concept for our quantum language with quantum control flow primitives. First, we gave a language overview with the different data types, operations, control flow statements, and other language concepts. Next, we defined a formal grammar for our language. Based on the grammar, we defined formal translation functions that translate our custom programming language to OpenQASM. Lastly, we introduced different optimization rules that we intend to apply in the code optimization stage of the compiler. Furthermore, we defined a circuit graph, which is a graphical representation of a quantum circuit or program, and discussed an algorithm to apply the optimization rules to the graph.

Next, the concept for the programming language and its translation was used to implement a compiler for the language. First, we discussed functionalities of the compiler, such as the command line interface, as well as general concepts. Next, we explained how the grammar is used to implement the lexical and syntactic analysis with the ANTLR tool. Then, we discussed the semantic analysis, including declaration analysis and type checking, as well as general error handling in the compiler. This is followed by a discussion of the code generation stage and an example translation.

Lastly, we explained the optimization implementation based on the circuit graph and discussed some additional algorithms for constructing and translating the graph.

Finally, in the evaluation we analyzed two aspects of our compiler, the optimizations performed by the compiler and the execution times of the different compilation stages. The quantum ripple-carry adder was implemented in Luie and used with different input values for the evaluation. In the case of classical inputs, the optimization stage could reduce the circuit to only the necessary gates to initialize the result of the addition. While the optimization was not as effective for inputs in superposition, depending on the size of the registers and overall inputs, the circuit could be reduced significantly. Next, the performance of the compilation was evaluated with the same example algorithm with differing register sizes. While the semantic analysis stage did not take longer with increasing register sizes, the time to generate the target code increased linearly. Similarly, the optimization stage showed a quadratic increase in execution time when increasing the register size.

6.2 Improvements and Future Work

In our project we designed a quantum programming language and implemented a corresponding compiler with all basic elements, lexical, syntactic, and semantic analysis, as well as code generation and optimization. However, the evaluation revealed aspects that can be improved. Furthermore, there is also room for further improvements to aspects of the language and compiler that were not considered in the evaluation.

The first aspect that was evaluated was the optimizations performed on the generated code. Our current implementation focuses on high-level optimizations to the quantum circuit. Therefore, the compiler can work in tandem with the Qiskit kit but, in turn, is less useful as a standalone tool and is best utilized as part of a toolchain. A possible improvement is the addition of more hardware-focused features such as transpilation and compilation to concrete devices. Furthermore, with a more hardware-specific focus, the optimization rules can be extended with more hardware-specific rules. Additionally, the set of optimizations can be expanded to include more high-level ones. For example, when the value of a quantum state is discarded, it is implicitly measured. In our language, this happens each time a qubit or register is no longer in scope. Therefore, the compiler could automatically reuse qubits or registers that are out of scope or no longer used, as suggested in Sec. 3.1.5, by measuring and resetting them; in turn, the qubit count of the circuit is reduced. However, this does not automatically perform uncomputations, and they need to be performed manually by the user, as is the case currently.

Secondly, besides the optimization rules, we evaluated the performance of the different compilation stages. While the semantic analysis and code generation stages showed, at worst, linear time complexity, the optimization stage grew approximately quadratically with the input size. Therefore, the optimization stage is inefficient for a large number of gates or qubits. To improve the performance of the compiler, the first focus should be to optimize the optimization stage. One suggestion to improve the

performance would be to parallelize the iteration over the qubit wires, with each wire being scanned for optimizations by a separate thread. However, while this approach should reduce the execution time of the optimization significantly, the resulting algorithm would be more complex; new aspects such as race conditions need to be considered, and, in turn, the quantum circuit graph needs to be protected from being modified by multiple threads at the same time. This could be achieved by, e.g., a mutex. Additionally, currently each optimization rule is a static object. When the optimization process is running on multiple threads, the intermediate values of the rule may be overwritten. Therefore, either a new rule object needs to be created for each thread, increasing the used memory, or the rules need to recalculate the values, impacting the performance of the program.

Besides the evaluated aspects, there are also other parts that can be improved. Currently, a weak aspect of the language is the typecasting. All elements of an expression are of the same type, and, if they are not, they are cast implicitly to the target type. For example, when an identifier that is a constant double with a value of 9.9 is used in an expression for the size of a register, by definition an integer expression, the value is cast to be an integer with a value of 9. While the behavior is consistent, it differs from most other strongly typed languages where floating point values need to be explicitly cast to an integer type. In turn, the language could be expanded to include not only implicit but also explicit casting.

Furthermore, the range of predefined functions, provided by the language, is currently limited to only a handful of functions. These could be expanded by additional functions such as sine and cosine or even mathematical constants such as π and Euler's number e. Moreover, the language could be extended to provide functionalities for users to define their own functions with macros or lambda functions. With such functionality, often-used and more complex expressions could be expressed as custom functions and reused.

Lastly, while our language can be used to express any quantum circuit or algorithm, most quantum languages offer functionalities that provide more control over the quantum circuit. One example is explicit measurements. Our language only provides implicit measurements, i.e., all qubits and registers are automatically measured at the end of the circuit, and their results are written to corresponding qubits. However, most quantum languages provide the user with the ability to declare classical bits and registers, explicitly measure qubits and quantum registers, and write the measurement results to the corresponding classical data type. With explicit measurements, the user has more control over when measurements take place and what qubits need to be measured. For example, many algorithms use auxiliary qubits that do not need to be measured. Furthermore, with explicit measurements, the user can measure and reuse qubits for other purposes and, in turn, reduce the overall qubit count. Lastly, with explicit measurement, the measurement results can be used to conditionally execute parts of a circuit; this is required for, e.g., quantum teleportation.

Bibliography

[Aaby03] Anthony A. Aaby. *Compiler construction using flex and bison.* 2003.

[AlGr05] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 249–258. IEEE, 2005. ISBN 0-7695-2266-1. doi: 10.1109/LICS.2005.1.

[AGY07] Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible machine code and its abstract processor architecture. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *Computer Science – Theory and Applications*, volume 4649 of *Lecture Notes in Computer Science*, pages 56–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74509-9. doi: 10.1007/978-3-540-74510-5_9.

[BGB*18] Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. Encoding electronic spectra in quantum circuits with linear t complexity. *Physical Review X*, 8(4), 2018. doi: 10.1103/PhysRevX.8.041015.

[BFA22] Medina Bandic, Sebastian Feld, and Carmen G. Almudever. Full-stack quantum computing systems in the nisq era: algorithm-driven and hardware-aware compilation techniques. In Cristiana Bolchini, editor, *Proceedings of the 2022 Conference et Exhibition on Design, Automation et Test in Europe*, ACM Conferences, pages 1–6. European Design and Automation Association, Leuven,Belgium, 2022. ISBN 978-3-9819263-6-1. doi: 10.23919/DATE54114.2022.9774643.

[Benn73] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973. doi: 10.1147/rd.176.0525.

[BeLa17] Daniel J. Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017. doi: 10.1038/nature23461.

[BeVa93] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In Rao Kosaraju, David Johnson, and Alok Aggarwal, editors, *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing - STOC '93*, pages 11–20. ACM Press, New York, New York, USA, 1993. ISBN 0897915917. doi: 10.1145/167088.167097.

[BBGV20] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300. ACM, New York, NY, USA, 2020. ISBN 9781450376136. doi: 10.1145/3385412.3386007.

[BrBr02] Jean-Luc Brylinski, Ranee Brylinski, Ranee K. Brylinski, and Goong Chen. Universal quantum gates. In Goong Chen and Ranee Brylinski, editors, *Mathematics of Quantum Computation*, volume 20022356 of *Computational Mathematics*. Chapman and Hall/CRC, 2002. ISBN 978-1-58488-282-4. doi: 10.1201/9781420035377.

[Copp02] D. Coppersmith. An approximate fourier transform useful in quantum factoring, 2002. doi: 10.48550/arXiv.quant-ph/0201067.

[CJA*22] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022. doi: 10.1145/3505636.

[CBSG17] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. doi: 10.48550/arXiv.1707.03429.

[CDKM04] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit, 2004. doi: 10.48550/arXiv.quant-ph/0410184.

[Oliv07] José de Oliveira Guimarães. Learning compiler construction by examples. *ACM SIGCSE Bulletin*, 39(4):70–74, 2007. doi: 10.1145/1345375.1345418.

[DeRe74] Franklin L. DeRemer. Lexical analysis. In F. L. Bauer, F. L. de Remer, M. Griffiths, U. Hill, J. J. Horning, C. H. A. Koster, W. M. McKeeman, P. C. Poole, W. M. Waite, and J. Eickel, editors, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 109–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974. ISBN 978-3-540-06958-4. doi: 10.1007/978-3-662-21549-4_5.

[DeJo92] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992. doi: 10.1098/rspa.1992.0167.

[DMN13] Simon J. Devitt, William J. Munro, and Kae Nemoto. Quantum error correction for beginners. *Reports on progress in physics. Physical Society (Great Britain)*, 76(7):076001, 2013. doi: 10.1088/0034-4885/76/7/076001.

[Dijk75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975.

[DiCh20b] Yongshan Ding and Frederic T. Chong. Introduction. In Yongshan Ding and Frederic T. Chong, editors, *Quantum Computer Systems*, Synthesis Lectures on Computer Architecture, pages 3–12. Springer International Publishing, Cham, 2020. ISBN 978-3-031-00637-1. doi: 10.1007/978-3-031-01765-0_1.

[DiCh20c] Yongshan Ding and Frederic T. Chong. Quantum application design. In Yongshan Ding and Frederic T. Chong, editors, *Quantum Computer Systems*, Synthesis Lectures on Computer Architecture, pages 55–70. Springer International Publishing, Cham, 2020. ISBN 978-3-031-00637-1. doi: 10.1007/978-3-031-01765-0_3.

[DiCh20] Yongshan Ding and Frederic T. Chong. *Quantum Computer Systems*. Springer International Publishing, Cham, 2020. ISBN 978-3-031-00637-1. doi: 10.1007/978-3-031-01765-0.

[DiCh20a] Yongshan Ding and Frederic T. Chong. Think quantumly about computing. In Yongshan Ding and Frederic T. Chong, editors, *Quantum Computer Systems*, Synthesis Lectures on Computer Architecture, pages 13–54. Springer International Publishing, Cham, 2020. ISBN 978-3-031-00637-1. doi: 10.1007/978-3-031-01765-0_2.

[DoSt99] Charles Donnelly and Richard Stallman. *Bison manual: The YACC-compatible parser generator, 3 November 1999, Bison Version 1.29*. Free Software Foundation, Boston, Mass., 1999. ISBN 1-882114-44-2.

[Drap00] Thomas G. Draper. Addition on a quantum computer, 2000. doi: 10.48550/arXiv.quant-ph/0008033.

[FNML21] Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li. Quantum circuit optimization with deep reinforcement learning, 2021. doi: 10.48550/arXiv.2103.07585.

[GFH82] Mahadevan Ganapathi, Charles N. Fischer, and John L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys*, 14(4):573–592, 1982. doi: 10.1145/356893.356897.

[GaCh11] Juan Carlos Garcia-Escartin and Pedro Chamorro-Posada. Equivalent quantum circuits, 2011. doi: 10.48550/arXiv.1110.2998.

[Gref99] Gregory Grefenstette. Tokenization. In Nancy Ide, Jean Véronis, and Hans van Halteren, editors, *Syntactic Wordclass Tagging*,

volume 9 of *Text, Speech and Language Technology*, pages 117–133. Springer Netherlands, Dordrecht, 1999. ISBN 978-90-481-5296-4. doi: 10.1007/978-94-015-9273-4_9.

[HHHH09] Ryszard Horodecki, Paweł Horodecki, Michał Horodecki, and Karol Horodecki. Quantum entanglement. *Reviews of Modern Physics*, 81(2):865–942, 2009. doi: 10.1103/RevModPhys.81.865.

[HuLe99] J. C. Huang and T. Leng. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122)*, pages 244–248. IEEE Comput. Soc, 1999. ISBN 0-7695-0122-2. doi: 10.1109/ASSET.1999.756775.

[Inou24] Jun Inoue. Quantum programming without the quantum physics, 2024. doi: 10.48550/arXiv.2408.16234.

[JTK*24] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with qiskit, 2024. doi: 10.48550/arXiv.2405.08810.

[Jozs05] Richard Jozsa. An introduction to measurement based quantum computation, 2005. doi: 10.48550/arXiv.quant-ph/0508124.

[KMO*23] Fabian Kreppel, Christian Melzer, Diego Olvera Millán, Janis Wagner, Janine Hilder, Ulrich Poschinger, Ferdinand Schmidt-Kaler, and André Brinkmann. Quantum circuit compiler for a shuttling-based trapped-ion quantum computer. *Quantum*, 7:1176, 2023. doi: 10.22331/q-2023-11-08-1176.

[Land61] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961. doi: 10.1147/rd.53.0183.

[LPM*24] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. Quarl: A learning-based quantum circuit optimizer. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):555–582, 2024. doi: 10.1145/3649831.

[LBZ21] Ji Liu, Luciano Bello, and Huiyang Zhou. Relaxed peephole optimization: A novel compiler optimization for quantum circuits. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 301–314. IEEE Press, [S.l.], 2021. ISBN 978-1-7281-8613-9. doi: 10.1109/CGO51591.2021.9370310.

[LoCh19] Guang Hao Low and Isaac L. Chuang. Hamiltonian simulation by qubitization. *Quantum*, 3:163, 2019. doi: 10.22331/q-2019-07-12-163.

[MVDJ18] Vasileios Mavroeidis, Kamer Vishi, Mateusz D., and Audun Jøsang. The impact of quantum computing on present cryptography. *International Journal of Advanced Computer Science and Applications*, 9(3), 2018. doi: 10.14569/IJACSA.2018.090354.

[McKe65] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965. doi: 10.1145/364995.365000.

[MHH19] Gary J. Mooney, Charles D. Hill, and Lloyd C. L. Hollenberg. Entanglement in a 20-qubit superconducting quantum computer. *Scientific reports*, 9(1):13465, 2019. doi: 10.1038/s41598-019-49805-7.

[Niel06] Michael A. Nielsen. Cluster-state quantum computation. *Reports on Mathematical Physics*, 57(1):147–161, 2006. doi: 10.1016/S0034-4877(06)80014-5.

[PaQu95] T. J. Parr and R. W. Quong. Antlr: A predicated– ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. doi: 10.1002/spe.4380250705.

[PaFi11] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the antlr parser generator. *ACM SIGPLAN Notices*, 46(6):425–436, 2011. doi: 10.1145/1993316.1993548.

[PeMa02] SIMON PEYTON JONES and SIMON MARLOW. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, 2002. doi: 10.1017/S0956796802004331.

[Pres18] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018. doi: 10.22331/q-2018-08-06-79.

[RDB*22] Roman Rietsche, Christian Dremel, Samuel Bosch, Léa Steinacker, Miriam Meckel, and Jan-Marco Leimeister. Quantum computing. *Electronic Markets*, 32(4):2525–2536, 2022. doi: 10.1007/s12525-022-00570-y.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. doi: 10.1145/359340.359342.

[RLB*24] Francisco J. R. Ruiz, Tuomas Laakkonen, Johannes Bausch, Matej Balog, Mohammadamin Barekatain, Francisco J. H. Heras, Alexander Novikov, Nathan Fitzpatrick, Bernardino Romera-Paredes, John van de Wetering, Alhussein Fawzi, Konstantinos Meichanetzidis, and Pushmeet Kohli. Quantum circuit optimization with alphatensor, 2024. doi: 10.48550/arXiv.2402.14396.

[SWW*88] V. Seshadri, S. Weber, D. B. Wortman, C. P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In R. L. Wexelblat, editor, *Proceedings*

of the ACM SIGPLAN 1988 conference on Programming language design and implementation, pages 233–240. ACM, New York, NY, USA, 1988. ISBN 0897912691. doi: 10.1145/53990.54013.

[Shor97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. doi: 10.1137/S0097539795293172.

[TvS82] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using peephole optimization on intermediate code. *ACM Transactions on Programming Languages and Systems*, 4(1):21–36, 1982. doi: 10.1145/357153.357155.

[TGS22] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. Understanding and exploiting optimal function inlining. In Babak Falsafi, Michael Ferdman, Shan Lu, and Tom Wenisch, editors, *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 977–989. ACM, New York, NY, USA, 2022. ISBN 9781450392051. doi: 10.1145/3503222.3507744.

[TAG12] Michael Kirkedal Thomsen, Holger Bock Axelsen, and Robert Glück. A reversible processor architecture and its reversible logic design. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Alexis de Vos, and Robert Wille, editors, *Reversible Computation*, volume 7165 of *Lecture Notes in Computer Science*, pages 30–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-29516-4. doi: 10.1007/978-3-642-29517-1_3.

[VSSD07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers: Principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd ed. edition, 2007. ISBN 0-321-48681-1.

[Wait74] W. M. Waite. Semantic analysis. In F. L. Bauer, F. L. de Remer, M. Griffiths, U. Hill, J. J. Horning, C. H. A. Koster, W. M. McKeeman, P. C. Poole, W. M. Waite, and J. Eickel, editors, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 157–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974. ISBN 978-3-540-06958-4. doi: 10.1007/978-3-662-21549-4_8.

[WeZa91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991. doi: 10.1145/103135.103136.

[Wino78] S. Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, 32(141):175–199, 1978. doi: 10.1090/S0025-5718-1978-0468306-4.

[Ying11] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems*, 33(6):1–49, 2011. doi: 10.1145/2049706.2049708.

[YYF12] Mingsheng Ying, Nengkun Yu, and Yuan Feng. Defining quantum control flow, 2012. doi: 10.48550/arXiv.1209.4379.

[YuCa22] Charles Yuan and Michael Carbin. Tower: data structures in quantum superposition. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):259–288, 2022. doi: 10.1145/3563297.

[YVC24] Charles Yuan, Agnes Villanyi, and Michael Carbin. Quantum control machine: The limits of control flow in quantum programming. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1–28, 2024. doi: 10.1145/3649811.

[ZLY17] Ye Zhang, Yuliang Lu, and Bin Yang. Parsing statement list program using flex and bison. In *2017 First International Conference on Electronics Instrumentation & Information Systems (EIIS)*, pages 1–4. IEEE, 2017. ISBN 978-1-5386-0843-2. doi: 10.1109/EIIS.2017.8298547.

List of Figures

List of Tables

List of Algorithms

A Translations

$$
\begin{aligned}
nControl &: QubitArgument \times QASM \to QASM \\
nControl(qArg, qStm_1 \dots qStm_n) &= nControl(qArg, qStm_1) \\
&\quad \dots \\
&\quad nControl(qArg, qStm_n) \\
nControl(qArg, qubitDcl) &= qubitDcl
\end{aligned}
$$

$$
\begin{aligned}
&nControl(qArg, && && id\ qArg_1, \dots, qArg_{n'};) = \\
&\texttt{negctrl(1)} && && @\ id\ qArg, qArg_1, \dots, qArg_{n'}; \\
&nControl(qArg, \texttt{negctrl}(n) && && @\ id\ qArg_1, \dots, qArg_{n'};) = \\
&\texttt{negctrl}(n+1) && && @\ id\ qArg, qArg_1, \dots, qArg_{n'}; \\
&nControl(qArg, \texttt{ctrl}(n) && && @\ id\ qArg_1, \dots, qArg_{n'};) = \\
&\texttt{ctrl(n)} && @\ \texttt{negctrl}(1) && @\ id\ qArg_1, \dots, qArg_n, qArg, \dots, qArg_{n'}; \\
&nControl(qArg, \texttt{ctrl}(n_1) && @\ \texttt{negctrl}(n_2) && @\ id\ qArg_1, \dots, qArg_{n'};) = \\
&\texttt{ctrl}(n_1) && @\ \texttt{negctrl}(n_2+1) && @\ id\ qArg_1, \dots, qArg_{n_1}, qArg, \dots, qArg_{n'};
\end{aligned}
$$

B Grammar Terminals

```
1
2  CONSTANTGATE : XGATE | CXGATE | CCXGATE
3               | ZGATE | YGATE  | HGATE;
4  PARAMETERIZEDGATE    : PHASEGATE;
5  fragment XGATE      : 'x';
6  fragment CXGATE     : 'cx';
7  fragment CCXGATE    : 'ccx';
8  fragment ZGATE      : 'z';
9  fragment YGATE      : 'y';
10 fragment HGATE      : 'h';
11 fragment PHASEGATE : 'p';
12
13 FUNCTION : SIZEOF | POWER | MIN | MAX;
14 fragment SIZEOF : 'sizeof';
15 fragment POWER  : 'power';
16 fragment MIN    : 'min';
17 fragment MAX    : 'max';
18
19 SKIPSTAT        : 'skip';
20
21 TYPE : 'int'| 'uint' | 'double';
22
23 GATEKEYWORD      : 'gate';
24 REGISTERKEYWORD  : 'qubit';
25 CONSTANTKEYWORD  : 'const';
26 RANGE            : 'range';
27 IF               : 'qif';
28 ELSE             : 'else';
29 DO               : 'do';
30 END              : 'end';
31 FOR              : 'for';
32 IN               : 'in';
33
34 INTEGER    : [1-9] [0-9]* | '0';
35 IDENTIFIER : [a-zA-Z_] [a-zA-Z_0-9]*;
36 COMMENT    : ( '//' ~[\r\n]* | '/*' .*? '*/' ) -> skip;
37 SPACE      : [ \t\r\n\u000C] -> skip;
```

C Evaluation

C.1 Majority and Unmajority Gates

```
gate majority(a, b, c) do
    qif c do
        x b;
        x a;
    end
    qif a do
        qif b do
            x c; end end
end
```

Figure C.1: The `MAJ` gate that used in the quantum ripple-carry adder.

```
gate unmaj(a, b, c) do
    qif a do
        qif b do
            x c; end
    end
    qif c do x a; end

    qif a do x b; end
end
```

Figure C.2: The `UMA` gate that used in the quantum ripple-carry adder.

C.2 Classical Inputs Optimized

```
OPENQASM 3.0;
include "stdgates.inc";
qubit[4] id0;
qubit[4] id1;
qubit id3;
x id0[0];
x id3;
bit[4] id0_measurement;
measure id0 -> id0_measurement;
bit[4] id1_measurement;
measure id1 -> id1_measurement;
bit id3_measurement;
measure id3 -> id3_measurement;
```

C.3 Inputs in Superposition Optimized

```
OPENQASM 3.0;
include "stdgates.inc";
qubit[4] id0;
qubit[4] id1;
qubit id2;
h id0[0];
ctrl(1) @ x id0[0], id0[1];
ctrl(1) @ x id0[0], id1[0];
ctrl(1) @ x id0[0], id2;
ctrl(2) @ x id2, id1[0], id0[0];
ctrl(1) @ x id0[1], id1[1];
ctrl(1) @ x id0[1], id0[0];
ctrl(2) @ x id0[0], id1[1], id0[1];
ctrl(1) @ x id0[1], id1[2];
ctrl(2) @ x id0[0], id1[1], id0[1];
ctrl(1) @ x id0[1], id0[0];
ctrl(1) @ x id0[0], id1[1];
ctrl(2) @ x id2, id1[0], id0[0];
ctrl(1) @ x id0[0], id2;
ctrl(1) @ x id2, id1[0];
x id1[3];
bit[4] id0_measurement;
measure id0 -> id0_measurement;
bit[4] id1_measurement;
measure id1 -> id1_measurement;
bit id2_measurement;
measure id2 -> id2_measurement;
```