Intel® Quantum SDK Functional Language Extension for Quantum

Developer Guide and Reference

February 22, 2024

Release Version 1.1

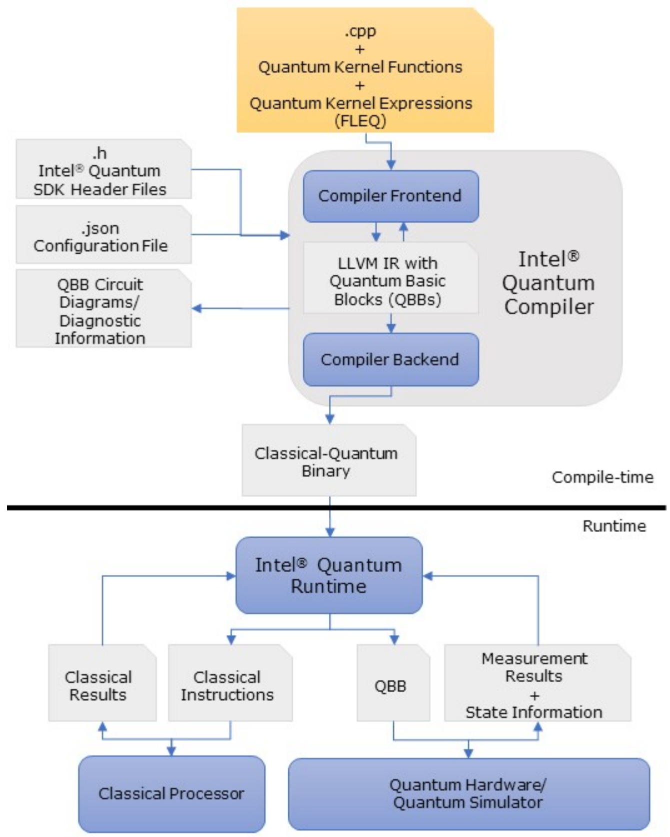
Contents:

1.0 Introduction

1.1 Motivation

1.1.1 Recap of the Intel® Quantum Software Development Kit language

The Intel® Quantum Software Development Kit (SDK) is an extension of C++ with a full LLVM-based C++ compiler, augmented to compile quantum-classical hybrid programs using quantum hardware, simulator, or other quantum backend as an accelerator. The backend is controlled by imperative gate calls inside a **quantum kernel function**—a C++ function with the `quantum_kernel` function specifier. Quantum kernel functions act as containers for quantum gate calls, analogous to a quantum circuit. These functions are compiled into **quantum basic blocks** (QBBs)—sequential lists of quantum instructions—which are dispatched to the backend when the `quantum_kernel` function is called. The compiler and runtime also define and manage the interfaces between (both classical and quantum) data and instructions so as to be nearly invisible to the user.



This style of using imperative function calls as containers for quantum gates is in contrast to a common approach for quantum programming using circuit generation frameworks. Quantum circuit generation frameworks use an object-oriented programming (OOP) paradigm, which allow users to construct circuit objects populated by gate objects. During classical runtime, a compiler or transpiler converts these circuit objects into a format readable by a qubit chip or simulator.

Circuit generation languages are extremely modular in that they allow programmers to construct and manipulate circuit objects as first class values. For example, many circuit generation languages support transformations that invert an entire unitary circuit by inverting each gate and reversing their order. Implementing such a transformation for `quantum_kernel` functions would require modifying the compiler directly, and is thus not very accessible for users.

The Intel® Quantum SDK does not adopt the circuit generation paradigm, because embedding a circuit generation framework inside C++ would compromise the strict compiler guarantees maintained by the SDK needed to interface seamlessly with the quantum runtime and backend. However, the SDK does support a collection of features, known collectively as the Functional Language Extension for Quantum (FLEQ), that provide many of the benefits of circuit generation frameworks while preserving these compiler guarantees. To understand how FLEQ operates, the first step is to unravel the tension between compiler modularity and runtime guarantees by understanding the distinction between **compile-time** and **runtime** in the context of the SDK.

1.1.2 Compile-time vs. runtime

A compile-time construct, abstraction, or object is one that is completely resolved by the compiler such that the final binary has no trace of the construct. Two examples of compile-time constructs in C++ are classes (which are elaborated into structs and global functions in the compiler) and templates. In both cases, at some point in the compiler intermediate representation (IR), all trace of the construct has been removed. In principle, equivalent IR could have been generated by more rudimentary means only using the C language. Other terms associated with compile-time constructs are "static", "resolvable" or "a priori", i.e. a variable might be resolvable or known statically or a priori.

A runtime construct or object is one in which the final compiled binary does maintain some signature of the construct, or where the construct must be computed or known when the binary is executed. An example of a runtime construct in C++ is runtime polymorphism, where several versions of the same function or method might exist, and the decision of which should be executed may depend on runtime factors. As such, every version of that function must be compiled to binary with the addition of a `vtable` to allow for runtime selection. Many C++ standard library containers are also runtime constructs as aspects like size and contents are not known a priori. Other terms associated with runtime constructs are "dynamic" or "unresolvable (by the compiler)."

Quantum kernel functions contain examples of both compile-time and runtime constructs. Loops and branches using quantum gates and qubit gate arguments are compile-time constructs as the compiler must fully resolve these inside each `quantum_kernel` function. If these were runtime constructs, the runtime would have to take on prohibitive costs such as performing on-the-fly qubit routing. On the other hand, classical parameters to gate arguments such as rotation angles are runtime variables as they can be handled by the quantum runtime library with little to no overhead for the quantum backend.

1.1.3 What is the Functional Language Extension for Quantum (FLEQ)?

Many of the limitations of the Intel® Quantum SDK can be traced back to the compile-time versus runtime constraints imposed by quantum hardware. For example, top-level `quantum_kernel` functions can not have qubit arguments, as all qubit references must be resolved at compile time. Moreover, the SDK does not enable meta-transformations on `quantum_kernel` functions because they themselves are not objects as in circuit generation languages.

However, OOP-style circuit generation is not a solution either. Member methods that manipulate circuit classes in C++ are runtime constructs, and have side effects that can be all but impossible to infer from IR. Circuit objects in C++ thus could not be transformed into QBBs at compile-time, in which case the SDK would be unable to meet compile-time and runtime requirements for issuing instructions to the quantum backend.

Seasoned C++ programmers might suggest adding modularity by passing quantum kernel functions via function pointers and lambdas (anonymous functions). This approach is in line with the spirit of the SDK, but must be done in such a way that the compiler can reason about the transformations and meet compile-time constraints.

To solve these problems, this document introduces the **Functional Language Extension for Quantum** (FLEQ) as a feature set for the Intel® Quantum Compiler (IQC). FLEQ allows for flexible, modular development of complex quantum logic while maintaining the compile-time constraints needed to generate QBBs. It is compatible with all other features of the SDK, and enhances the SDK's seamless quantum-classical interfaces and efficient runtime execution. FLEQ adapts a functional programming paradigm that treats quantum kernels as first-class, immutable constructs that can be passed into and out of functions to facilitate robust, expressive, and easy-to-use compile-time reasoning.

FLEQ consists of:

1. An immutable type `QExpr` or **quantum kernel expression** that acts like a function pointer or lambda for quantum kernels.
2. A set of compile-time methods for constructing quantum kernel expressions, with pure functional APIs that are guaranteed to have no side effects.
3. Features that alleviate many of the pain points to modular quantum code development with the SDK, including but not limited to:
   i. Built-in meta-transformations such as unitary inversion and both unitary and classical control;
   ii. Support for custom generic and complex meta-transformations;
   iii. Adaptable and reusable submodule libraries using compile-time recursion and compile-time and runtime branching;
   iv. Support for algorithms and problem instances that abstract away gate-level implementation details; and
   v. Compile-time and runtime debugging features.

In version 1.1 of the Intel® Quantum SDK, FLEQ is in its beta release and is still being actively developed. Bug reports, feature requests and general feedback are much appreciated. See Support for more details.

1.2 Basic concepts

1.2.1 Quantum kernel expressions (`QExpr`)

The central component of FLEQ is the **quantum kernel expression**, or `QExpr`. A value of type `QExpr` is an immutable compile-time representation of a block of quantum logic, as well as associated classical instructions. A `QExpr` value is a quantum program that has not yet been issued to the backend; it can be thought of as an unspecified or opaque function pointer to a `quantum_kernel` function.

To see the difference between a `QExpr` value and a `quantum_kernel` function, consider a quantum block that prepares a single qubit in the $|+\rangle$ basis. As a quantum kernel function, this block might look like:

```
qbit q;
quantum_kernel void prep_plus_qk() {
    PrepZ(q);
    H(q);
}
```

The same block of quantum instructions can be represented as a quantum kernel expression by writing a function that returns a `QExpr` value:

```
QExpr prep_plus_qexpr(qbit& q) {
    return qexpr::_PrepZ(q) + qexpr::_H(q);
}
```

Syntax aside, (see **Features** for details) these two appear very similar, but there is a fundamental difference. The `quantum_kernel` function `prep_plus_qk` is a fully specified sequence of quantum instructions; as such, calling the function issues those instructions in the form of a QBB. The quantum kernel expression that is **returned** by `prep_plus_qexpr(q)` ostensibly represents the same set of quantum instructions, but calling the function alone will **not** issue those instructions. Instead, the `QExpr` returned by `prep_plus_qexpr(q)` is just a **representation** of those instructions that can be manipulated by the programmer.

For example, the quantum kernel expression `prep_plus_qexpr(q)` can be appended with a `Z` gate to take the plus state $|+\rangle$ to the minus state $|-\rangle$: `prep_plus_qexpr(q) + _Z(q)`. Furthermore, this $|+\rangle$-to-$|-\rangle$ transformation can be generalized to any quantum kernel expression, as shown by a function of the form:

```
QExpr appendZ(QExpr e, qbit& q) {
    return e + qexpr::_Z(q);
}
```

Then, the $|-\rangle$ preparation sequence is the `QExpr` value returned by `appendZ(prep_plus_qexpr(q), q)`. Note that all `QExpr` values are immutable, so it is not that `appendZ` modifies its argument, but rather returns a new unique `QExpr` value.

1.2.2 Evaluating quantum kernel expressions

Once a `QExpr` value has been constructed, the programmer must issue the instructions it represents to the quantum backend. Doing so is referred to as the **evaluation** of a `QExpr` and is achieved by one of two evaluation functions: `void eval_hold(QExpr)` or `void eval_release(QExpr)`. For our examples above, the following `main` function issues two identical state preparation sequences to the backend:

```
qbit q;
quantum_kernel void prep_plus_qk() { ... }
QExpr prep_plus_qexpr(qbit& q) { ... }

int main() {
    ...                    // Initialization of the Intel Quantum SDK runtime omitted
    prep_plus_qk();        // Invoke the qubit chip by calling a quantum_kernel function
    eval_hold(prep_plus_qexpr(q)); // Invoke the qubit chip by evaluating a QExpr value
    return 1;
}
```

The call to `eval_hold` tells the compiler that its `QExpr` argument should be compiled and sent to the quantum runtime at this point in the code. The difference between `eval_hold` and `eval_release` is analogous to the `release_quantum_state` directive (see the Developer Guide and Reference (Language Extensions)): `eval_hold(e)` guarantees that the quantum state is preserved after executing the QBB produced by compiling `e`; `eval_release(e)` makes no guarantees of the quantum state after execution, and should be used when the user is only interested in measurement results.

1.2.3 Compile-time lists (`QList` and `DataList`)

FLEQ enables modular quantum programming by allowing users to build quantum kernel expressions whose contents depend on compile-time arguments. For example, suppose a programmer wants to prepare each qubit in an array into a $|+\rangle$ state. To do this with `quantum_kernel` functions, the programmer would need make extensive use of C++ templates, since qubit arrays and loop bounds must both be determined at compile-time. However, using quantum kernel expressions, a user can write a function that takes as input a compile-time qubit list and returns a `QExpr`. This compile-time qubit list (`QList`) and the corresponding type of compile-time strings (`DataList`) give programmers flexibility while ensuring the compiler has what it needs to interact with the back-end.

A **qubit list** or `QList` is a compile-time wrapper around static `qbit` arrays. `QList` values can be concatenated to form a new single `QList`, or sliced into sub-lists to form arbitrary orderings of qubits. Qubits can be individually addressed via the subscript (`[]`) operator, and the size of the array can be resolved at compile-time using member function `size()`. A `QList` can be declared using the `listable` macro, as in

```
const int N = 5;
qbit listable(qs, N);
```

A `DataList` value is compile-time string; like a `QList`, it is a wrapper around around statically defined `char` arrays, and can be joined and sliced into form arbitrary permutations of the string. Data lists also support a variety of substring search functions; type conversion to `int`, `bool` and `double` types; and string comparison. The `DataList` feature can even be used to develop domain-specific languages (DSLs) that allow for higher-level representations that abstract away gate-level implement details. See **DataList** and **Domain-specific languages using FLEQ** for more details.

1.2.4 Getting started

Using FLEQ requires the `quintrinsics.h` header that is required for the base use of the SDK. In addition, the three core features of FLEQ are provided by three additional header files:

```
#include <clang/Quantum/quintrinsics.h> // always required
#include <clang/Quantum/qexpr.h>        // required for QExpr features
#include <clang/Quantum/qlist.h>        // required for QList features
#include <clang/Quantum/datalist.h>     // required for DataList features
```

Programs using FLEQ are compiled using the same command-line flags and arguments as the base SDK, with some additional optional flags specific to FLEQ (as described in the relevant sections of **Features**). All other features and flags are fully compatible with FLEQ, including the use of `quantum_kernel` functions, with one exception: `quantum_kernel` functions that call basic gates cannot also contain `QExpr` evaluation calls. See **Known limitations**.

We also note here that for FLEQ-generated QBBs, there is no appreciable difference between the use of the `-O0` and `-O1` optimization flags for reasons described in **Overview of FLEQ compilation**; see also **Known limitations**.

2.0 Features

FLEQ provides three namespaces, qexpr, qlist and datalist, which are available in the header files qexpr.h, qlist.h, and datalist.h, respectively. The functions described below will be scoped by their appropriate namespace, where this scoping can be avoided using the typical C++ keywords using namespace <name>.

2.1 Basics: evals, join, identity, and QExpr-returning functions

The simplest quantum kernel expressions are basic primitives representing single quantum gates.

```
QExpr qexpr::_<g>(Args... args)
```

For each primitive gate <g>(Args...) in the Intel® Quantum SDK (see Developer Guide and Reference), there is a corresponding primitive quantum kernel expression _<g>(Args...). The arguments of _<g> are the same as those of <g>.

Other basic primitives include

```
QExpr qexpr::identity()
```

An identity or no-op quantum kernel expression.

```
QExpr qexpr::global_phase(double angle)
```

A quantum kernel expression that applies a global phase but otherwise has no effect. The distinction between identity and global_phase is primarily relevant to unitary control of a QExpr; see **Control**. The angle argument to global_phase() can be dynamic–it need not be resolvable at compile-time.

Two quantum kernel expressions e1 and e2 can be combined together in several equivalent ways:

```
QExpr qexpr::join(QExpr e1, QExpr e2)
```

Returns the sequential composition of e1 followed by e2. In other words, when evaluated, qexpr::join(e1,e2) will execute the logic associated with e1 followed by the logic associated with e2.

```
e1 + e2
```

Shorthand for qexpr::join(e1, e2).

```
e2 * e1
```

Combines the quantum kernel expressions in **composition order**, meaning that it evaluates its second argument first. Composition order is natural when thinking in terms of matrix multiplication or function composition. Equivalent to qexpr::join(e1,e2).

The third core component of QExpr functionality are the evaluation functions introduced in **Evaluating quantum kernel expressions**.

```
void qexpr::eval_hold(QExpr e)
```

Executes the quantum instructions represented by the quantum kernel expression e. It guarantees that the quantum state will be maintained after execution of the instructions.

`void qexpr::eval_release(QExpr e)`

Executes the quantum instructions represented by e under the assumption that the quantum state need not be preserved after execution. It is the direct analog to the `release_quantum_state` directive for ordinary `quantum_kernel` functions; see the Developer Guide and Reference for details.

When an evaluation function is called inside a classical function `f()`, the compiler treats `f()` itself as a `quantum_kernel` function. This helps lift some of the limitations placed on conventional quantum kernel functions. For one, multiple evaluation calls, or even a single evaluation call in the case of **Branching** and **Barriers and binding**, can result in multiple quantum basic blocks. In addition, local qubits, which can only be declared inside `quantum_kernel` functions, can now be declared in functions that make evaluation calls to quantum kernel expressions. See **Local qubits** for more details.

A quantum kernel expression can be constructed inline inside an evaluation call, such as

```
qexpr::eval_hold(qexpr::_PrepZ(q) + qexpr::_H(q));
```

Alternatively, ordinary C++ functions can return a quantum kernel expression of type `QExpr`, which can then be evaluated.

```
QExpr prep_plus_qexpr(qbit& q) {
    return qexpr::_PrepZ(q) + qexpr::_H(q);
}
int main() {
  ...
  qexpr::eval_hold(prep_plus_qexpr(q));
  ...
}
```

`QExpr`-returning functions do come with some limitations and special features:

- A function returning a `QExpr` must have a single return statement; this is enforced by the FLEQ compilation stage of the Intel® Quantum Compiler (IQC). If the user desires branching or conditional `QExpr` returns, they should use the `cIf` functionality described in **Branching**.
- Traditional C++ branching and looping is allowed under the same constraints as a `quantum_kernel` function for classical data, but generally is not encouraged. Best practice is to use `cIf` (**Branching**) and recursion (**Recursion**). There are known cases where traditional branching and FLEQ are incompatible, in particular:
  - FLEQ does not support loops that are bound by FLEQ functions such as the size of a `QList` or `DataList`. This is because classical loop unrolling comes before FLEQ processing in compilation (see **Overview of FLEQ compilation**). For example, the following function results in a compiler error:

    ```
    quantum_kernel void prepAll_qlist_BAD(qlist::QList qs) {
      for (int i=0; i<qs.size(); i++) {
        PrepZ(qs[i]);
      }
    }
    ```

- The loop above **can** be written using a global `QList` of a fixed size. For example, the following function is acceptable:

```
const int N = 5;
qbit qs[N];

quantum_kernel void prepAll_qlist_global() {
  for (int i=0; i<N; i++) {
    PrepZ(qs[i]);
  }
}
```

- Quantum kernel expressions and evaluation calls can be placed in a for-loop like `prepAll_qlist_global()`, but only inside functions annotated by the `quantum_kernel` keyword. Ordinarily C++ functions that evaluate quantum kernel expressions need not have the `quantum_kernel` keyword explicitly.

```
const int N = 5;
qbit qs[N];

quantum_kernel void prepAll_qexpr_global_eval() {
  for (int i=0; i<N; i++) {
    qexpr::eval_hold(qexpr::_PrepZ(qs[i]));
  }
}
```

- `qbit` arguments must be passed by reference (enforced by the FLEQ compilation stage of IQC).
- FLEQ values of type `QExpr`, `QList`, and `DataList` should be passed by value, as they are immutable functional data structures. This is not strictly enforced by the compiler but can result in compilation failures.
- Data intended to be treated as an output of the `QExpr` function, such as `cbit` or `bool` values or arrays, should be returned through a reference argument (**return-by-reference**), as the function must return a `QExpr` type.
- `cbit` variables and arrays populated by quantum measurements (i.e. via `_MeasZ`) in a `QExpr` are not written to until after the evaluation call resolves, unless they come before a `qexpr::fence` or `qexpr::bind` statement; see **Barriers and binding**. This has implications for classical branching with `QExpr`; see **Branching** and **Barriers and binding**.
- Consider a `QExpr`-returning function `foo` that returns-by-reference additional data, i.e. writes to some memory such as `cbit` measurement data. If a user intends for the returned data to be read by another `QExpr`-returning function `bar` in the same evaluation or return statement, then (1) `foo` and `bar` must be separated by a `qexpr::bind` statement (see **Barriers and binding**); and (2) the data must be passed to `bar` by reference. For example, the following is valid:

```
QExpr foo(cbit &write_to) { ... }
QExpr bar(cbit &read_from) { ... }

int main() {
  ...
  cbit data;
```

(continues on next page)

(continued from previous page)

```
  qexpr::eval_hold(foo(data) << bar(data));
}
```

- `QExpr`-returning function pointers can be used both as traditional C++ function arguments and template arguments to form higher-order `QExpr` transformations. The `qexpr::map` utility function is one such example; see **Higher-order functions**.

2.2 Quantum kernel function conversion

`quantum_kernel` functions can be converted to a quantum kernel expression using three methods. If the quantum kernel function is already written and one does not want to modify or copy it, one can use the following template function:

`template<auto qk_function> QExpr qexpr::convert(Args... args)`

where `qk_function` is any `quantum_kernel` function with arbitrary parameters, so long as it has a `void` return type. The argument type list `Args` must be the same as that of `qk_function` and the argument list `args` is subject to the same constraints as those imposed on `qk_function` as a `quantum_kernel` function (see the Developer Guide and Reference). An exception is that `qbit` arguments to converted `quantum_kernel` functions **are** allowed, so long as they are compile-time resolvable when evaluated; see **Local qubits**. For example:

```
const int N = 4;
qbit qs[N];

void quantum_kernel prepAll() {
    for (int i=0; i<N; i++) {
        PrepZ(qs[i]);
    }
}
void quantum_kernel bell00(qbit& a, qbit& b) {
  PrepZ(a);
  PrepZ(b);
  H(a);
  CNOT(a,b);
}

int main() {
  ...
  qexpr::eval_hold(qexpr::convert<prepAll>());
  qexpr::eval_hold(qexpr::convert<bell00>(qs[2], qs[3]));
  ...
}
```

This kind of conversion can be useful for debugging `quantum_kernel` functions; see **Debugging**.

Alternatively, a `quantum_kernel` function can be modified directly to return a quantum kernel expression by adding a return statement using the keyword `this_as_expr`. For reasons outlined in **Overview of FLEQ compilation**, any function which returns `this_as_expr` must have the added attribute `PROTECT` as enforced by the FLEQ compilation stage of IQC. For example:

```
PROTECT QExpr prepAll_qexpr() {
    for (int i=0; i<N; i++) {
        PrepZ(qs[i]);
    }
    return this_as_expr;
}
```

Functions returning `this_as_expr` are subject to the constraints of both `quantum_kernel` functions and `QExpr`-returning functions. It is valid to mix `this_as_expr` with other `QExpr` expressions, but care has to be taken that the outcome ordering is as expected: `this_as_expr` represents the gate sequence present in the body of the function. For example, the following two functions are logically equivalent:

```
PROTECT QExpr myRX1(qbit &q, double angle) {
  H(q);
  RZ(q, angle);
  H(q);
  return this_as_expr;
}

PROTECT QExpr myRX2(qbit &q, double angle) {
  RZ(q, angle);
  return qexpr::_H(q) + this_as_expr + qexpr::_H(q);
}
```

The final method is to construct a new function that returns a `QExpr` value made up of the same primitive gates as the target `quantum_kernel` function. Note that this kind of translation looks slightly different for `quantum_-kernel` functions that use C++ loops and branches; equivalent results can be achieved using Branching and Recursion.

2.3 Coherent transformations

A quantum kernel expression that does not use any preparation or measurement gates is called **coherent** and corresponds to a unitary transformation. There are several operations that can act on coherent quantum kernel expressions in convenient and powerful ways. If these operations are inappropriately called on non-coherent quantum kernel expressions, the FLEQ compilation stage of IQC will exit with a warning.

2.3.1 Control

Coherent qubit control is a unitary transformation that applies an input unitary conditioned on the computation eigenstate of a control qubit. Note the qubit state must be independent of the input unitary for the transformation to be well-defined (enforced by the FLEQ compilation stage of IQC).

FLEQ supports several variations on coherent qubit control:

`QExpr qexpr::control(qbit &q, bool b = true, QExpr u)`

> Returns a quantum kernel expression implementing unitary control of the unitary `u` based on the qubit `q` being in state $|b\rangle$. That is, if `b == true`, the control is on the $|1\rangle$ state and if `b == false`, the control is on the $|0\rangle$ state. The value of `b` can be dynamic i.e. it does not have to be resolved at compile-time. Its default value is `true`.

`QExpr qexpr::qIf(qbit &q, QExpr uT, QExpr uF)`

Returns a quantum kernel expression that controls the unitary uT on the condition of $q = |1\rangle$ and uF on the condition of $q = |0\rangle$.

`QExpr qexpr::control(QList qs, unsigned int ctl_on = -1, QExpr u)`

Returns a quantum kernel expression that controls the unitary u on the state of the qubits in `qs` such that u is applied to the state only if $\texttt{qs}[i] = |(\texttt{ctl\_on} << i)\%2\rangle$ for all $i < \texttt{qs.size()}$. Said differently, the binary representation of the computational state of `qs` matches the binary representation of `ctl_on` where earlier qubits in `qs` correspond to the more significant bits (big-endian). `ctl_on` can be dynamic i.e. does not have to be resolved at compile time and has a default value of -1 which for `unsigned int` translates to $2^{32} - 1 = 1111\ldots$. The size of `qs` is limited to 8 qubits as enforced by the FLEQ compilation stage of IQC. This was done since the function does not add ancilla qubits and as such, the circuit cost grows exponentially with the size of `qs`. The recursion limit can be overcome by chaining `control` calls using recursion (see **Recursion**) or with the explicit use of ancilla. See **Known limitations**.

2.3.2 Inversion

Coherent quantum kernel expressions can be inverted using the operation `invert`.

`QExpr qexpr::invert(QExpr u)`

Returns a quantum kernel expression implementing the unitary $U^\dagger$, provided u is a coherent quantum kernel expression implementing the unitary U.

`invert` has three equivalent operator overloads, `!u`, `~u` and `-u`.

2.3.3 Power

The power of a quantum kernel expression refers to repeated application of its logic on the quantum backend.

`QExpr qexpr::power(unsigned int n, QExpr e)`

If $n > 0$, returns a quantum kernel expression that joins e with itself n times. If $n = 0$, returns the identity quantum kernel expression. In these two cases, e need not be coherent.

If $n < 0$, e must be coherent, in which case `power(e,n)` is equivalent to `power(invert(e), -n)`.

Currently, n must be resolvable at compile-time, though future versions will relax this constraint.

`power` has an operator overload `e^n` which is equivalent to `qexpr::power(e, n)`.

2.4 `QList`

A quantum list of type `qlist::QList` is an immutable compile-time list for the `qbit` type. It is a C++ class that conceptually can be thought of as list of references to existing `qbit` declarations. A `QList` can be constructed around a separate `qbit` array by passing its pointer to the `QList` constructor. For convenience, both the `qbit` declaration and creation of a `QList` from it are provided by a single call to the `listable` macro:

```
#include <clang/Quantum/qlist.h>
const int N = 5;

qbit listable(qs, N);
// equivalent to:
//
// qbit qs_raw[N];
// const qlist::QList qs(qs_raw);
```

Note that with the use of `listable(<name>, N)`, `<name>` is a variable of type `QList`, and the underlying `qbit` array has the name `<name>_raw`. During compilation, for example during circuit printing, the variable is displayed as `<name>_raw`. See **Known limitations**.

A custom qubit placement as described in Developer Guide and Reference (Qubit Placement and Scheduling) can also be specified via the `listable` macro. A custom placement list using parenthesis can be passed as a third argument:

```
#include <clang/Quantum/qlist.h>
const int N = 5;

qbit listable(qs, N, (1, 3, 5, 12, 0));
// equivalent to:
//
// qbit qs_raw[N] = {1, 3, 5, 12, 0};
// const qlist::QList qs(qs_raw);
```

The same limitations on custom qubit placement apply for this macro. Most notably, custom placement can only be specified for global qubits.

The `qlist.h` library provides several operations on qubit lists.

`unsigned int qlist::QList::size()`

Returns the size of a `QList`. Compile-time resolvable.

`qbit& qlist::QList::operator[](unsigned long i)`

Index into a `QList` via operator overload, i.e. `qs[i]`. Compile-time resolvable.

`qlist::QList qlist::operator+(QList q1, QList q2)`

Concatenate two `QList` values together in sequence, i.e. `qs1 + qs2`. Compile-time resolvable.

If a user needs a `QList` at runtime, rather than at compile-time, the following function is provided:

`std::vector<std::reference_wrapper<qbit>> qlist::to_ref_wrappers(QList qs)`

Convert a `QList` into a vector of reference wrappers, used to interact with backend simulators as detailed in the Developer Guide and Reference. Note however that the `QList` argument must be compile-time resolvable and that the returned reference wrappers are available at runtime only.

Qubit lists can also be sliced into sublists, which can then be joined together in different orders via +.

`qlist::QList qlist::QList::operator()(unsigned long start, unsigned long end)`

Returns the slice of a `QList` starting at index `start` and ending at index `end-1`; i.e. `qs(start, end)` returns the slice of `qs` from `start` inclusively to end exclusively.

`qlist::QList qlist::operator>>(QList qs, unsigned long i)`

Returns the slice of qs shifted to the right by offset i i.e. `qs >> i == qs(i, qs.size())`.

`qlist::QList qlist::operator<<(QList qs, unsigned long i)`

Returns the slice of qs shifted to the left by offset i i.e. `qs << i == qs(0, qs.size()-i)`

`qlist::QList qlist::operator+(QList qs, unsigned long i)`

Returns the slice of qs shifted to the right by offset i i.e. `qs + i == qs(i, qs.size())`.

`qlist::QList qlist::operator++()`

Returns the slice of qs shifted to the right by 1 i.e. `++qs == qs(1, qs.size())`.

If a single qubit `q` is passed to a function that expects a `QList`, that qubit will be automatically converted to a `QList` of length 1 containing q. For example:

```
QExpr foo(qlist::QList qs) { ... }
int main() {
  ...
  qbit q;
  eval_hold(foo(q)); // Equivalent to eval_hold(foo(QList(q)));
}
```

2.5 Branching

In an ordinary `quantum_kernel` function, it is not possible to use classical `if` statements to decide what quantum operations to apply unless the condition can be resolved at compile-time. For example, the following code is not supported.

```
quantum_kernel void conditional_qk_FAIL(qbit &q, bool b) {
    if (b) {
        H(q);
    }
}
```

Classical control **is** natively supported by quantum kernel expressions, however, via the **classical if** or `cIf`.

`QExpr qexpr::cIf(bool b, QExpr condTrue, QExpr condFalse)`

If the classical boolean value `b` is true, execute the quantum instructions given by `condTrue`; and otherwise execute the instructions given by `condFalse`. The boolean value b may be dynamic (need not be resolvable at compile-time).

Classical control is the first indication that the `QExpr` type does not just represent a simple gate sequence. As an example, the above invalid quantum kernel function can be achieved by evaluating the following `QExpr` function:

```
QExpr conditional_qexpr_SUCCESS(qbit &q, bool b) {
    return qexpr::cIf(b, qexpr::_H(q), qexpr::identity());
}
```

Though the condition argument `b` to `cIf` can be dynamic, care should be taken with unresolved conditionals. Due to compile-time constraints, the FLEQ compilation stage of IQC will combinatorally generate all possible QBBs from unresolved branches and insert classical branching instructions to select amongst them. As a result, the number of QBBs generated by an unresolved conditional can grow exponentially, especially when used alongside `qexpr::join` or recursion (see **Recursion**). This exponential explosion can be overcome via separate evaluation calls or through the use of `qexpr::bind`; see **Barriers and binding**.

Two additional variants of `cIf` are also added for convenience:

`QExpr qexpr::cIfTrue(bool b, QExpr condTrue)`

> If the classical boolean value `b` is true, return `condTrue`, and otherwise return the identity `QExpr`; i.e. equivalent to `qexpr::cIf(b, condTrue, qexpr::identity())`.

`QExpr qexpr::cIfFalse(bool b, QExpr condFalse)`

> If the classical boolean value `b` is false, return `condFalse`, and otherwise return the identity `QExpr`; i.e. equivalent to `qexpr::cIf(b, qexpr::identity(), condFalse)`.

2.6 Recursion

With the introduction of branching, function recursion is now possible by providing exit conditions for recursion, i.e. by acting as a **recursion guard**. These recursive calls are the FLEQ equivalent of loops in standard C++ and represent the single most powerful tool for modular code development with FLEQ.

A quintessential example of a recursive `QExpr` function is one that iterates over all the qubits in a `QList` and applies one or more gates to each of them. For example, the following function applies a `PrepZ` gate to every qubit in a `QList`.

```
QExpr prepAll(qlist::QList qs) {
  return qexpr::cIf(qs.size() == 0,
                    qexpr::identity(),                     // qs.size() == 0
                    qexpr::_PrepZ(qs[0]) + prepAll(qs + 1) // qs.size() > 0
  );
}
```

This function uses a conditional `cIf` to determine if the `QList` is empty. If it is, it will terminate by returning `identity`. If the `QList` is non-empty, the condition resolves to the second branch, which applies `_PrepZ` on the first element and recursively calls `prepAll` to the tail of the `QList`.

For those familiar with imperative-style `for` and `while` loops, it may take some practice to master function recursion, but the results can be elegant and useful.

A recursive function is best matched conceptually to a `while` loop with a "continue" condition. For example, the general structure of a `while` loop is

```
while(cont){
  <body>
  cont = new_cont;
}
```

The analogous structure using FLEQ would be

```
QExpr while_analog(bool cont, QExpr body) {
  bool new_cont = ...;
  return qexpr::cIf(cont, while_analog(new_cont, body) * body, //cont == true
                          qexpr::identity()                    //cont == false
            );
}
```

Like the `while` loop, the `while_analog` function first checks if the `cont` condition is `true`, in which case it "executes" `body`. Then, the `while_analog` continues the loop by making a recursive call with boolean guard `new_cont`. This recursive call is parallel to the `while` loop implicitly looping back and rechecking the updated value of `cont`. When the `cont` condition is `false`, the `while_analog` function exits the recursion by returning `qexpr::identity()`.

A `for` loop can be translated into a recursive `QExpr` function in a similar way.

For nested loops, each loop will require at least one function. For example, consider a function that applies a `CNOT` gate to every unique pair of qubits in a `QList`. This will require two loops and thus two functions:

```
// This function uses the fixed argument q as the control
// and loops over each qubit in after_q as the target
QExpr CNOTOnAll_helper(qbit & q, qlist::QList after_q){
  return qexpr::cIf(after_q.size() > 0,
                    qexpr::_CNOT(q, after_q[0])
                      + CNOTOnAll_helper(q, after_q + 1), // after_q.size() > 0
                    qexpr::identity()                     // after_q.size() == 0
                    );
}

// This function loops over every qubit in qs, calling the helper
// function on it and every qubit that comes after it
QExpr CNOTOnAll(qlist::QList qs){
  qlist::QList q_after = qs + 1;
  return qexpr::cIf(qs.size() > 0,
                    CNOTOnAll_helper(qs[0], q_after)
                      + CNOTOnAll(q_after),           // qs.size() > 0
                    qexpr::identity()                 // qs.size() == 0
            );
}
```

Note that recursion does not require the function be called inside its body directly; rather, it is a matter of the function having a dependency on itself (see **Overview of FLEQ compilation**), which can occur through other functions called inside the body via mutual recursion. For example, consider the following pair of mutually recursive functions, one of which applies one sequence of gates to even qubits in a `QList`, and the other of which applies another sequence to odd qubits.

```
QExpr oddQubits(qlist::QList qs) {
  return qexpr::cIfFalse(qs.size() == 0,
                qexpr::_X(qs[0]) + evenQubits(qs+1)
  );
}

QExpr evenQubits(qlist::QList qs) {
  return qexpr::cIfFalse(qs.size() == 0,
                qexpr::_H(qs[0]) + oddQubits(qs+1)
  );
}
```

It is important to note that all recursion in quantum kernel expressions must be able to be unrolled at compile-time. This ensures that quantum kernel expressions can be compiled to quantum basic blocks as described in the **Introduction**. Practically, this means that the arguments to conditionals used as recursive guards must all be resolvable at compile-time through the FLEQ compilations unrolling mechanism. Examples of such compile-time guards include but are not limited to:

- constant integers or boolean values;
- the size of a `QList`, as in `prepAll` above; or
- the size or contents of a `DataList` (see **DataList**).

Users can set the recursion limit via a command-line flag.

`-F recursion-limit-power=<INT>`

Sets the FLEQ recursion limit to scale as a power `<INT>` of the number of global qubits; default = 1.

`-F recursion-limit=<INT>`

Sets the FLEQ recursion limit to a fixed number `<INT>`; default is the maximum of 1000 or `<Value from power>`, if set. The `recursion-limit` option overrides the `recursion-limit-power` option.

FLEQ does not currently support repeat-until-success loops, although they can be implemented by evaluating a `QExpr` inside a classical loop. For example, consider a repeat-until-success (RUS) loop that applies a quantum kernel expression op over and over until its measurement result returns 1:

```
QExpr  op(double param, bool& result);
double new_param(double old_param);

int RUS(double initial_param) {
  double param  = initial_param;
  bool   result = false;
  while (!result) {
    qexpr::eval_release(op(param, result));
    param = new_param(param);
  }
}
```

FLEQ may allow for runtime recursion in future versions.

2.7 Let/get, printing, and exiting

This section covers some useful builtin utilities for working with quantum kernel expressions.

2.7.1 Let/get

All variables of type `QExpr` are constant, which means that they cannot be assigned using ordinary C++ assignment statements. FLEQ provides several ways to assign temporary variables to help break up large quantum kernel expressions.

1. Write a function that returns a `QExpr`, as illustrated throughout this document.
2. Use `qexpr::let` and `qexpr::get` to assign a `QExpr` to a constant string name.

`void qexpr::let(const char key[], QExpr e)`

Associate a key value `key` to the quantum kernel expression `e`.

`QExpr qexpr::get(const char key[])`

Return the quantum kernel expression associated with `key`.

As an example:

```
qexpr::let("coin_toss", qexpr::_PrepZ(q) + qexpr::_H(q) + qexpr::_MeasZ(q,c));
```

The variable `coin_toss` can then be recalled later in the program with the `get` function:

```
qexpr::eval_hold(qexpr::cIfTrue(b, qexpr::get("coin_toss")));
```

The scope of a `let` call for a given key value is as local as possible. If called inside a function with no traditional C++ branching, the scope is contained within that function. Thus, if the function recurses, the definition corresponding to a given key value is as defined in that recursive iteration. For example, consider the following function:

```
QExpr recursive_let(qlist::QList qs, double angle){

  qexpr::let("rotation", qexpr::_RZ(qs[0], angle / (double)qs.size()));

  return qexpr::cIf(qs.size() > 0,
                    qexpr::get("rotation") + recursive_let(qs + 1, angle),
                    qexpr::identity()
                  );
}
```

In the above, the `_RZ` angle for the key `"rotation"` is dependent on the size of `qs` as one would expect. If the function contains traditional C++ branching, then the scope of the key value is the containing branch of the code (i.e. within the same LLVM IR Basic Block). Best practice is to not use `let` and `get` with C++ branching (see **Known limitations**).

2.7.2 Printing

Since quantum kernel expressions can become quite large, especially with recursion and branching, it is often useful to check what a `QExpr` evaluates to without having to inspect intermediate LLVM files or the final circuit diagram. Likewise, one may also want to print messages at different points in the building of quantum logic for an evaluation call. For this, FLEQ introduces two compile-time printing functions:

`QExpr qexpr::printQuantumLogic(QExpr e)`

> At compile-time, print out a representation of the quantum logic associated with the quantum kernel expression e, and then return e.

`QExpr qexpr::printDataList(datalist::DataList d, QExpr e)`

> At compile-time, print out the data list `d` as resolved during the evaluation build (see **DataList**), and return the quantum kernel expression e.

Both functions return their quantum kernel expression argument e, but they trigger a compile-time message that is added to the FLEQ compilation print buffer for the argument's evaluation call. Once that evaluation call is fully built, the print buffer is displayed. The ordering in which messages in the print buffer are displayed is a topological ordering of the print nodes in the underlying graph representation of the evaluation call (see **Overview of FLEQ compilation**). For example: nested calls to `printDataList` or `printQuantumLogic` will be displayed from the outside in:

```
qexpr::eval_hold(qexpr::printDataList("Prints First\n",
                 qexpr::printDataList("Prints Second\n",
                 qexpr::identity())));
```

However, when separate print functions are combined with a join, they are displayed in reverse order:

```
qexpr::eval_hold(qexpr::printDataList("Prints Second\n", qexpr::identity())
               + qexpr::printDataList("Prints First\n",  qexpr::identity()));
```

There are no guarantees on the order in which separate evaluation calls are built.

While `printQuantumLogic` prints a representation of the quantum logic of the passed `QExpr`, it does not capture the classical branching or any other classical structure, so each unique QBB attached to a given evaluation call is printed as a separate node. The representation used is that of the **PCOAST graph** as described in [PCOAST2023]; also see **Overview of FLEQ compilation**. The PCOAST graph is used as an intermediate representation (IR) for each generated QBB, and is synthesized into a quantum gate sequence later in IQC compilation.

A PCOAST graph is centered around Pauli operator representations of the quantum logic and as such does require some interpretation to understand. The basic features of the printed PCOAST graph for a given node are:

`QBB IR name`

> The QBB name attached to this PCOAST graph as found in the LLVM IR. This is a means to verify the classical branching is translated appropriately, although it does require the ability of the user to read and parse the textual IR; see **Debugging**.

`Qubit mapping`

Provides the mapping of declared qubits to a numerical index.

`Global Phase`

The global phase associated with the contained quantum logic.

`List of Elements`

The list of non-Clifford PCOAST nodes in the PCOAST graph in sequential order.

`End Frame`

A residual Clifford unitary applied at the end of the quantum operator encoded in a Pauli frame/tableau [PCOAST2023].

These compile-time printing features can be turned on and off via the command-line flag, `-F print=<OPT>`.

`OPT` can be one of four options:

- `always` - always print the buffer to screen for compilation failure and success
- `fail` - only print the buffer on failure to compile an evaluation call
- `success` - only print the buffer on successful compilation of an evaluation call
- `never` - never print the buffer to screen for compilation failure or success

2.7.3 Exiting

Because of its functional methodology, branching in FLEQ requires all possibilities be handled, i.e. a default outcome must be explicitly defined. In many cases, one might want that default to represent an error or undesired behavior. To this end, FLEQ introduces two functions that exit and display an error message:

`QExpr qexpr::exitAtCompile(datalist::Datalist err = "")`

When evaluated, adds `err` to the print buffer and throws a compile-time error after the evaluation is built (to fully populate the print buffer) or a different failure point is found. This is understood as a failure with respects to the print flags. In the case of `printQuantumLogic`, an `exitAtCompile` node will be appear empty and will "poison" nodes which depend on it. For example, a `qexpr::join` between `exitAtCompile` and any other `QExpr` will also appear as empty. The only exception is `qexpr::cIf` where only the poisoned branch will appear as empty.

`QExpr qexpr::exitAtRuntime(datalist::Datalist err = "")`

Returns a `QExpr` that, when encountered during runtime, throws a runtime error with the error message `err`. `exitAtRuntime` will also poison nodes which depended on it up to `qexpr::cIf`.

Both are intended to be used in conjunction with `qexpr::cIf()`. The distinction between the two is when the exit is triggered. If the branching condition is not intended to be resolved by the compiler, one should use `exitAtRuntime()` so that a quantum runtime exit call is inserted in the appropriate branch(es) along with the passed exit message to be printed upon reaching that branch at runtime. However, if one expects said condition to be resolved by the compiler, one should use `exitAtCompile`.

For example, the following function compares a character against `0`, `1`, `+`, or `-`, and prepares a qubit in the specified state. If any other character is given as input, the function will return a compile-time error.

```
QExpr stateToQExpr(qbit& q, const char c) {
  return
      qexpr::cIf(c == '0', qexpr::_PrepZ(q),
      qexpr::cIf(c == '1', qexpr::_PrepZ(q) + qexpr::_X(q),
      qexpr::cIf(c == '+', qexpr::_PrepZ(q) + qexpr::_H(q),
      qexpr::cIf(c == '-', qexpr::_PrepZ(q) + qexpr::_X(q) + qexpr::_H(q),
      qexpr::exitAtCompile("Expected a character in the set {0, 1, +, -}.")
      ))));
}
```

2.8 DataList

Recursion over `QList` values imposes a qubit-based view of quantum programming. Many domain-specific quantum algorithms require a higher level of abstraction. For this purpose, FLEQ introduces the `datalist::DataList` type for compile-time strings. Like a `QList`, a `DataList` is an immutable compile-time list that wraps statically-defined C strings or `char` arrays. A `DataList` can be constructed from a string literal via the constructor, or it can be constructed from a file using a pair of macros.

```
// in file "text.txt" ////////////////////////

STRINGIFYDATA(
This is from a file!
)

//////////////////////////////////////////////

const datalist::DataList str_src("This is from source!");

import_with_name_begin(file_str)
#include "test.txt"
import_with_name_end(file_str);
// equivalent to
//
// char file_str_raw[] = "this is from file!";
// const DataList file_str(file_str_raw);
```

2.8.1 Basic DataList operations

Like `QList` and ordinary C++ `std::string`, a `DataList` can be sliced, concatenated, sized, and addressed.

`unsigned int datalist::DataList::size()`

Return the length of the `DataList`. Compile-time resolvable.

`char datalist::DataList::operator[](unsigned long i)`

Index into a `DataList`, e.g. `data[i]`.

`datalist::DataList datalist::operator+(DataList data1, DataList data2)`

Concatenate two data lists, e.g. `data1 + data2`.

`datalist::DataList datalist::DataList::operator()(unsigned long start, unsigned long end)`

`datalist::DataList datalist::DataList::operator()(DataList start, unsigned long end)`

`datalist::DataList datalist::DataList::operator()(unsigned long start, DataList end)`

`datalist::DataList datalist::DataList::operator()(DataList start, DataList end)`

> Each of the four variants above returns a slice of a `DataList`, starting at the index `start` (inclusive) and ending at the index end (exclusive); e.g. `data(start, end)`. When `start` or end are `DataList` values, the index associated with that `DataList` is the result of `find(start)/find(end)` respectively.

`datalist::DataList datalist::operator>>(DataList data, unsigned long i)`

> Return the right shift of `data` by offset `i`, i.e. `data >> i == data(i, data.size())`.

`datalist::DataList datalist::operator<<(DataList data, unsigned long i)`

> Return the left shift of `data` by offset `i`, i.e. `data << i == data(0, data.size() - i)`.

`datalist::DataList datalist::operator+(DataList data, unsigned long i)`

> Return the right shift of `data` by offset `i`, i.e. `data + i == data(i, data.size())`.

`datalist::DataList datalist::operator++()`

> Return the right shift of a `DataList` by 1, i.e. `++data == data(1, data.size())`.

`unsigned long datalist::DataList::count(DataList sub_str1, DataList sub_str2, ...)`

> Return the number of occurrences of any of the substring arguments in the current `DataList`.
>
> In the case of variadic arguments like `count`, the behavior is `OR`-ed over all the arguments. So for example, `data.count("A", "B", "C")` returns the number of times the `DataList` contains `"A"` or `"B"` or `"C"`.

2.8.2 `DataList` conversions

FLEQ provides several functions for casting a `DataList` of a specific form to basic C types and visa versa:

`int datalist::DataList::to_int()` and `int datalist::_i(DataList data)`

> Convert a `DataList` into an integer; analogous to `std::stoi`.

`double datalist::DataList::to_double()` and `double datalist::_d(DataList data)`

> Convert a `DataList` into a double; analogous to `std::stod`.

`bool datalist::DataList::to_bool()` and `bool datalist::_b(DataList data)`

> Convert a `DataList` into an bool.

`datalist::DataList(int i)`

> Produce a `DatatList` from an integer, e.g. `DataList x (5);` produces a `DataList` variable x with value `"5"`.

If a cast fails (i.e. if the `to_int()` is called on a `DataList` that does not consist of digits), the behavior is undefined, which could lead to incorrect results. Note that the compiler does not exit in this case, though this behavior may be changed in future versions. Best practice is to confirm the `DataList` has the appropriate form before casting. For example, the following `QExpr` function expects a `DataList` of the form `0` or `1`, and casts the result to a boolean used in the conditional of a `cIf`.

```
QExpr unguarded_cast(qbit &q, datalist::DataList cond) {
  return qexpr::cIf(cond.to_bool(), qexpr::_X(q), qexpr::identity());
}
```

If this function is evaluated with an ill-formed input, such as `unguarded_cast(q, "X")`, the program will still compile, but will fail at runtime without any kind of error handling.

A user can prevent this failure case by using an additional `cIf` to ensure the `DataList` has the appropriate form. For example:

```
QExpr guarded_cast(qbit &q, datalist::DataList cond) {
  return qexpr::cIf(cond == datalist::DataList("0")
                || cond == datalist::DataList("1"),
            qexpr::cIf(cond.to_bool(), qexpr::_X(q), qexpr::identity()),
            qexpr::exitAtCompile("Expected 0 or 1.")
        );
}
```

If this function is evaluated with an ill-formed input, the compiler will exit with the error message "Expected 0 or 1".

2.8.3 Parsing

To aid with parsing, `DataList` includes several substring search functions returning an index into the current `DataList`.

`unsigned long datalist::DataList::find(DataList sub_str)`

> Returns the index of the start of the first occurrence of `sub_str` in the `DataList`.

`unsigned long datalist::DataList::find_last(DataList sub_str)`

> Returns the index of the start of the last occurrence of `sub_str` in the `DataList`.

`unsigned long datalist::DataList::find_any(DataList chars)`

> Returns the index of the first occurrence of any of the characters in `chars`. For example, `DataList("xyz20").find_any("012")` will return the index 3 as `"xyz20"[3] = "2"`.

`unsigned long datalist::DataList::find_any_last(DataList chars)`

> Returns the index of the last occurrence of any of the characters in `chars`.

`unsigned long datalist::DataList::find_not(DataList sub_str)`

> Returns the index of the first character not matching any character of `sub_str`. Will return `0` if `sub_str` is not a prefix of the current `DataList`.

`unsigned long datalist::DataList::find_not_last(DataList sub_str)`

Returns the index of the last character not matching any character of `sub_str`. Will return `size()` if `sub_str` is not a suffix of the current `DataList`.

In addition, `DataList` contains several utilities that return substrings.

`datalist::DataList datalist::DataList::next(DataList sub_str1, DataList sub_str2, ...)`

Returns the `DataList` slice beginning at the first occurrence of any of the arguments. For example:

```
DataList("find this 123 or this 345").next("this") = "this 123 or this
345"
```

```
DataList("The quick brown fox.").next("fox", "quick") = "quick brown
fox."
```

```
DataList("The quick brown fox.").next("fox", "house") = "fox."
```

`datalist::DataList datalist::DataList::after_next (DataList sub_str1, DataList sub_str2, ...)`

Returns the `DataList` slice beginning directly after the first occurrence of any of the arguments. For example:

```
DataList("find this 123 or this 345").after_next("this") = " 123 or
this 345"
```

```
DataList("The quick brown fox.").after_next("fox", "quick") = " brown
fox."
```

```
DataList("The quick brown fox.").after_next("fox", "house") = "."
```

`datalist::DataList datalist::DataList::next_not(DataList sub_str)`

Returns the `DataList` occurring after the index `find_not(sub_str)`.

`datalist::DataList datalist::DataList::next_block(DataList chars)`

Returns the first `DataList` slice whose elements all match any of the characters in `chars`.

For example, `d.next_block("0123456789")` will return the first integer in `d`.

```
DataList("July 18, 1968").next_block("0123456789") = "18"
```

`datalist::DataList datalist::DataList::last(DataList sub_str1, DataList sub_str2, ...)`

Returns the `DataList` slice beginning at the last occurrence of any of the substring arguments.

`datalist::DataList datalist::DataList::after_last(DataList sub_str1, DataList sub_str2, ...)`

Returns the `DataList` slice beginning immediately after the last occurrence of any of the substring arguments.

`datalist::DataList datalist::DataList::last_not(DataList sub_str)`

Returns the slice of the current `DataList` starting at the index `find_not_last(sub_str)`.

`datalist::DataList datalist::DataList::last_block(DataList sub_str)`

Returns the last `DataList` block whose elements all match any of the characters in `chars`. For example:

```
DataList("July 18, 1968").last_block("0123456789") = "1968"
```

2.8.4 Runtime and compile-time conversions

If a compile-time `DataList` is needed at runtime, it can be converted into either a runtime equivalent `std::string`, or to a `char` array.

`std::string datalist::to_string(DataList data)`

Return the runtime C++ string represented by the `DataList` `data`.

`char * datalist::to_char_array(DataList data).`

Return the runtime C string represented by the `DataList` `data`.

Some DSL examples (see **Domain-specific languages using FLEQ**) may want to create an array or `QList` whose size is specified by a `DataList` input. The `datalist` namespace thus also includes a template function that can take as input a constant integer and allocates an array of that size of any type as specified by the template argument, including `qbit`.

`template<typename Type> Type * datalist::IQC_alloca(DataList name = "", const unsigned long N = 1)`

At compile-time, generate an array of `Type` objects of size `N` with IR name `name` (for printing and debugging purposes) and return a pointer to the first element. Note, no additional memory management is required. The scope of the variable will be the same as if a standard array allocation.

For example, suppose a user has a quantum kernel expression that takes as input a `QList` of arbitrary size and returns (by reference) a boolean result. The user want to write a function that takes a `DataList` argument that specifies the number of qubits to use in the `QList`. They could use `IQC_alloca` to allocate the appropriate `QList` at compile-time as follows:

```
QExpr op(qlist::QList qs, bool& result) {...}

QExpr opOnNQubits(datalist::DataList N, bool &result) {

  qbit *qs_raw = datalist::IQC_alloca<qbit>("qs", N.to_int());
  qlist::QList qs (qs_raw);

  return op(qs, result);
}
```

This function can be invoked on 3 qubits via a call to `qexpr::eval_hold(opOnNQubits("3", result))`.

2.9 Barriers and binding

2.9.1 Quantum basic blocks and barriers

Quantum programming languages often provide **circuit barriers** which prevent optimization across a boundary. That is, a barrier guarantees that all gates that come before it are are executed before any of the gates that come after. In the Intel® Quantum SDK, every two top-level `quantum_kernel` function calls or FLEQ evaluation calls are separated implicitly by a barrier; in other words, barriers separate any two QBBs. Each QBB is a standalone block of quantum logic, and does not know a priori what quantum logic is executed before or after it. As a result, each QBB must be implicitly bookended by circuit barriers.

Consider an example of two separate evaluation calls to a unitary not gate.

```
qexpr::eval_hold(qexpr::_X(q)); // 1 QBB with 1 gate
qexpr::eval_hold(qexpr::_X(q)); // 1 QBB with 1 gate
```

These two evaluation calls produce two separate QBBs, each containing one gate. If, however, these quantum kernel expressions were joined together and executed in a single evaluation call, and thus a single QBB, the QBB would be optimized by the compiler, canceling out both gates and producing an empty QBB with zero gates.

```
qexpr::eval_hold(qexpr::_X(q) + qexpr::_X(q)); // 1 optimized QBB with 0 gates
```

Barriers in the SDK have the additional property that measurement outcomes executed before the barrier are not returned from the backend to the classical runtime (i.e. accessible by the program) until after the barrier. In this sense, separate `quantum_kernel` function calls or FLEQ evaluation calls also act as a "measurement return barrier". For example, a measurement in a quantum kernel expression joined with a conditional `cIf` statement (see Branching) will **not** correctly propagate the measurement result to the conditional because they occur in the same QBB. Consider the following example:

```
bool b = true;
// WRONG: Always executes H(q2) because the measurement result of q1
// is not written to b until the end of the current QBB.
qexpr::eval_hold(qexpr::_MeasZ(q1, b)
                 +
                 qexpr::cIf(b, qexpr::_H(q2), qexpr::identity()));
```

On the other hand, if the measurement and conditional calls occur in separate evaluation calls, there is an implicit barrier between them, which will produce correct results.

```
bool b = true;
qexpr::eval_hold(qexpr::_MeasZ(q1, b));
// Measurement results are written to b a the end of the above QBB,
// and so are available by the start of the next QBB.
qexpr::eval_hold(qexpr::cIf(b, qexpr::_H(q2), qexpr::identity()));
```

2.9.2 Bind

While separate `quantum_kernel` functions and evaluation calls act as a barrier, it can also be useful for programmers to insert their own barriers within a single quantum kernel expression, for the purposes of expressivity, debugging, or branching control. This is provided by the `qexpr::bind` functionality, which is also referred to as a "barriered join". A `bind` combines two quantum kernel expressions much in the same way as an ordinary `join`, but acts as a barrier between them. Under the hood, the `bind` function produces two separate QBBs, one for each quantum kernel expression, which implicitly imposes the barrier.

There are four variations on the `bind` method.

`QExpr qexpr::bind(QExpr e1, QExpr e2)`

> Returns the barriered join of `e1` with `e2` in **sequential order**, meaning that it evaluates `e1` followed by `e2`.

`e1 << e2`

> Shorthand for `qexpr::bind(e1, e2)`. Analogous to `e1 + e2`.

`e1 >> e2`

> Binds the quantum kernel expressions in **composition order**, meaning that it evaluates `e2` followed by `e1`. Analogous to `e1 * e2`.

`QExpr qexpr::fence(QExpr e)`

> Shorthand for `qexpr::bind(qexpr::identity(), e)`.

To remember the difference between left shift `e1 << e2` and right shift `e1 >> e2` operators for quantum kernel expressions, recall that sequential composition aligns with the left shift operator for stream output e.g. `std::cout << "Hello " << "World!"` where "Hello " and "World!" are composed in sequential order.

A `bind` call is logically equivalent to a `join` call in nearly every case except when acting as a measurement barrier. However, whereas the exclusive use of `join` ensures that at runtime, exactly one QBB is issued per evaluation call, even with unresolved branching (see **Branching**), each call to `bind` increases the number of QBBs issued at runtime per evaluation call by one (for that branch) with the order of the issuing as specified by the ordering or directionality of the `bind`. Because each QBB is bookened by circuit barriers and the end barrier is also a measurement return barrier, `bind` is the direct analog of the barrier for `QExpr`. There are three common reasons to use `bind` over `join`:

1. **Debugging.** As discussed in **Printing**, FLEQ compilation uses the PCOAST graph as a quantum IR. Due to its level of abstraction, it can be difficult to determine the gate sequence used to form the PCOAST graph representation, obscuring errors in the gate logic. `bind` can be used to prevent some of the implicit optimizations to better debug gate logic errors. When convinced of the logical correctness, a programmer can change the `bind`'s to `join`'s to regain the optimization and efficiency.
2. **Branching on measurement outcomes.** As discussed in **Quantum basic blocks and barriers**, barriers can be necessary to separate measurements from conditionals that depend on those measurements, like a `cIf`. For example, consider a qubit reset function (logically equivalent to `_PrepZ`) which measures the qubit and conditioned on the outcome, applies an `_X` gate to the `true` case:

```
PROTECT QExpr resetQubit(qbit &q) {
  cbit c = false;
  return qexpr::_MeasZ(q, c) << qexpr::cIf(c, qexpr::_X(q), qexpr::identity());
}

PROTECT QExpr NOT_A_RESET(qbit &q) {
  cbit c = false;
  return qexpr::_MeasZ(q, c) + qexpr::cIf(c, qexpr::_X(q), qexpr::identity());
}
```

The first function uses `bind` to fuse the measurement to the conditional, thus ensuring the measurement is performed and stored to `c` before evaluating the `cIf`. The second case does not use `bind` and is logically equivalent to only the measurement since `c` remains its initial value of `false` when the `cIf` is evaluated.

3. **Unresolved branching control.** As discussed in Branching, FLEQ compilation handles unresolved branching (in the absence of `bind`) by combinatorially building all possible QBBs and inserting classical branching to select exactly one at runtime. This means that every `join` between two unresolved `cIf` calls doubles the number of QBBs. This can cause an exponential increase in the number and complexity of that branching, although FLEQ compilation performs admirably in generating these branches (on-the-order of thousands without a prohibitive increase in compile-time and binary size). However, this exponential increase eventually will become problematic if not controlled. One way to avoid this is by replacing `join`'s with `bind`'s. By issuing multiple QBBs per evaluation, FLEQ compilation no longer needs to generate every unique combination, but only those bookended by the `bind`.

Each core FLEQ function has a different distributive or associative behavior with regards to `bind`:

- `qexp::control`, `qexpr::power`, `qexpr::printQuantumLogic`, and `qexpr::printDataList` all distribute over `bind`, i.e. `f(e1 << e2)` is equivalent to `f(e1) << f(e2)` for `f` being one of these four functions.

  All of these are intuitive except for `power`. Consider that `power(2, e1 << e2)` is equivalent to `power(2, e1) << power(2, e2)` – that is, `e1 + e1 << e2 + e2` – and **not** `e1 << e2 + e1 << e2`. This is a known issue which can be overcome by using a recursive version of power:

  ```
  QExpr recursivePower(const int n, QExpr e) {
    return qexpr::cIf(n == 0, qexpr::identity(),
           qexpr::cIf(n > 0,  e + recursivePower(n-1, e),
           qexpr::invert(e) + recursivePower(n+1, e)
    ));
  }
  ```

- `qexpr::invert` distributes in the analogous way to `qexpr::join`, i.e. `qexpr::invert(e1 << e2)` is equivalent to `qexpr::invert(e2) << qexpr::invert(e1)`. Note, this inversion of ordering also holds for classical instructions attached to the quantum kernel expressions `e1` and `e2`; see Ordering of classical and quantum operations.

- `qexpr::join` is associative with `bind`, not distributive. For example:

  1. `e1 + (e2 << e3)` is equivalent to `(e1 + e2) << e3`
  2. `(e1 << e2) + e3` is equivalent to `e1 << (e2 + e3)`
  3. `e1 * (e2 << e3)` is equivalent to `e2 << (e1 * e3)`

- `qexpr::cIf` does not distribute over or associate with `bind`. If the conditional is resolved, then just as described, the resolved branch `QExpr`, `bind`-ed or not, is returned. If the conditional is not resolved, then the `bind` behavior is dependent on the branch as expected. For example, `cIf(b, e1, e2 << e3)` generates a `true` branch which only represents the QBB(s) for `e1` whereas the `false` branch represents the consecutive QBB(s) for `e2` followed by those for `e3`. `bind` effectively distributes over `cIf`, i.e. `e1 << cIf(b, e2, e3)` is logically equivalent to `cIf(b, e1 << e2, e1 << e3)` but the branching in the IR will be different as the former represents the QBB(s) of `e1` which then branch between `e2` and `e3` whereas the latter branches to one of two copies of `e1` which then branches to either `e2` or `e3` based on the condition.

Custom functions inherent their behavior with respects to `bind` from their constituent core function calls.

2.10 Advanced topics

2.10.1 Ordering of classical and quantum operations

Like a `quantum_kernel` function, a quantum kernel expression is not just the quantum logic as encapsulated in QBBs. It also includes the interface between the QBBs and and the classical processes that surround them. Examples of classical processes include generating dynamic angles to be passed to gates and retrieving and analyzing measurement results. As such, the FLEQ compilation stage of IQC must aggregate and rearrange such classical operators as encapsulated in the constituent `QExpr`-returning functions. FLEQ compilation recognizes three sections for each such function: the pre-quantum section, the quantum section and the post-quantum section. Note, the post-quantum section is trivial except for cases where the function is converted from a `quantum_kernel` function or the returned value is dependent on a call to `this_as_expr`. In both cases, the post-quantum section contains all classical logic that follows the last imperative gate call.

In the absence of any `bind` calls, the pre-quantum sections are aggregated so as to be executed on the classical processor before the QBB is issued to the quantum processor in a topologically sorted order; see **Overview of FLEQ compilation**. Likewise, the post-sections are aggregated so as to be executed on the classical processor in the analogous topologically sorted order. In the case of unresolved branching, classical logic associated with a given branch will only be executed in that branch. For example,

```
qbit q;

PROTECT QExpr true_branch() {
  double ang = compute_angle_true();
  return _RX(q, ang);
}

PROTECT QExpr false_branch() {
  double ang = compute_angle_false();
  return _RX(q, ang);
}

QExpr foo(bool b) {
  return cIf(b, true_branch(), false_branch());
}
```

In this example, if `foo` is evaluated such that its argument can not be resolved at compile-time, the function `compute_angle_true` is only executed at runtime on the classical processor if the argument is evaluated to be `true` at runtime, and likewise for `c_foo_false` if the argument evaluates to `false` at runtime. Note the use of the `PROTECT` attribute for `true_branch` and `false_branch`. This is added because the base LLVM processing in IQC may prematurely inline these function into `foo`, preventing the FLEQ compilation stage from discriminating which classical logic belongs in which branch. This kind of attention to ordering and control over execution on the classical processor is primarily relevant if such function calls have side- effects outside of the scope of FLEQ. For example, calls to member functions of a class may manipulate class member data in which case, special care must be given to insure the execution order is as desired.

A `QExpr`-returning function can introduce locally scoped variables, including locally-scoped qubits; see **Local qubits**. However, such functions should **always** use the `PROTECT` attribute as once again, premature inlining can cause improper scoping of these variables and possibly result in a runtime segmentation fault. This is not currently caught by the FLEQ compilation stage of IQC; see **Known limitations**.

When `bind` functionality is introduced, the pre-quantum and post-quantum sections are now shifted relative to the `bind`-generated QBB as discussed in section **Quantum basic blocks and barriers**. For example,

```
PROTECT QExpr measureBit(qbit &anc, cbit &cont) {
  return _MeasZ(anc, cont);
}

PROTECT QExpr rotateIfTrue(qbit &q, cbit &cont) {
  double ang = 3.14 / 4. * (double)cont;
  return _RX(q, ang);
}

int main() {

  qbit anc;
  qbit q;
  cbit cont = false;
   eval_hold(_H(anc) + measureBit(anc, cont)
                     << rotateIfTrue(q, cont));
}
```

In this case, the instructions which calculate the rotation angle in `rotateIfTrue` based on the passed `cbit` value is inserted in the pre-quantum section for the QBB associated with the right-side of the `bind`, and thus is executed in between the two QBBs, as expected. Note again the use of the `PROTECT` attribution to prevent premature inlining. Also note as specified in section **Basic concepts** that the `cbit` is passed to `rotateIfTrue` by reference.

As discussed earlier, non-trivial post-quantum sections are only generated through the use of `convert` or `this_as_expr` functionality. Moreover, it is a known issue that in certain cases where unresolved branching and `bind` functionality are used together, the post-quantum sections may not be inserted into the correct branches or the correct order; see **Known limitations**. To enforce an ordering on classical logic, one can always encapsulate the logic in an "empty" `QExpr`-returning function, i.e. one that returns only `qexpr::identity` or `this_as_expr` without any imperative gate calls. Ordering is then enforced via `bind` functionality with these empty `QExpr`-returning functions.

As an example, suppose one writes a quantum function `QExpr getData(QList qdata, cbit cdata[])` to extract quantum measurement data and classical data analysis function `bool analyzeData(cbit cdata[])`. One can then wrap the analysis function as an empty `QExpr` and `bind` the two together to form a complete calculation as a `QExpr`:

```
QExpr getData(QList qdata, cbit cdata[]);
bool analyzeData(cbit cdata[]);

PROTECT QExpr analyzeData(cbit &result, cbit cdata[]) {
  result = analyzeData(cdata);
  return this_as_expr;
}

PROTECT QExpr calculateData(cbit &result, QList qdata) {
  //use IQC_alloca for the cbit array so that
  // we can use the size of the QList to determine size
  cbit *cdata = IQC_alloca<cbit>("", data.size());
  return getData(qdata, cdata) << analyzeData(result, cdata);
}
```

2.10.2 Higher-order functions

Recursive `QExpr` functions are extremely effective at producing reusable quantum kernel expressions that can map over arbitrary compile-time `QList` or `DataList` values. However, in many cases these recursive functions can result in significant boilerplate code, and patterns emerge common to functional programming.

For example, one of the most common idioms in recursive `QExpr` functions is mapping a particular `QExpr` function over a `QList`, applying it to each qubit in a sequential join. Consider the following recursive `QExpr` function that applies a `PrepZ_` gate to every qubit in a `QList`:

```
QExpr prepAll_recursive(qlist::QList qs) {
  return qexpr::cIf(qs.size() > 0,
                    qexpr::_PrepZ(qs[0]) + prepAll_recursive(qs + 1), // qs non-empty
                    qexpr::identity()                                 // qs empty
  );
}
```

Using C++ function pointers, it is possible to write higher-order functions—that is, functions that take as input other function pointers—to reduce this boilerplate overhead.

For convenience, the library `qexpr_utils.h` provides several examples of higher-order functions.

For instance, the following function, `map1`, takes as input a function pointer and applies it to each qubit in a `QList`. The recursive structure of `map1` is identical to that of `prepAll_recursive`, just with an extra function pointer parameter.

```
// qexpr_utils.h
template<typename QExprFun>
QExpr qexpr::map1(QExprFun f, qlist::QList qs) {
  return qexpr::cIf(qs.size() > 0,
          f(qs[0]) + qexpr::map1(f, qs + 1),  // qs non-empty
          qexpr::identity()                   // qs empty
  );
}
```

This templated function applies a function `f`, which takes a `qbit` and returns a `QExpr`, to every `qbit` in a `QList`, and returns the join of all the results. Then, instead of a user writing the function `prepAll_recursive()`, they can simply invoke `map1(qexpr::_PrepZ, qs)` to prepare all the qubits in `qs`.

Not all recursive `QExpr` functions fit exactly into this pattern. For example, suppose a user wants to apply rotation gates to each qubit in a `QList`, with different rotation arguments for each qubit. The relevant recursive function would need to map over both the `QList` argument and an array of rotation parameters, for example:

```
// Assume that params is an array of doubles of size qs.size()
QExpr RZAll_recursive(qlist::QList qs, double* params) {
  return qexpr::cIf(qs.size() > 0,
                    qexpr::_RZ(qs[0], params[0]) + RZAll_recursive(qs+1, params+1),
                    qexpr::identity()
  );
}
```

Because `_RZ` does not take a single qubit argument, it is not possible to apply `map1` directly. However, the `qexpr_utils.h` library also supplies a more general `map` function, which takes as input a function `f` that takes in any number of arguments, the first of which is a `qbit`. The `map` function then expects (1) a `QList` to map over; (2) some number of `QList` or array arguments (it applies `f` to every element in the array); and (3) some number of scalar arguments, which it passes directly to `f`.

```
// qexpr_utils.h
template<typename QExprFun, typename... Args>
QExpr qexpr::map(QExprFun f, qlist::QList qs, Args... args) noexcept;
```

Instead of `RZAll_recursive(qs, params)`, a user can just apply `qexpr::map(qexpr::_RZ, qs, params)` to obtain the same result.

This `map` function can be used in several ways:

1. Like `map1`, can be used to map a single-qubit gate over a `QList`:

   ```
   qexpr::map(qexpr::_PrepZ,qs)
   ```

2. Map a multi-qubit gate, like `CNOT`, over two `QLists`:

   ```
   qexpr::map(qexpr::_CNOT, qs1, qs2)
   ```

3. Map a single-qubit gate that accepts parameters, e.g. `RZ`, over a `QList`, with the same scalar parameter applied to each argument:

   ```
   qexpr::map(qexpr::_RZ,qs,M_PI/2)
   ```

4. Map a single-qubit gate with parameters, like `RZ`, over (1) a `QList` and (2) an array of rotation parameters:

   ```
   double params[3] = {M_PI/2, M_PI/4, M_PI/8};
   qexpr::eval_hold(qexpr::map(qexpr::_RZ, qs, params));
   ```

2.10.3 Local qubits

Like `quantum_kernel` functions, `QExpr` functions can use both local and global qubits. However, while top-level `quantum_kernel` functions cannot accept qubit arguments, there is no such restriction for quantum kernel expressions. Qubits or `QList` values can now be declared locally in a non-quantum function and passed to top-level `QExpr` functions. For example:

```
int main() {
  qbit q;
  bool b;
  eval_hold(_PrepZ(q) + _H(q) + _MeasZ(q, b));
}
```

The underlying reason for this is that every evaluation call to `eval_hold` or `eval_release` from a (classical) function `foo()` tells the compiler to treat `foo()` as a `quantum_kernel` function. Because local qubits can be declared inside `quantum_kernel` functions, they can therefore be declared in classical functions with evaluation calls.

If local qubits are declared within a `QExpr`-returning function, it is best practice to use the `PROTECT` attribute; see Known limitations.

2.10.4 Domain-specific languages using FLEQ

This section will illustrate some techniques for using quantum kernel expressions and compile-time lists to implement domain-specific representations of programs, collectively known as domain specific languages (DSLs).

For example, suppose a user wants to take as input a string indicating an n-qubit basis state such as $|01+-\rangle$, and prepare a `QList` of length n in that state. The format of the input string can be thought of as a simple DSL for specifying state preparations.

To implement such a DSL, a user must write a `QExpr` function that takes as input the `DataList` and a `QList` and returns a quantum kernel expression. To start, the function `stateToQExpr` below returns the quantum kernel expression corresponding to a single character, while `multiStateToQExpr` recursively applies `stateToQExpr` to each character in a `DataList`.

```
QExpr stateToQExpr(qbit& q, const char c) {
  return
      qexpr::cIf(c == '0', qexpr::_PrepZ(q),
      qexpr::cIf(c == '1', qexpr::_PrepZ(q) + qexpr::_X(q),
      qexpr::cIf(c == '+', qexpr::_PrepX(q),
      qexpr::cIf(c == '-', qexpr::_PrepX(q) + qexpr::_Z(q),
      qexpr::cIf(c == 'R', qexpr::_PrepY(q),
      qexpr::cIf(c == 'L', qexpr::_PrepY(q) + qexpr::_Z(q),
      qexpr::exitAtCompile("prepState: Expected a character in the set {0,1,+,-,R,L}.")
      ))))));
}

QExpr multiStateToQExpr(const qlist::QList qs, const datalist::DataList src) {
    return qexpr::cIf(qs.size() == 0,
                        qexpr::identity(),
                        stateToQExpr(qs[0], src[0])
```

(continues on next page)

(continued from previous page)

```
                                + multiStateToQExpr(qs>>1, src>>1)
    );
}
```

Finally, the function `prepState` checks the input `DataList` and ensures it has the correct format, before stripping the beginning and ending characters that make up the ket syntax.

```
QExpr prepState(const datalist::DataList src, const qlist::QList qs) {
  return
      qexpr::qassert(src[0] == '|',
              "prepState: Expected a datalist of the form |state>")
      +
      qexpr::qassert(src[src.size()-1] == '>',
              "prepState: Expected a datalist of the form |state>")
      +
      qexpr::qassert(src.size() == qs.size() + 2,
              datalist::DataList("prepState: Expected a state of size ")
              + datalist::DataList(qs.size()))
      +
      // Strip the ket from the datalist
      multiStateToQExpr(qs, src("|",">") >> 1)
      ;
}
```

The function `qassert` is defined in `qexpr_utils.h` and will raise a compile-time error if the boolean condition is false.

The full example is shown in the sample file `state_preparation.cpp` (see Developer Guide and Reference (Samples)).

While the `prepState` function is rather straightforward, these features can be generalized to deal with more advanced DSL features, including persistent state and symbol tables. In these cases, a `DataList` representing the current state would be passed to each `QExpr` function and updated, similar to the concept of a state monad in functional programming.

For example, suppose a user needs to process some input `DataList` into a different form before it can be used to construct a `QExpr` via a function `stateToQExpr`. Then the user may write the following function, `processInput`, which iteratively converts an input `DataList` into a `DataList` of the correct form, before feeding the well-formed state to `stateToQExpr`.

```
QExpr stateToQExpr(datalist::DataList state);
datalist::DataList updateState (datalist::DataList state, const char c);

QExpr processInput(datalist::DataList state, datalist::DataList input) {
  return qexpr::cIf(input.size() == 0,
                      stateToQExpr(state),
                      processInput(updateState(state, input[0]),
                                   input>>1)
  );
}
```

2.10.5 Debugging

FLEQ enables higher level of abstraction for quantum programming through quantum kernel expressions, conditionals, recursion, `DataList` DSLs, and more. With this higher level of abstraction however, it becomes more difficult to debug quantum programs that are behaving in unexpected ways. FLEQ supplies several utilities, most of which have already been introduced, to aid in debugging. This section provides an overview of how these utilities can be utilized to debug quantum programs.

1. The functions `qexpr::exitAtCompile` and `qexpr::exitAtRuntime` (see **Exiting**) should be used liberally to catch improper inputs to quantum kernel expression functions.

2. If a bug arises from a `DataList` argument, or if a user wants to determine how far in evaluation of a `QExpr` the compiler was able to reach, they can use `qexpr::printDataList()` (see **Printing**) to act like a print statement during FLEQ evaluation as part of compilation. Users should be aware that such messages are buffered in a print buffer during compile-time, and the order of those messages is based on a topological sort of the FLEQ graph (see **Overview of FLEQ compilation**), which may not always align with their intuition.

3. If a large quantum kernel expression contains a bug, the function `qexpr::printQuantumLogic()` (see **Printing**) can be used to narrow down the problem to a sub-expression. In particular, `qexpr::printQuantumLogic()` will display the PCOAST graph generated by a particular quantum kernel expression, even if it occurs within a larger `QExpr`. The PCOAST representation, described in detail in [PCOAST2023], is a compact representation of quantum logic, so if a user knows what PCOAST graph should be represented by a certain subcomponent of their `QExpr`, they can compare the actual output against the expected output.

4. `qexpr::printQuantumLogic()` can also be used in conjunction with `qexpr::convert<>` to display the PCOAST graph associated with a quantum kernel function, even if FLEQ is not being used more broadly.

5. Barriers in a quantum kernel expression prevent optimization across boundaries. Users may want to replace calls to `join` in a `QExpr` with calls to `bind` (see **Barriers and binding**) to isolate a problem. After the problem is fixed, they may be able to move back towards using `join`, which results in a more highly optimized circuit implementation.

6. The compiler flag `-P` can be used for debugging quantum kernel expressions in a similar way as for ordinary `quantum_kernel` functions. The `-P` flag prints all fully evaluated quantum kernel expressions (as well as all `quantum_kernel` functions) to either the console, `.tex` files compatible with LaTeX, or `.json` files. For quantum kernel expressions, however, only the fully optimized and synthesized versions of the circuits will be displayed.

7. The compiler flag `-v` prints out statistics for each evaluated quantum kernel expression and `quantum_kernel` function.

8. Users can examine the intermediate LLVM files produced by the compiler by specifying the compiler flag `-k`. This flag will generate a number of intermediate files with the `.ll` extension. For the purposes of FLEQ debugging, the user should examine `<filename>_lowered.ll`, which is the earliest LLVM file generated by `-k` with results of the FLEQ compilation stage. (**Overview of FLEQ compilation**).

9. Ill-formed uses of `DataList` and `QList` **do not** cause the compilation to exit, but do generate warnings which can be seen through the use of command-line flags. See **Overview of FLEQ compilation** for details.

10. As discussed in **Ordering of classical and quantum operations**, if classical code is interwoven with quantum kernel expressions, it is best practice to add the `PROTECT` attribute to avoid premature inlining. This is especially true if the function introduces local variables.

11. In the v1.1 release of the Intel® Quantum SDK, FLEQ is still in beta. Please feel free to voice any comments, concerns, questions or suggestions to the URL provided in **Support**. This feedback is vital to shape FLEQ and maximize its utility for quantum applications developers and researchers.

2.10.6 Overview of FLEQ compilation

Though not strictly required, it can be helpful to understand how the FLEQ compilation stage of the Intel® Quantum Compiler (IQC) functions in order to better understand how to best leverage the tools. By design, FLEQ compilation is one stage in the transformation of the LLVM intermediate representation (IR) (generated from the source code) into a hybrid quantum-classical binary executable.

FLEQ compilation happens just after many of the classical code IR optimizations (standard LLVM loop unrolling, inlining, and optimization), but just before most quantum code IR transformations (quantum optimization, native gate decomposition, qubit placement, routing and scheduling). As a result, FLEQ compilation must meet the IR constraints of fully formed QBBs (in IR).

In particular, FLEQ compilation produces quantum logic in the form of a PCOAST graph for each generated QBB, which is passed to a circuit synthesis stage used by the `O1` optimization flag. This is why the optimization flags `O0` versus `O1` have no appreciable effect on the gate sequences generated by FLEQ; they pass through this synthesis step no matter the flag. The PCOAST graph was chosen as it is an efficient, easily manipulated quantum IR, but future versions of FLEQ may use different IRs.

Note that the use of `O0` versus `O1` does still affect the way quantum kernel functions are processed later on in the compiler, regardless of the use of FLEQ.

Before FLEQ compilation, all the user-facing FLEQ functions are replaced with IQC builtin LLVM intrinsic function equivalents (in other words, they are opaque to core LLVM). The FLEQ builtins are then processed by FLEQ compilation and ultimately removed so that none of these builtins are present in the IR after the FLEQ compilation stage.

To begin FLEQ compilation, the IQC identifies each evaluation call (`eval_hold` or `eval_release`) and from it builds a **FLEQ Evaluation Graph** or FLEQ graph. Each `QExpr` function in the dependency of the evaluation call is represented as a node in the FLEQ graph with edges to `QExpr` arguments to that function.

From the FLEQ graph, the quantum logic is then built in the place of the call through three primary steps:

1. **Validation and Conditioning.** Each user-defined function in the FLEQ graph is checked to verify that it satisfies the conditions outlined in **Basic concepts**. The function is then "conditioned", i.e. manipulated so as to be amenable to the rest of the process. As discussed in **Ordering of classical and quantum operations**, this identifies three sections, similar to the core IQC: the pre-quantum section, the quantum section and the post-quantum section. For functions which do not use imperative gate calls (i.e. not generated by `convert` or use of `this_as_expr`), the post-quantum section is trivial.

2. **Inlining and Unrolling.** All user-defined functions are inlined in the place of the evaluation call. The structure of the FLEQ graph indicates recursion through loops in the graph. So, inlining is performed in two steps.

   1. A shallow inlining is applied bottom-up on functions that do not recurse, and do not have the `PROTECT` attribute.

   2. The loop unrolling step is performed top-down. In this step, recursive and `PROTECT`-ed functions are inlined and branching nodes are resolved if possible. A counter is added for every recursing function to keep track of the number of times they have been inlined to prevent infinite looping. This recursion limit can be adjusted via command-line flags (see **Recursion**).

3. **Building of Branching and Quantum Logic.** The previous inlining step guarantees that the FLEQ graph has no remaining user-defined functions and contains no cycles i.e is a directed acyclic graph (DAG). As a result, it is possible to obtain a topological sort to the nodes of the FLEQ graph. The branching logic is built in LLVM IR top-down and to each (initially empty) QBB, the process assigns a list of leaf nodes that QBB is dependent on. Then, going bottom-up, the PCOAST graph for each node and each QBB is progressively built. This build order determines the order of the print buffer (see **Printing**). The final assignment of QBBs to PCOAST graphs for the evaluation call is then stored to be passed to the synthesis step later in IQC compilation.

The final step of the process is a clean-up phase where any remain calls to FLEQ functions are removed. This includes additional `DataList` and `QList` intrinsics which the process attempts to resolve and replace. The process of building evaluation calls often leaves behind unused and even ill-defined calls to these intrinsics, especially for recursing functions. This is because the inlining step must fully insert the recursing function even when it hit the recursion guard. Once the recursion guard condition is resolved, the FLEQ graph prunes off the unused exit, but the IR remains even if it is unused and ill-formed. A common example is indexing into an empty `QList` when the `QList`'s size is used as the exit condition. In its current form, the clean-up phase has no way to distinguish between these legal but ill-defined remnants of the evaluation build process, or illegal and ill-defined uses of the `QList` or `DataList` features generated by the user. For this reason, FLEQ compilation **does not** exit when such ill-formed cases are found but instead they are replaced by undefined behavior (as handled by core LLVM). Instead, warnings are thrown, but by default, these warnings are suppressed. These warning can be seen by setting the command-line flag:

```
-F verbose-cleanup=true
```

Also, the inlining process often generates trivial branching, where an LLVM Basic Block unconditionally branches to the next Basic Block and is the unique predecessor to that Basic Block. This is by virtue of the conditioning and aggregation of the pre- and post- quantum sections around newly generated QBBs. By default, the clean-up process collapses this trivial branching, but this can be stopped via the command-line flag:

```
-F bb-cleanup=false
```

For the advanced user, this generates a textual IR which is cluttered but can provide insight into the building of evaluation calls useful for debugging.

3.0 Support

3.1 Known limitations

3.1.1 Limitations on FLEQ types

- Quantum kernel expressions must be invoked via an evaluation call `eval_hold` or `eval_release`. If a function that returns a `QExpr` type is simply called from a top-level function, it will have no effect on the quantum backend (**Basics: evals, join, identity, and QExpr-returning functions**). For example, consider the following example:

```
QExpr myQExprFunction(qbit &q);

int main() {
  ...
  // WRONG: Returns an unused QExpr value; does NOT invoke the quantum runtime
  myQExprFunction(q);

  // CORRECT: Invokes the quantum runtime by evaluating a QExpr value
  qexpr::eval_hold(myQExprFunction(q));
}
```

- Evaluation calls are not supported inside of `quantum_kernel` functions that also call basic quantum gates.
- For quantum kernel expressions, there is no appreciable difference between the use of the `-O0` and `-O1` optimization flags. See **Overview of FLEQ compilation**.
- All `qbit` arguments must be passed by reference.
- Conventional data passed into and out of `QExpr`-returning functions within the same evaluation or function body should be passed by reference (see **Basics: evals, join, identity, and QExpr-returning functions**).
- Arguments of FLEQ types `QExpr`, `QList`, and `DataList` should be passed by value.
- The `PROTECT` atribute should be added to a `QExpr`-returning function, `f`, if:
  1. `f` uses `this_as_expr`.
  2. `f` introduces any local variables including local qubits within its body.
  3. the evaluation behavior of classical logic inside the body of `f` is order-dependent or branch dependent with respects to other such calls within the evaluation, especially with respect to `bind`-enforced ordering or when said classical logic generates side-effects outside the scope of FLEQ.
  4. any of the other cases above are in doubt or unclear.
- There are known cases where post-quantum sections are **not** moved/inserted into the approriate branch. This happens in special cases where both `bind` and unresolved branching are used together. When post-processing is desired within a `QExpr`, it is best practice to enforce desired order using `bind` and empty `QExpr` -returning functions (return `identity` or `this_as_expr` without gates) wrapping classical post-processing instructions and including the `PROTECT` attribute.

- QList declarations created using the listable macro e.g. listable(<name>, N) will be displayed during circuit printing and debugging as <name>_raw. See **QList**. The same is true for the DataList type and the import_with_name macros.
- DataList casts to int, double, and bool will result in undefined behavior if the DataList cannot be coerced to that type. See **DataList**.
- The command line -P flag for printing quantum kernels will only print quantum kernel expressions inside evaluation calls. In addition, it will only print the circuit after optimization and synthesis.
- Multi-qubit quantum control is limited to control by up to 8 qubits (see **Control**). This limit can be overcome by chaining control calls using recursion (see **Recursion**) or with the explicit use of ancilla.

3.1.2 Classical control flow

- Functions that return quantum kernel expressions must have a single return statement. The control flow of the function cannot depend on C++ conditionals (if statements) or loops. All classical conditionals inside a QExpr function should instead use cIf (see **Branching**).
- Similarly, let assignments cannot vary based on classical conditionals (see **Let/get**). Best practice is to not use let and get with C++ branching.
- Recursive QExpr functions must be able to be unrolled at compile-time. FLEQ does not currently support repeat-until-success loops in a single quantum kernel expression. See **Recursion**.
- cIf statements that cannot be resolved at compile-time can produce an exponential growth in the number of QBBs produced by that QExpr (see **Branching**). This is especially the case when using join over unresolved cIf branching. This exponential blowup can be avoided by using bind in the place of join (see **Barriers and binding**).
- Traditional C++ if statements and loops that contain only classical instructions are not encouraged. Best practice is to use cIf and recursion as described in **Branching** and **Recursion**.

3.1.3 Barriers

- Measurement results are not available within the same QExpr where measurements are invoked, unless they are separated by a barrier like bind or fence (see **Barriers and binding**).
- qexpr::power does not distribute over bind the way it does with join. Best practice is to never use bind with qexpr::power(). See **Barriers and binding**.

3.1.4 Print buffer

- The order in which printQuantumLogic and printDataList are displayed is based on a topological sort of the evaluation call dependencies, i.e. the order of traversal of the FLEQ evaluation graph (see **Overview of FLEQ compilation**).
- When exitAtCompile is found in the dependency of an evaluation call, the exit message is added to the print buffer and the build failure is only triggered once the build is complete, or a different failure point is found.
- exitAtCompile triggers a fail regardless of whether the call is within an unresolved branch or not. if one desires an exit be triggered only in one branch of many unresolved branches, consider using exitAtRuntime instead.

3.2 Bug reporting and feature requests

Users can get technical support, share ideas, and report bugs by visiting Intel Communities.

Bibliography

[PCOAST2023] J. Paykin, A. T. Schmitz, M. Ibrahim, X. -C. Wu and A. Y. Matsuura, “PCOAST: A Pauli-Based Quantum Circuit Optimization Framework.” Proceedings of QCE 2023. doi: 10.1109/QCE57702.2023.00087.