Contents

Q# Language

Q# is part of Microsoft's Quantum Development Kit and provides rich IDE support and tools for program visualization and analysis. Our goal is to support the development of future large-scale applications while supporting user's first efforts in that direction on current quantum hardware.

The type system permits Q# programs to safely interleave and naturally represent the composition of classical and quantum computations. A Q# program may express arbitrary classical computations based on quantum measurements that execute while qubits remain live, meaning they are not released and maintain their state. Even though the full complexity of such computations requires further hardware development, Q# programs can be targeted to run on various quantum hardware backends in Azure Quantum.

Q# is a stand-alone language offering a high level of abstraction. There is no notion of a quantum state or a circuit; instead, Q# implements programs in terms of statements and expressions, much like classical programming languages. Distinct quantum capabilities (such as support for functors and control-flow constructs) facilitate expressing, for example, phase estimation and quantum chemistry algorithms.

Program execution

The following program gives a first glimpse at how a Q# command-line application is implemented:

```
namespace Microsoft.Quantum.Samples {

    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Arrays as Array;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Diagnostics as Diagnostics;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Preparation;

    operation ApplyQFT (reg : LittleEndian) : Unit
    is Adj + Ctl {

        let qs = reg!;
        SwapReverseRegister(qs);

        for (i in Array.IndexRange(qs)) {
            for (j in 0 .. i-1) {
                Controlled R1Frac([qs[i]], (1, i - j, qs[j]));
            }
            H(qs[i]);
        }
    }

    @EntryPoint()
    operation RunProgram(vector : Double[]) : Unit {

        let n = Floor(Log(IntAsDouble(Length(vector))) / LogOf2());
        if (1 <<< n ≠ Length(vector)) {
            fail "Length(vector) needs to be a power of two.";
        }

        let amps = Array.Mapped(ComplexPolar(_,0.), vector);
        use qs = Qubit[n] {
            let reg = LittleEndian(qs);

            PrepareArbitraryState(amps, reg);
            Message("Before QFT:");
            Diagnostics.DumpRegister((), qs);

            ApplyQFT(reg);
            Message("After QFT:");
            Diagnostics.DumpRegister((), qs);

            ResetAll(qs);
        }
    }
}
```

The operation `PrepareArbitraryState` initializes a quantum state where the amplitudes for each basis state

correspond to the normalized entries of the specified vector. A quantum Fourier transformation (QFT) is then applied to that state.

The corresponding project file to build the application is the following:

```
<Project Sdk="Microsoft.Quantum.Sdk/0.12.20070124">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

The first line specifies the version number of the software development kit used to build the application, and line 4 indicates that the project is executable opposed to e.g. a library that cannot be invoked from the command line.

To run the application, you will need to install .NET Core. Then put both files in the same folder and run `dotnet build <projectFile>`, where `<projectFile>` is to be replaced with the path to the project file.

To run the program after having built it, run the command

```
    dotnet run --no-build --vector 1. 0. 0. 0.
```

The output from this invocation shows that the amplitudes of the quantum state after application of the QFT are evenly distributed and real. Note that the reason that we can so readily output the amplitudes of the state vector is that the previous program is, by default, run on a full state simulator, which supports outputting the tracked quantum state via `DumpRegister` for debugging purposes. The same would not be possible if we were to run it on quantum hardware instead, in which case the two calls to `DumpRegister` wouldn't do anything. You can see this by targeting the application to a particular hardware platform by adding the project property `<ExecutionTarget>honeywell.qpu</ExecutionTarget>` after `<PropertyGroup>`.

Namespaces

At its top-level, a Q# program consists of a set of namespaces. Aside from comments, namespaces are the only top-level elements in a Q# program, and any other elements must reside within a namespace. Each file may contain zero or more namespaces, and each namespace may span multiple files. Q# does not support nested namespaces.

A namespace block consists of the keyword `namespace`, followed by the namespace name, and the content of the block inside braces `{ }`. Namespace names consist of a sequence of one or more legal symbols separated by a dot (`.`). Double underscores (`__`), double dots (`..`), or an underscore followed by a dot (`_.`) are not permitted since these character sequences are reserved. More precisely, a fully qualified name may not contain such a sequence, and namespace names correspondingly cannot end with an underscore. While namespace names may contain dots for better readability, Q# does not support relative references to namespaces. For example, two namespaces `Foo` and `Foo.Bar` are unrelated, and there is no notion of a hierarchy. In particular, for a function `Baz` defined in `Foo.Bar`, it is *not* possible to open `Foo` and then access that function via `Bar.Baz`.

Within a namespace block, open directives precede any other namespace elements. Aside from `open` directives, namespace blocks may contain operation, function, and type declarations. These may occur in any order and are recursive by default, meaning they can be declared and used in any order and can call themselves; there is no need for the declaration of a type or callable to precede its use.

Open Directives

By default, everything declared within the same namespace can be accessed without further qualification. However, declarations in a different namespace can only be used by qualifying their name with the name

of the namespace they belong to or by opening that namespace before it is used, as shown in the following example.

```
namespace Microsoft.Quantum.Samples {

    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Arrays as Array;

    // ...
}
```

The example uses an `open` directive to import all types and callables declared in the Microsoft.Quantum.Artithmetic namespace. They can then be referred to by their unqualified name unless that name conflicts with a declaration in the namespace block or another opened namespace.

To avoid typing out the full name while still distinguishing where certain elements come from, you can define an alternative name, or *alias*, which is usually shorter, for a particular namespace. In this case, all types and callables declared in that namespace can be qualified by the defined short name instead. In the previous example, this is the case for the `Microsoft.Quantum.Arrays` namespace. A function `IndexRange` declared in `Microsoft.Quantum.Arrays`, for example, can then be used via `Array.IndexRange` within that namespace block.

Defining namespace aliases is particularly helpful when combined with the code completion functionality provided by the Q# extensions available for Visual Studio Code and Visual Studio. With the extension installed, typing the namespace alias followed by a dot will show a list of all the available elements in that namespace that are valid at the current location.

Whether you are opening a namespace or defining an alias, `open` directives need to precede any other namespace elements and are valid throughout the namespace piece in that file only.

Type Declarations

Q# supports user-defined types. User-defined types are similar to record types in F#; they are immutable but support a copy-and-update construct.

User-defined types

User-defined types may contain both named and anonymous items. The following declaration within a namespace, for example, defines a type `Complex` which has two named items `Real` and `Imaginary`, both of type `Double`:

```
    newtype Complex = (Real: Double, Imaginary : Double);
```

Any combination of named and unnamed items is supported, and inner items may also be named. For example, the type `Nested`, defined as

```
newtype Nested = (Double, (ItemName : Int, String));
```

contains two anonymous items of type `Double` and `String` respectively, and a named item `ItemName` of type `Int`.

You can access the contained items via their name or by *deconstruction* (for more information, see item access). You can also access a tuple of all items where the shape matches the one defined in the declaration via the unwrap operator.

User-defined types are useful for two reasons. First, as long as the libraries and programs that use the defined types access items via their name rather than by deconstruction, the type can be extended to contain additional items later on without breaking any library code. Because of this, accessing items via deconstruction is generally discouraged.

Second, Q# allows you to convey the intent and expectations for a specific data type since there is no automatic conversion between values of two user-defined types, even if their item types are identical.

For example, the arithmetic library includes quantum arithmetic operations for both big-endian and little-endian quantum integers. It hence defines two types, `BigEndian` and `LittleEndian`, both of which contain a single anonymous item of type `Qubit[]`:

```
newtype BigEndian = Qubit[];
newtype LittleEndian = Qubit[];
```

These types allow operations to specify whether they are written for big-endian or little-endian representations and leverages the type system to ensure at compile-time that mismatched operands aren't allowed.

Type names must be unique within a namespace and may not conflict with operation and function names. Types may not have circular dependencies in Q#; that is, defining something like a directly or indirectly recursive type is not allowed. For example, the following construct will give a compilation error:

```
newtype Foo = (Foo, Int); // gives an error
newtype Bar = Baz;        // gives an error
newtype Baz = Bar;        // gives an error
```

User-defined constructors

> Constructors for user-defined types are automatically generated by the compiler. Currently, it is not possible to define a custom constructor, though this may be an addition to the language in the future.

Callable declarations

Callable declarations, or *callables*, are declared at a global scope and publicly visible by default; that is, they can be used anywhere in the same project and in a project that references the assembly in which they are declared. Access modifiers allow you to restrict their visibility to the current assembly only, such that implementation details can be changed later on without breaking code that relies on a specific library.

Q# supports two kinds of callables: operations and functions. The topic Operations and Functions elaborates on the distinction between the two. Q# also supports defining *templates*; for example, type-parameterized implementations for a certain callable. For more information, see Type parameterizations.

> [!NOTE] Such type-parametrized implementations may not use any language constructs that rely on particular properties of the type arguments; there is currently no way to express type constraints in Q#, or to define specialized implementations for particular type arguments. However, it is conceivable to introduce a suitable mechanism, similar to type classes in Haskell, for example, to allow for more expressiveness in the future.

Callables and functors

Q# allows specialized implementations for specific purposes; for example, operations in Q# can implicitly or explicitly define support for certain *functors*, and along with it the specialized implementations to invoke when a specific functor is applied to that callable.

A functor, in a sense, is a factory that defines a new callable implementation that has a specific relation to the callable it was applied to. Functors are more than traditional higher-level functions in that they require access to the implementation details of the callable they have been applied to. In that sense, they are similar to other factories, such as templates. Correspondingly, they can be applied not just to callables, but templates as well.

The example program shown in Program implementation, for example, defines the operation `ApplyQFT` and the operation `RunProgram`, which is used as an entry point. `ApplyQFT` takes a tuple-valued argument containing an integer and a value of type `LittleEndian` and returns a value of type `Unit`. The annotation `is`

`Adj + Ctl` in the declaration of `ApplyQFT` indicates that the operation supports both the `Adjoint` and the `Controlled` functor. (For more information, see Operation characteristics). If `Unitary` is an operation that has an adjoint and a controlled specialization, the expression `Adjoint Unitary` accesses the specialization that implements the adjoint of `Unitary`, and `Controlled Unitary` accesses the specialization that implements the controlled version of `Unitary`. In addition to the original operation's argument, the controlled version of an operation takes an array of control qubits and then applies the original operation conditional on all of these control qubits being in a $|1\rangle$ state.

In theory, an operation for which an adjoint version can be defined should also have a controlled version and vice versa. In practice, however, it may be hard to develop an implementation for one or the other, especially for probabilistic implementations following a repeat-until-success pattern. For that reason, Q# allows you to declare support for each functor individually. However, since the two functors commute, an operation that defines support for both also has to have an implementation (usually implicitly defined, meaning compiler-generated) for when both functors are applied to the operation.

There are no functors that can be applied to functions, such that functions currently have exactly one body implementation and no further specializations. For example, the declaration

```
function Hello (name : String) : String {
    return $"Hello, {name}!";
}
```

is equivalent to

```
function Hello (name : String) : String {
    body ( ... ) {
        return $"Hello, {name}!";
    }
}
```

Here, `body` specifies that the given implementation applies to the default body of the function `Hello`, meaning the implementation is invoked when no functors or other factory mechanisms have been applied prior to invocation. The three dots in `body ( ... )` correspond to a compiler directive indicating that the argument items in the function declaration should be copy and pasted into this spot.

The reasons behind explicitly indicating where the arguments of the parent callable declaration are to be copied and pasted are twofold: one, it is unnecessary to repeat the argument declaration, and two, and more importantly, it ensures that functors that require additional arguments, like the `Controlled` functor, can be introduced in a consistent manner.

The same applies to operations; when there is exactly one specialization defining the implementation of the default body, the additional wrapping of the form `body ( ... ){ <implementation> }` may be omitted.

Recursion

Q# callables can be directly or indirectly recursive and can be declared in any order; an operation or function may call itself, or it may call another callable that directly or indirectly calls the caller.

When running on quantum hardware, stack space may be limited, and recursions that exceed that stack space limit result in a runtime error.

Specialization declarations

As explained in the section about callable declarations, there is currently no reason to explicitly declare specializations for functions. This topic applies to operations and elaborates on how to declare the necessary specializations to support certain functors.

It is quite a common problem in quantum computing to require the adjoint of a given transformation. Many quantum algorithms require both an operation and its adjoint to perform a computation. Q# employs sym-

bolic computation that can automatically generate the corresponding adjoint implementation for a particular body implementation. This generation is possible even for implementations that freely mix classical and quantum computations. There are, however, some restrictions that apply in this case. For example, auto-generation is not supported for performance reasons if the implementation makes use of mutable variables. Moreover, each operation called within the body generates the corresponding adjoint needs to support the `Adjoint` functor itself.

Even though one cannot easily undo measurements in the multi-qubit case, it is possible to combine measurements so that the applied transformation is unitary. In this case, it means that, even though the body implementation contains measurements that on their own don't support the `Adjoint` functor, the body in its entirety is adjointable. Nonetheless, auto-generating the adjoint implementation will fail in this case. For this reason, it is possible to manually specify the implementation. The compiler automatically generates optimized implementations for common patterns such as conjugations. Nonetheless, an explicit specialization may be desirable to define a more optimized implementation by hand. It is possible to specify any one implementation and any number of implementations explicitly.

> [!NOTE] The correctness of such a manually specified implementation is not verified by the compiler.

In the following example, the declaration for an operation `SWAP`, which exchanges the state of two qubits `q1` and `q2`, declares an explicit specialization for its adjoint version and its controlled version. While the implementations for `Adjoint SWAP` and `Controlled SWAP` are thus user-defined, the compiler still needs to generate the implementation for the combination of both functors (`Controlled Adjoint SWAP`, which is the same as `Adjoint Controlled SWAP`).

```
operation SWAP (q1 : Qubit, q2 : Qubit) : Unit
is Adj + Ctl {

    body ( ... ) {
        CNOT(q1, q2);
        CNOT(q2, q1);
        CNOT(q1, q2);
    }

    adjoint ( ... ) {
        SWAP(q1, q2);
    }

    controlled (cs, ... ) {
        CNOT(q1, q2);
        Controlled CNOT(cs, (q2, q1));
        CNOT(q1, q2);
    }
}
```

Auto-generation directives

When determining how to generate a particular specialization, the compiler prioritizes user-defined implementations. This means that if an adjoint specialization is user-defined and a controlled specialization is auto-generated, then the controlled adjoint specialization is generated based on the user-defined adjoint and vice versa. In this case, both specializations are user-defined. As the auto-generation of an adjoint implementation is subject to more limitation, the controlled adjoint specialization defaults to generating the controlled specialization of the explicitly defined implementation of the adjoint specialization.

In the case of the `SWAP` implementation, the better option is to adjoint the controlled specialization to avoid unnecessarily conditioning the execution of the first and the last `CNOT` on the state of the control qubits. Adding an explicit declaration for the controlled adjoint version that specifies a suitable *generation directive* forces the

compiler to generate the controlled adjoint specialization based on the manually specified implementation of the controlled version instead. Such an explicit declaration of a specialization that is to be generated by the compiler takes the form

```
    controlled adjoint invert;
```

and is inserted inside the declaration of `SWAP`. On the other hand, inserting the line

```
    controlled adjoint distribute;
```

forces the compiler to generate the specialization based on the defined (or generated) adjoint specialization. See this partial specialization inference proposal for more details.

For the operation `SWAP`, there is a better option. `SWAP` is *self-adjoint*, that is, it is its own inverse; the -defined implementation of the adjoint merely calls the body of `SWAP`. You express this with the directive

```
    adjoint self;
```

Declaring the adjoint specialization in this manner ensures that the controlled adjoint specialization that is automatically inserted by the compiler merely invokes the controlled specialization.

The following generation directives exist and are valid:

Specialization	Directive(s)
`body` specialization:	-
`adjoint` specialization:	`self`, `invert`
`controlled` specialization:	`distribute`
`controlled adjoint` specialization:	`self`, `invert`, `distribute`

That all generation directives are valid for a controlled adjoint specialization is not a coincidence; as long as functors commute, the set of valid generation directives for implementing the specialization for a combination of functors is always the union of the set of valid generators for each one.

In addition to the previously listed directives, the directive `auto` is always valid; it indicates that the compiler should automatically pick a suitable generation directive. The declaration

```
    operation DoNothing() : Unit {
        body (...) { }
        adjoint auto;
        controlled auto;
        controlled adjoint auto;
    }
```

is equivalent to

```
    operation DoNothing() : Unit
    is Adj + Ctl { }
```

The annotation `is Adj + Ctl` in this example specifies the *operation characteristics*, which contain the information about what functors a particular operation supports.

While for readability's sake, it is recommended that you annotate each operation with a complete description of its characteristics, the compiler automatically inserts or completes the annotation based on explicitly declared specializations. Conversely, the compiler also generates specializations that haven't been declared explicitly but need to exist based on the annotated characteristics. We say the given annotation has *implicitly declared* these specializations. The compiler automatically generates the necessary specializations if it can, picking a suitable directive. Q# thus supports inference of both operation characteristics and existing specializations based on (partial) annotations and explicitly defined specializations.

In a sense, specializations are similar to individual overloads for the same callable, with the caveat that certain restrictions apply to which overloads you can declare.

Comments

Comments begin with two forward slashes (`//`) and continue until the end of line. Such end-of-line comments may appear anywhere in the source code. Q# does not currently support block comments.

Documentation Comments

Comments that begin with three forward slashes, `///`, are treated specially by the compiler when they appear before a type or callable declaration. In that case, their contents are taken as documentation for the defined type or callable, as for other .NET languages.

Within `///` comments, text to appear as a part of API documentation is formatted as Markdown, with different parts of the documentation indicated by specially-named headers. As an extension to Markdown, cross-references to operations, functions, and user-defined types in Q# can be included using `@"<ref target>,"` where `<ref target>` is replaced by the fully qualified name of the code object being referenced. Optionally, a documentation engine may also support additional Markdown extensions.

For example:

```
/// # Summary
/// Given an operation and a target for that operation,
/// applies the given operation twice.
///
/// # Input
/// ## op
/// The operation to be applied.
/// ## target
/// The target to which the operation is to be applied.
///
/// # Type Parameters
/// ## 'T
/// The type expected by the given operation as its input.
///
/// # Example
/// ```Q#
/// // Should be equivalent to the identity.
/// ApplyTwice(H, qubit);
/// ```
///
/// # See Also
/// - Microsoft.Quantum.Intrinsic.H
operation ApplyTwice<'T>(op : ('T ⇒ Unit), target : 'T) : Unit {
    op(target);
    op(target);
}
```

Q# recognizes the following names as documentation comment headers.

- **Summary**: A short summary of a function or operation's behavior or the purpose of a type. The first paragraph of the summary is used for hover information. It should be plain text.
- **Description**: A description of a function or operation's behavior or the purpose of a type. The summary and description are concatenated to form the generated documentation file for the function, operation, or type. The description may contain in-line LaTeX-formatted symbols and equations.

- **Input**: A description of the input tuple for an operation or function. May contain additional Markdown subsections indicating each element of the input tuple.
- **Output**: A description of the tuple returned by an operation or function.
- **Type Parameters**: An empty section that contains one additional subsection for each generic type parameter.
- **Named Items**: A description of the named items in a user-defined type. May contain additional Markdown subsections with the description for each named item.
- **Example**: A short example of the operation, function, or type in use.
- **Remarks**: Miscellaneous prose describing some aspect of the operation, function, or type.
- **See Also**: A list of fully qualified names indicating related functions, operations, or user-defined types.
- **References**: A list of references and citations for the documented item.

Statements

Q# distinguishes between statements and expressions. Q# programs consist of a mixture of classical and quantum computations, and the implementation looks much like any other classical programming language. Some statements, such as the `let` and `mutable` bindings, are well-known from classical languages, while others, such as conjugations or qubit allocations, are unique to the quantum domain.

The following statements are currently available in Q#:

- **Call statement** A call statement consists of an operation or function call returning `Unit`. The invoked callable needs to satisfy the requirements imposed by the current context. See Call statements for more details.
- **Return statement** A return statement terminates the execution within the current callable context and returns control to the caller. Any finalizing tasks are run after the return value is evaluated but before control is returned. See Returns and termination for more details.
- **Fail statement** A fail statement stops the entire program and collects information about the current program state before terminating in an error. It aggregates the collected information and presents it to the user along with the message specified as part of the statement. See returns and termination for more details.
- **Variable declaration** Defines one or more local variables that are valid for the remainder of the current scope, and binds them to the specified values. Variables can be permanently bound or declared to be reassignable later on. See Variable declarations and reassignments for more details.
- **Variable reassignment** Variables that have been declared as being reassignable can be rebound to contain different values. See Variable declarations and reassignments for more details.
- **Iteration** An iteration is a loop-like statement that, during each iteration, assigns the declared loop variables to the next item in a sequence (a value of array or `Range` type) and runs a specified block of statements. See Iterations for more details.
- **While statement** If a specified condition evaluates to `true`, a block of statements is run. The statements are run until the condition evaluates to `false`. See Conditional loops for more details.
- **Repeat statement** A quantum-specific loop that breaks based on a condition. The statement consists of an initial block of statements that is run before a specified condition is evaluated. If the condition evaluates to `false`, an optional subsequent `fixup` block is run before entering the next iteration of the loop. The loop only terminates when the condition evaluates to `true`. See Conditional loops for more details.
- **If statement** The if statement consists of one or more blocks of statements, each preceded by a boolean expression. The first block in which the boolean expression evaluates to `true` is run. Optionally, you can specify a block of statements that will run if none of the conditions evaluate to `true`. See Conditional branching for more details.

- **Conjugation** A conjugation is a special quantum-specific statement, where a block of statements that applies a unitary transformation to the quantum state is run, followed by another statement block, before the transformation applied by the first block is reverted again. In mathematical notation, conjugations describe transformations of the form $U\dagger VU$ to the quantum state. See Conjugations for more details.
- **Qubit allocation** Instantiates and initializes qubits, or arrays of qubits, binds them to the declared variables and runs a block of statements. The instantiated qubits are available for the duration of the block, and will be automatically released when the statement terminates. See Quantum memory management for more details.

Quantum memory management

A program always starts without qubits, meaning you cannot pass values of type `Qubit` as entry point arguments. This restriction is intentional since the purpose of Q# is to express and reason about a program in its entirety. Instead, a program allocates and releases qubits, or quantum memory, as it goes. In this regard, Q# models the quantum computer as a qubit heap.

Rather than supporting separate *allocate* and *release* statements for quantum memory, Q# supports quantum memory allocation in the form of *block statements*, where the memory is accessible only within the scope of that block statement. The statement block can be implicitly defined when allocating qubits for the duration of the current scope, as described in more detail in the sections about the `use` and `borrow` statements. Attempting to access the allocated qubits after the statement terminates results in a runtime exception.

Q# has two statements, `use` and `borrow`, that instantiate qubit values, arrays of qubits, or any combination thereof. You can only use these statements within operations. They gather the instantiated qubit values, bind them to the variables specified in the statement, and then run a block of statements. At the end of the block, the bound variables go out of scope and are no longer defined.

Q# distinguishes between the allocation of *clean* and *dirty* qubits. Clean qubits are unentangled and are not used by another part of the computation. Dirty qubits are qubits whose state is unknown and can even be entangled with other parts of the quantum processor's memory.

Use statement

Clean qubits are allocated by the `use` statement.

- The statement consists of the keyword `use` followed by a binding and an optional statement block.
- If a statement block is present, the qubits are only available within that block. Otherwise, the qubits are available until the end of the current scope.
- The binding follows the same pattern as `let` statements: either a single symbol or a tuple of symbols, followed by an equals sign `=`, and either a single tuple or a matching tuple of *initializers*.

Initializers are available either for a single qubit, indicated as `Qubit()`, or an array of qubits, `Qubit[n]`, where `n` is an `Int` expression. For example,

```
use qubit = Qubit();
// ...

use (aux, register) = (Qubit(), Qubit[5]);
// ...

use qubit = Qubit() {
    // ...
}

use (aux, register) = (Qubit(), Qubit[5]) {
```

```
    // ...
}
```

The qubits are guaranteed to be in a $|0\rangle$ state upon allocation. They are released at the end of the scope and are required to either be in a $|0\rangle$ state upon release, or measured right beforehand. This requirement is not compiler-enforced since this would require a symbolic evaluation that quickly gets prohibitively expensive. When running on simulators, the requirement can be runtime enforced. On quantum processors, the requirement cannot be runtime enforced; an unmeasured qubit may be reset to $|0\rangle$ via unitary transformation. Failing to do so results in incorrect behavior.

The `use` statement allocates the qubits from the quantum processor's free qubit heap and returns them to the heap no later than the end of the scope in which the qubits are bound.

Borrow statement

The `borrow` statement grants access to qubits that are already allocated but not currently in use. These qubits can be in an arbitrary state and need to be in the same state again when the borrow statement terminates. Some quantum algorithms can use qubits without relying on their exact state, and without requiring that they are unentangled with the rest of the system. That is, they require extra qubits temporarily, but they can ensure that those qubits are returned exactly to their original state, independent of which state that was.

If there are qubits that are in use but not touched during parts of a subroutine, those qubits can be borrowed for use by such an algorithm instead of allocating additional quantum memory. Borrowing instead of allocating can significantly reduce the overall quantum memory requirements of an algorithm and is a quantum example of a typical space-time tradeoff.

A `borrow` statement follows the same pattern described previously for the `use` statement, with the same initializers being available. For example,

```
borrow qubit = Qubit();
// ...

borrow (aux, register) = (Qubit(), Qubit[5]);
// ...

borrow qubit = Qubit() {
    // ...
}

borrow (aux, register) = (Qubit(), Qubit[5]) {
    // ...
}
```

The borrowed qubits are in an unknown state and go out of scope at the end of the statement block. The borrower commits to leaving the qubits in the same state as when they were borrowed; that is, their state at the beginning and the end of the statement block is expected to be the same.

The `borrow` statement retrieves in-use qubits that are guaranteed not to be used by the program from the time the qubit is bound until the last use of that qubit. If there aren't enough qubits available to borrow, then qubits are allocated from and returned to the heap like a `use` statement.

> [!NOTE] Among the known use-cases of dirty qubits are implementations of multi-controlled CNOT gates that require very few qubits, and implementations of incrementers. This paper on factoring with qubits provides an example of an algorithm that utilizes borrowed qubits.

Conditional branching

Conditional branching is expressed in the form of `if` statements. An `if` statement consists of an `if` clause, followed by zero or more `elif` clauses and optionally an else-block. Each clause follows the pattern

```
keyword condition {
    <statements>
}
```

where `keyword` is replaced with `if` or `elif` respectively, `condition` is an expression of type `Bool`, and `<statements>` is to be replaced with zero or more statements. The optional `else`-block consists of the keyword `else` followed by zero or more statements enclosed in braces, `{ }`.

The first block for which the `condition` evaluates to `true` will run. The `else` block, if present, runs if none of the conditions evaluate to `true`. The block is executed in its own scope, meaning any bindings made as part of the statement block are not visible after the block ends.

For example, suppose `qubits` is value of type `Qubit[]` and `r1` and `r2` are of type `Result`,

```
if r1 == One {
    let q = qubits[0];
    H(q);
}
elif r2 == One {
    let q = qubits[1];
    H(q);
}
else {
    H(qubits[2]);
}
```

You can also express simple branching in the form of a conditional expression.

Target-specific restrictions

The tight integration between control-flow constructs and quantum computations poses a challenge for current quantum hardware. Certain quantum processors do not support branching based on measurement outcomes. As such, comparison for values of type `Result` will always result in a compilation error for Q# programs that are targeted to run on such hardware.

Other quantum processors support specific kinds of branching based on measurement outcomes. The more general `if` statements supported in Q# are compiled into suitable instructions that can be run on such processors. The imposed restrictions are that values of type `Result` may only be compared as part of the condition within `if` statements in operations. Furthermore, the conditionally run blocks cannot contain any return statements or update mutable variables that are declared outside that block.

Conditional loops

Like most classical programming languages, Q# supports loops that break based on a condition: loops for which the number of iterations is unknown and may vary from run to run. Since the instruction sequence is unknown at compile-time, the compiler handles these conditional loops in a particular way in a quantum runtime.

As long as the condition does not depend on quantum measurements, conditional loops are processed with a just-in-time compilation before sending the instruction sequence to the quantum processor. In particular, using conditional loops within functions is unproblematic since code within functions can always run on conventional (non-quantum) hardware. Q#, therefore, supports to use of traditional `while` loops within functions.

Q# also allows you to express control flow that depends on the results of quantum measurements. This capability enables probabilistic implementations that can significantly reduce computational costs. A common example is the *repeat-until-success* pattern, which repeats a computation until a certain condition - which usually depends on a measurement - is satisfied. Such `repeat` loops are widely used in particular classes of quantum algorithms. Q# hence has a dedicated language construct to express them, despite that they still pose a challenge for execution on quantum hardware.

Repeat statement

The `repeat` statement takes the following form

```
repeat {
    // ...
}
until condition
fixup {
    // ...
}
```

or alternatively

```
repeat {
    // ...
}
until condition;
```

where `condition` is an arbitrary expression of type `Bool`.

The `repeat` statement runs a block of statements before evaluating a condition. If the condition evaluates to true, the loop exits. If the condition evaluates to false, an additional block of statements defined as part of an optional `fixup` block, if present, is run prior to entering the next loop iteration.

The compiler treats all parts of the `repeat` statement (both blocks and the condition) as a single scope for each repetition; symbols that are defined within the `repeat` block are visible both to the condition and within the `fixup` block. As for other loops, symbols go out of scope after each iteration, such that symbols defined in the `fixup` block are not visible in the `repeat` block.

Target-specific restrictions Loops that break based on a condition are challenging to process on quantum hardware if the condition depends on measurement outcomes since the length of the instruction sequence to run is not known ahead of time.

Despite their common presence in particular classes of quantum algorithms, current hardware does not yet provide native support for these kinds of control flow constructs. Running on quantum hardware can potentially be supported in the future by imposing a maximum number of iterations, or as additional hardware support becomes available.

While loop

A more familiar-looking statement for classical computations is the `while` loop. It is supported only within functions.

A `while` statement consists of the keyword `while`, an expression of type `Bool`, and a statement block. For example, if `arr` is an array of positive integers,

```
mutable (item, index) = (-1, 0);
while index < Length(arr) && item < 0 {
    set item = arr[index];
```

```
    set index += 1;
}
```

The statement block is run as long as the condition evaluates to `true`.

> [!NOTE] Due to the challenge they pose for execution, we discourage the use of loops that break based on a condition and hence do not support `while` loops within operations. The use of `while` loops within operations may be considered in the future, with the restriction that the condition cannot depend on the outcome of a quantum measurement.

Iterations

Loops that iterate over a sequence of values are expressed as `for` loops in Q#. A `for` loop in Q# does not break based on a condition but instead corresponds to an iteration, or what is often expressed as `foreach` or `iter` in other languages. There are currently two data types in Q# that support iteration: arrays and ranges.

The statement consists of the keyword `for`, followed by a symbol or symbol tuple, the keyword `in`, an expression of array or `Range` type, and a statement block.

The statement block (the body of the loop) is run repeatedly, with one or more loop variables bound to each value in the range or array. The same deconstruction rules apply to the defined loop variables as to any other variable assignment, such as bindings in `let`, `mutable`, `set`, `use` and `borrow` statements. The loop variables themselves are immutably bound, cannot be reassigned within the body of the loop, and go out of scope when the loop terminates. The expression over which the loop iterates is evaluated before entering the loop and does not change while the loop is running.

This is illustrated in the following example. Suppose `qubits` is a value of type `Qubit[]`, then

```
for qubit in qubits {
    H(qubit);
}

mutable results = new (Int, Result)[0];
for index in 0 .. Length(qubits) - 1 {
    set results += [(index, M(qubits[index]))];
}

mutable accumulated = 0;
for (index, measured) in results {
    if measured == One {
        set accumulated += 1 <<< index;
    }
}
```

Target-specific restrictions

Because there are no `break` or `continue` primitives in Q#, the length of the loop is known as soon as the iteration value is known. As such, `for` loops can be run on all quantum hardware.

Conjugations

Conjugations are common in quantum computations. In mathematical terms, they are patterns of the form $UtVU$ for two unitary transformations U and V. That pattern is relevant due to the particularities of quantum memory: computations build up quantum correlations, or *entanglement*, to leverage the unique assets of quantum. However, that also means that once a subroutine no longer needs its qubits, those qubits cannot easily be reset and released since observing their state would impact the rest of the system. For that reason,

the effect of a previous computation usually needs to be reversed before releasing and reusing quantum memory.

Q# hence has a dedicated statement for expressing computations that require such a cleanup. The statement consists of two code blocks, one containing the implementation of U and one containing the implementation of V. The *uncomputation* (that is, the application of $U\dagger$) is done automatically as part of the statement.

The statement takes the form

```
within {
    <statements>
}
apply {
    <statements>
}
```

where `<statements>` is replaced with any number of statements defining the implementation of U and V respectively. Both blocks may contain arbitrary classical computations, aside from the usual restrictions for automatically generating adjoint versions that apply to the `within` block. Mutably bound variables used as part of the `within` block may not be reassigned as part of the `apply` block.

The example of the `ApplyXOrIfGreater` operation defined in the arithmetic library illustrates the usage of such a conjugation: The operation maps $|\text{lhs}\rangle|\text{rhs}\rangle|\text{res}\rangle \rightarrow |\text{lhs}\rangle|\text{rhs}\rangle|\text{res} \oplus (\text{lhs}>\text{rhs})\rangle$, that is, it coherently applies an XOR to a given qubit `res` if the quantum integer represented by `lhs` is greater than the one in `rhs`. The two integers are represented in little-endian encoding, as indicated by the usage of the corresponding data type.

```
    operation ApplyXOrIfGreater(
        lhs : LittleEndian,
        rhs : LittleEndian,
        res : Qubit
    ) : Unit is Adj + Ctl {

        let (x, y) = (lhs!, rhs!);
        let shuffled = Zip3(Most(x), Rest(y), Rest(x));

        use anc = Qubit();
        within {
            ApplyToEachCA(X, x + [anc]);
            ApplyMajorityInPlace(x[0], [y[0], anc]);
            ApplyToEachCA(MAJ, shuffled);
        }
        apply {
            X(res);
            CNOT(Tail(x), res);
        }
    }
```

The temporary storage qubit `anc` is automatically cleaned up before it is released at the end of the operation. The statements in the `within` block are applied first, followed by the statements in the `apply` block, and finally, the automatically generated adjoint of the `within` block is applied to clean up the temporary qubit `anc`.

> [!NOTE] Returning control from within the `apply` block is not yet supported. However, it may be supported in the future. The expected behavior, in this case, is to evaluate the returned value before the adjoint of the `within` block is run, then release any qubits going out of scope (`anc` in this case), and finally, return control to the callee. In short, the statement should behave similarly to a `try-finally` pattern in C#.

Call statements

Call statements are an important part of any programming language. Operation and function calls, much like partial applications, can be used as an expression anywhere as long as the returned value is of a suitable type. However, they can also be used as statements if they return `Unit`.

The usefulness of calling functions in this form primarily lies in debugging, and such operation calls are one of the most common constructs in any Q# program. At the same time, operations can only be called from within other operations and not from within functions. For more information, see also Qubits.

With callables being first-class values, call statements are a generic way of supporting patterns that aren't common enough to merit their own dedicated language construct, or dedicated syntax has not (yet) been introduced for other reasons. Some examples of library methods that serve that purpose are `ApplyIf`, which invokes an operation conditional on a classical bit being set; `ApplyToEach`, which applies a given operation to each element in an array; and `ApplyWithInputTransformation`, as shown in the following sample.

```
operation ApplyWithInputTransformation<'TArg, 'TIn>(
    fn : 'TIn → 'TArg,
    op : 'TArg ⇒ Unit,
    input : 'TIn
) : Unit {

    op(fn(input));
}
```

`ApplyWithInputTransformation` takes a function `fn`, an operation `op`, and an `input` value as arguments and then applies the given function to the input before invoking the given operation with the value returned from the function.

For the compiler to auto-generate the specializations to support particular functors, it usually requires that the called operations support those functors as well. The two exceptions are calls in outer blocks of conjugations, which always need to support the `Adjoint` functor but never need to support the `Controlled` functor, and self-adjoint operations, which support the `Adjoint` functor without imposing any additional requirements on the individual calls.

> [!NOTE] Future development of more sophisticated generation directives may allow the compiler to further relax this requirement.

Returns and termination

There are two statements available that conclude the execution of the current subroutine or the program; the `return` and the `fail` statements. For callables that return any type other than `Unit`, each possible execution path needs to terminate in a `return` or a `fail` statement.

Return statement

The `return` statement exits from the current callable and returns control to the callee. It changes the context of the execution by popping a stack frame.

The statement always returns a value to the context of the callee; it consists of the keyword `return`, followed by an expression of the appropriate type and a terminating semicolon. The return value is evaluated before any terminating actions are performed and control is returned. Terminating actions include, for example, cleaning up and releasing qubits that are allocated within the context of the callable. When running on a simulator or validator, terminating actions often also include checks related to the state of those qubits, for example, whether they are properly disentangled from all qubits that remain live.

The `return` statement at the end of a callable that returns a `Unit` value may be omitted. In that case, control is returned automatically when all statements have run and all terminating actions have been performed.

Callables may contain multiple `return` statements, one for each possible execution path, albeit the adjoint implementation for operations containing multiple `return` statements cannot be automatically generated.

For example,

```
return 1;
```

or

```
return ();
```

Fail statement

The `fail` statement, on the other hand, ends the computation entirely. It corresponds to a fatal error that aborts the program.

It consists of the keyword `fail`, followed by an expression of type `String` and a terminating semicolon. The `String` value provides information about the encountered failure.

For example,

```
fail "Impossible state reached";
```

or, using an interpolated string,

```
fail $"Syndrome {syn} is incorrect";
```

In addition to the given `String`, a `fail` statement ideally collects and permits the retrieval of information about the program state. This facilitates diagnosing and remedying the source of the error, and requires support from the executing runtime and firmware, which may vary across different targets.

Variable declarations and reassignments

Values can be bound to symbols using the `let` and `mutable` statements. These kinds of bindings provide a convenient way to access a value via the defined handle. Despite the misleading terminology borrowed from other languages, handles declared on a local scope and containing values are called *variables*. This may be misleading because `let` statements define *single-assignment handles*, which are handles that remain bound to the same value for the duration of their validity. Variables that can be re-bound to different values at different points in the code need to be explicitly declared as such, and are specified using the `mutable` statement.

```
    let var1 = 3;
    mutable var2 = 3;
    set var2 = var2 + 1;
```

In this example, the `let` statement declares a variable named `var1` that cannot be reassigned and always contains the value `3`. The `mutable` statement defines a variable `var2` that is temporarily bound to the value `3` but can be reassigned to a different value later on using a `set` statement, as shown in the last line. You can express the same statement with the shorter version `set var2 += 1;`, as is common in other languages. For more information, see Evaluate and reassign statements.

To summarize:

- `let` is used to create an immutable binding.
- `mutable` is used to create a mutable binding.
- `set` is used to change the value of a mutable binding.

For all three statements, the left-hand side consists of a symbol or a symbol tuple. If the right-hand side of the binding is a tuple, then that tuple may be fully or partially deconstructed upon assignment. The only requirement for deconstruction is that the shape of the tuple on the right-hand side matches the shape of the symbol tuple on the left side. The symbol tuple may contain nested tuples or omitted symbols, or both, indicated by an underscore. For example:

```
let (a, (_, b)) = (1, (2, 3)); // a is bound to 1, b is bound to 3
mutable (x, y) = ((1, 2), [3, 4]); // x is bound to (1, 2), y is bound to [3, 4]
set (x, _, y) = ((5, 6), 7, [8]);  // x is re-bound to (5,6), y is re-bound to [8]
```

All assignments in Q# obey the same deconstruction rules, including, for example, qubit allocations and loop-variable assignments.

For both kinds of bindings, the types of the variables are inferred from the right-hand side of the binding. The type of a variable always remains the same, and a `set` statement cannot change it. Local variables can be declared as either being mutable or immutable. There are some exceptions, such as loop-variables in `for` loops, where the behavior is predefined and cannot be specified. Function and operation arguments are always immutably bound. In combination with the lack of reference types, as discussed in the Immutability topic, this means that a called function or operation can never change any values on the caller side.

Since the states of `Qubit` values are not defined or observable from within Q#, this does not preclude the accumulation of quantum side effects, which are observable only via measurements. For more information, see Quantum data types).

Independent of how a value is bound, the values themselves are immutable. In particular, this is true for arrays and array items. In contrast to popular classical languages where arrays often are reference types, arrays in Q# - like all types - are value types and always immutable; that is, you cannot modify them after initialization. Changing the values accessed by array-type variables thus requires explicitly constructing a new array and reassigning it to the same symbol. For more information, see Immutability and Copy and update expressions.

Evaluate-and-reassign statements

Statements of the form `set intValue += 1;` are common in many other languages. Here, `intValue` needs to be a mutably bound variable of type `Int`. Such statements provide a convenient way of concatenation if the right-hand side consists of applying a binary operator and the result is rebound to the left argument of the operator. For example, this code segment

```
mutable counter = 0;
for i in 1 .. 2 .. 10 {
    set counter += 1;
    // ...
}
```

increments the value of the counter `counter` in each iteration of the `for` loop and is equivalent to

```
mutable counter = 0;
for i in 1 .. 2 .. 10 {
    set counter = counter + 1;
    // ...
}
```

Similar statements exist for a wide range of operators. The `set` keyword in such evaluate-and-reassign statements must be followed by a single mutable variable, which is inserted as the left-most sub-expression by the compiler. Such evaluate-and-reassign statements exist for all operators where the type of the left-most sub-expression matches the expression type. More precisely, they are available for binary logical and bitwise operators including right and left shift, arithmetic expressions including exponentiation and modulus, and concatenations, as well as copy-and-update expressions.

The following function example computes the sum of an array of `Complex` numbers:

```
function ComplexSum(values : Complex[]) : Complex {
    mutable res = Complex(0., 0.);
    for complex in values {
```

```
        set res w/= Re <- res::Re + complex::Re;
        set res w/= Im <- res::Im + complex::Im;
    }
    return res;
}
```

Similarly, the following function multiplies each item in an array with the given factor:

```
function Multiplied(factor : Double, array : Double[]) : Double[] {
    mutable res = new Double[Length(array)];
    for i in IndexRange(res) {
        set res w/= i <- factor * array[i];
    }
    return res;
}
```

For more information, see Contextual expressions, which contains other examples where expressions can be omitted in a specific context when a suitable expression can be inferred by the compiler.

Binding scopes

In general, symbol bindings in Q# become inoperative at the end of the statement block they occur in. However, there are some exceptions to this rule.

Visibility of local variables

Scope	Visibility
Loop variables	Bindings of loop variables in a `for` loop are defined only for the body of the loop. They are inoperative outside of the loop.
`use` and `borrow` statements	Bindings of allocated qubits in `use` and `borrow` statements are defined for the body of the allocation, and are inoperative after the statement terminates. This only applies to `use` and `borrow` statements that have an associated statement block.
`repeat` statements	For `repeat` statements, both blocks, as well as the condition, are treated as a single scope, that is, symbols that are bound in the body are accessible in both the condition and in the `fixup` block.
Loops	Each iteration of a loop runs in its own scope, and all defined variables are bound anew for each iteration.

Bindings in outer blocks are visible and defined in inner blocks. A symbol may only be bound once per block; it is not valid to define a symbol with the same name as another symbol that is accessible (no "shadowing").

The following sequences are valid:

```
if a == b {
    ...
    let n = 5;
    ...             // n is 5
}
let n = 8;
...                 // n is 8
```

and

```
if a == b {
    ...
    let n = 5;
    ...             // n is 5
} else {
    ...
    let n = 8;
    ...             // n is 8
}
...                 // n is not bound to a value
```

The following sequences are invalid:

```
let n = 5;
...                 // n is 5
let n = 8;          // Error
...
```

and

```
let n = 8;
if a == b {
    ...             // n is 8
    let n = 5;      // Error
    ...
}
...
```

For more details, see Variable Declarations and Reassignments.

Expressions

At the core, Q# expressions are either value literals or identifiers, where identifiers can refer to either locally declared variables or to globally declared callables (there are currently no global constants in Q#). Operators, combinators, and modifiers can be used to combine these into a wider variety of expressions.

- *Operators* in a sense are nothing but dedicated syntax for particular callables.

  > Even though Q# is not yet expressive enough to formally capture the capabilities of each operator in the form of a backing callable declaration, that should be remedied in the future.

- *Modifiers* can only be applied to certain expressions. One or more modifiers can be applied to expressions that are either identifiers, array item access expressions, named item access expressions, or an expression within parenthesis which is the same as a single item tuple (see this section). They can either precede (prefix) the expression or follow (postfix) the expression. They are thus special unary operators that bind tighter than function or operation calls, but less tight than any kind of item access. Concretely, functors are prefix modifiers, whereas the unwrap operator is a postfix modifier.
- Like modifiers, function and operation calls as well as item access can be seen as a special kind of operator subject to the same restrictions regarding where they can be applied; we refer to them as *combinators*.

The section on precedence and associativity contains a complete list of all operators as well as a complete list of all modifiers and combinators.

Precedence and associativity

Precedence and associativity define the order in which operators are applied. Operators with higher precedence are bound to their arguments (operands) first, while operators with the same precedence bind in the direction of their associativity. For example, the expression `1+2*3` according to the precedence for addition and multiplication is equivalent to `1+(2*3)`, and `2^3^4` equals `2^(3^4)` since exponentiation is right-associative.

Operators

The following table lists the available operators in Q#, as well as their precedence and associativity. Additional modifiers and combinators are also listed, and bind tighter than any of these operators.

Description	Syntax	Operator	Associativity	Precedence
copy-and-update operator	`w/ ←`	ternary	left	1
range operator	`..`	infix	left	2
conditional operator	`? |`	ternary	right	5
logical OR	`or`	infix	left	10
logical AND	`and`	infix	left	11
bitwise OR	`|||`	infix	left	12
bitwise XOR	`^^^`	infix	left	13
bitwise AND	`&&&`	infix	left	14
equality	`=`	infix	left	20
inequality	`≠`	infix	left	20
less-than-or-equal	`⩽`	infix	left	25
less-than	`<`	infix	left	25
greater-than-or-equal	`⩾`	infix	left	25
greater-than	`>`	infix	left	25
right shift	`>>>`	infix	left	28
left shift	`<<<`	infix	left	28
addition or concatenation	`+`	infix	left	30
subtraction	`-`	infix	left	30
multiplication	`*`	infix	left	35
division	`/`	infix	left	35
modulus	`%`	infix	left	35
exponentiation	`^`	infix	right	40
bitwise NOT	`~~~`	prefix	right	45
logical NOT	`not`	prefix	right	45
negative	`-`	prefix	right	45

Copy-and-update expressions necessarily need to have the lowest precedence to ensure a consistent behavior of the corresponding evaluate-and-reassign statement. Similar considerations hold for the range operator to ensure a consistent behavior of the corresponding contextual expression.

Copy-and-update expressions

To reduce the need for mutable bindings, Q# supports copy-and-update expressions for value types with item access. User-defined types and arrays both are immutable and fall into this category. User-defined

types allow you to access items via name, whereas arrays allow you to access items via an index or range of indices.

Copy-and-update expressions instantiate a new value with all items set to the corresponding value in the original expression, except certain specified items(s), which are set to the one(s) defined on the right-hand side of the expression. They are constructed using a ternary operator `w/ ←`; the syntax `w/` should be read as the commonly used short notation for "with":

```
    original w/ itemAccess <- modification
```

where `original` is either an expression of user-defined type or an array expression. For the corresponding requirements for `itemAccess` and `modification`, see Copy-and-update of user-defined types and Copy-and-update of arrays.

In terms of precedence, the copy-and-update operator is left-associative and has lowest precedence, and, in particular, lower precedence than the range operator (`..`) or the ternary conditional operator (`? |`). The chosen left associativity allows easy chaining of copy-and-update expressions:

```
    let model = Default<SequentialModel>()
                    w/ Structure <- ClassifierStructure()
                    w/ Parameters <- parameters
                    w/ Bias <- bias;
```

As for any operator that constructs an expression of the same type as the left-most expression involved, the corresponding evaluate-and-reassign statement is available. The two following statements, for example, achieve the following: The first statement declares a mutable variable `arr` and binds it to the default value of an integer array. The second statement then builds a new array with the first item (with index 0) set to 10 and reassigns it to `arr`.

```
    mutable arr = [0, size = 3]; // arr contains [0,0,0]
    set arr w/= 0 <- 10;      // arr contains [10,0,0]
```

The second statement is just short-hand for the more verbose syntax `set arr = arr w/ 0 ← 10;`.

Copy-and-update of user-defined types If the value `original` is a user-defined type, then `itemAccess` denotes the name of the item that diverges from the original value. This is not just another expression, like `original` and `modification`, because the ability to use the item name without any further qualification is limited to this context; it is one of two contextual expressions in Q#.

The type of the `modification` expression needs to match the type of the named item that diverges. For instance, if `complex` contains the value `Complex(0., 0.)`, where the type `Complex` is defined here, then

```
complex w/ Re <- 1.
```

is an expression of type `Complex` that evaluates to `Complex(1.,0.)`.

Copy-and-update of arrays For arrays, `itemAccess` can be an arbitrary expression of a suitable type; the same types that are valid for array slicing are valid in this context. Concretely, the `itemAccess` expression can be of type `Int` or `Range`. If `itemAccess` is a value of type `Int`, then the type of `modification` has to match the item type of the array. If `itemAccess` is a value of type `Range`, then the type of `modification` has to be the same as the array type.

For example, if `arr` contains an array `[0,1,2,3]`, then

- `arr w/ 0 ← 10` is the array `[10,1,2,3]`.
- `arr w/ 2 ← 10` is the array `[0,1,10,3]`.
- `arr w/ 0..2..3 ← [10,12]` is the array `[10,1,12,3]`.

Copy-and-update expressions allow the efficient creation of new arrays based on existing ones. The implementation for copy-and-update expressions avoids copying the entire array by duplicating only the necessary parts to achieve the desired behavior and performs an in-place modification if possible. Hence, array initializations do not incur additional overhead due to immutability.

The `Microsoft.Quantum.Arrays` namespace provides an arsenal of convenient tools for array creation and manipulation. For example, the function `ConstantArray` creates an array of the specified length and initializes each item to a given value.

Copy-and-update expressions are a convenient way to construct new arrays on the fly; the following expression, for example, evaluates to an array with all items set to `PauliI`, except the item at index `i`, which is set to `PauliZ`:

```
ConstantArray(n, PauliI) w/ i <- PauliZ
```

Conditional expressions

Conditional expressions consist of three sub-expressions, where the left-most sub-expression is of type `Bool` and determines which one of the two other sub-expressions is evaluated. They are of the form

```
cond ? ifTrue | ifFalse
```

Specifically, if `cond` evaluates to `true`, then the conditional expression evaluates to the `ifTrue` expression; otherwise, it evaluates to the `ifFalse` expression. The other expression (the `ifFalse` and `ifTrue` expression, respectively) is never evaluated, much like the branches in an `if` statement. For instance, in an expression `a == b ? C(qs) | D(qs)`, if `a` equals `b`, then the callable `C` is invoked. Otherwise, `D` is invoked.

The types of the `ifTrue` and the `ifFalse` expression have to have a common base type. Independent of which one ultimately yields the value to which the expression evaluates, its type always matches the determined base type.

For example, if

- `Op1` is of type `Qubit[] => Unit is Adj`
- `Op2` is of type `Qubit[] => Unit is Ctl`
- `Op3` is of type `Qubit[] => Unit is Adj + Ctl`

then

- `cond ? Op1 | Op2` is of type `Qubit[] => Unit`
- `cond ? Op1 | Op3` is of type `Qubit[] => Unit is Adj`
- `cond ? Op2 | Op3` is of type `Qubit[] => Unit is Ctl`

For more details, see subtyping.

Comparative expressions

Equality comparisons *Equality comparisons* ($=$) and *inequality comparisons* ($\neq$) are currently limited to the following data types: `Int`, `BigInt`, `Double`, `String`, `Bool`, `Result`, `Pauli`, and `Qubit`. Equality comparisons of arrays, tuples, ranges, user-defined types, or callables are currently not supported.

Equality comparison for values of type `Qubit` evaluates whether the two expressions identify the same qubit. There is no notion of a quantum state in Q#; equality comparison, in particular, does *not* access, measure, or modify the quantum state of the qubits.

Equality comparisons for `Double` values may be misleading due to rounding effects. For instance, the following comparison evaluates to `false` due to rounding errors: `49.0 * (1.0/49.0) == 1.0`.

> [!NOTE] In the future, qsharp may support the comparisons of ranges, as well as arrays, tuples, and user-defined types provided their items support comparison. As for all types, the comparison would be by value, meaning two values are considered equal if all of their items are. For

values of user-defined type, their type also needs to match. Future support for the comparison of values of type `Range` follows the same logic; they should be equal as long as they produce the same sequence of integers, meaning the two ranges

```
let r1 = 0..2..5; // generates the sequence 0,2,4
let r2 = 0..2..4; // generates the sequence 0,2,4
```

should be considered equal.

Conversely, there is a good reason not to allow the comparison of callables, as the behavior would be ill-defined. Suppose the capability is introduced to define functions locally via a possible syntax

```
let f1 = (x → Bar(x)); // not yet supported
let f2 = Bar;
```

for some globally declared function `Bar`. The first line defines a new anonymous function that takes an argument `x`, invokes a function `Bar` with it, and then assigns it to the variable `f1`. The second line assigns the function `Bar` to `f2`. Since invoking `f1` or `f2` does the same thing, it should be possible to interchange `f1` and `f2` with each other without changing the behavior of the program. This wouldn't be the case if the equality comparison for functions was supported and `f1 == f2` evaluated to `false`. Conversely, if `f1 == f2` evaluates to `true`, then this leads to determining whether two callables have the same side effects and evaluate to the same value for all inputs, which is not possible to determine reliably. Therefore, if we want to be able to replace `f1` with `f2`, we can't allow equality comparisons for callables.

Quantitative comparison The operators *less-than* (`<`), *less-than-or-equal* (`<=`), *greater-than* (`>`), and *greater-than-or-equal* (`>=`) define quantitative comparisons. They can only be applied to data types that support such comparisons, that is, the same data types that can also support arithmetic expressions.

Logical expressions

Logical operators are expressed as keywords. Q# supports the standard logical operators *AND* (`and`), *OR* (`or`), and *NOT* (`not`). Currently, there is not an operator for a logical *XOR*. All of these operators act on operands of type `Bool`, and result in an expression of type `Bool`. As is common in most languages, the evaluation of *AND* and *OR* short-circuits, meaning if the first expression of *OR* evaluates to `true`, the second expression is not evaluated, and the same holds if the first expression of *AND* evaluates to `false`. The behavior of conditional expressions in a sense is similar, in that only ever the condition and one of the two expressions is evaluated.

Bitwise expressions

Bitwise operators are expressed as three non-letter characters. In addition to bitwise versions for *AND* (`&&&`), *OR* (`|||`), and *NOT* (`~~~`), a bitwise *XOR* (`^^^`) exists as well. They expect operands of type `Int` or `BigInt`, and for binary operators, the type of both operands has to match. The type of the entire expression equals the type of the operand(s).

Additionally, left- and right-shift operators (`<<<` and `>>>` respectively) exist, multiplying or dividing the given left-hand-side (lhs) expression by powers of two. The expression `lhs <<< 3` shifts the bit representation of `lhs` by three, meaning `lhs` is multiplied by `2^3`, provided that is still within the valid range for the data type of `lhs`. The lhs may be of type `Int` or `BigInt`. The right-hand-side expression always has to be of type `Int`. The resulting expression will be of the same type as the lhs operand.

For left- and right-shift, the shift amount (the right-hand-side operand) must be greater than or equal to zero; the behavior for negative shift amounts is undefined. If the left-hand-side operand is of type `Int`, then the shift amount additionally needs to be smaller than 64; the behavior for larger shifts is undefined.

Arithmetic expressions

Arithmetic operators are addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), negation (`-`), and exponentiation (`^`). They can be applied to operands of type `Int`, `BigInt`, or `Double`. Additionally, for integral types (`Int` and `BigInt`), an operator computing the modulus (`%`) is available.

For binary operators, the type of both operands must match, except for exponentiation; an exponent for a value of type `BigInt` must be of type `Int`. The type of the entire expression matches the type of the left operand. For exponentiation of `Int` and `BitInt`, the behavior is undefined if the exponent is negative or requires more than 32 bits to represent (that is, if it is larger than 2147483647).

Division and modulus for values of type `Int` and `BigInt` follow the following behavior for negative numbers:

`A`	`B`	`A / B`	`A % B`
5	2	2	1
5	-2	-2	1
-5	2	-2	-1
-5	-2	2	-1

That is, `a % b` always has the same sign as `a`, and `b * (a / b) + a % b` always equals `a`.

Q# does not support automatic conversions between arithmetic data types or any other data types for that matter. This is of importance especially for the `Result` data type and facilitates restricting how runtime information can propagate. It has the benefit of avoiding accidental errors, such as ones related to precision loss.

Concatenation

Concatenations are supported for values of type `String` and arrays. In both cases they are expressed via the operator `+`. For instance, `"Hello " + "world!"` evaluates to `"Hello world!"`, and `[1,2,3] + [4,5,6]` evaluates to `[1,2,3,4,5,6]`.

Concatenating two arrays requires that both arrays be of the same type, in contrast to constructing an array literal where a common base type for all array items is determined. This is because arrays are treated as invariant. The type of the entire expression matches the type of the operands.

Modifiers and combinators

Modifiers can be seen as special operators that can be applied to certain expressions only. They can be assigned an artificial precedence to capture their behavior.

For more information, see Expressions.

This artificial precedence is listed in the following table, along with how the precedence of operators and modifiers relates to how tightly item access combinators (`[,]` and `::` respectively) and call combinators (`(`, `)`) bind.

Description	Syntax	Operator	Associativity	Precedence
Call combinator	`( )`	n/a	left	900
Adjoint functor	`Adjoint`	prefix	right	950
Controlled functor	`Controlled`	prefix	right	950
Unwrap application	`!`	postfix	left	1000
Named item access	`::`	n/a	left	1100
Array item access	`[ ]`	n/a	left	1100

To illustrate the implications of the assigned precedences, suppose you have a unitary operation `DoNothing` (as defined in Specialization declarations), a callable `GetStatePrep` that returns a unitary operation, and an array `algorithms` that contains items of type `Algorithm` defined as follows

```
newtype Algorithm = (
    Register : LittleEndian,
    Initialize : Transformation,
    Apply : Transformation
);

newtype Transformation =
    LittleEndian ⇒ Unit is Adj + Ctl;
```

where `LittleEndian` is defined in Type declarations.

The following expressions, then, are all valid:

```
GetStatePrep()(arg)
(Transformation(GetStatePrep()))!(arg)
Adjoint DoNothing()
Controlled Adjoint DoNothing(cs, ())
Controlled algorithms[0]::Apply!(cs, _)
algorithms[0]::Register![i]
```

Looking at the precedences defined in the table above, you can see that the parentheses around `(Transformation(GetStatePrep()))` are necessary for the subsequent unwrap operator to be applied to the `Transformation` value rather than the returned operation. However, parentheses are not required in `GetStatePrep()(arg)`; functions are applied left-to-right, so this expression is equivalent to `(GetStatePrep())(arg)`. Functor applications also don't require parentheses around them in order to invoke the corresponding specialization, nor do array or named item access expressions. Thus, the expression `arr2D[i][j]` is perfectly valid, as is `algorithms[0]::Register![i]`.

Closures

Closures are callables that capture variables from the enclosing environment. Both function and operation closures can be created. An operation closure can be created inside a function, but it can only be applied in an operation.

Q# has two mechanisms for creating closures: lambda expressions and partial application.

Lambda expressions A lambda expression creates an anonymous function or operation. The basic syntax is a symbol tuple to bind the parameters, an arrow (→ for a function and ⇒ for an operation), and an expression to evaluate when applied.

```
// Function that captures 'x':
y → x + y

// Operation that captures 'qubit':
deg ⇒ Rx(deg * PI() / 180.0, qubit)

// Function that captures nothing:
(x, y) → x + y
```

Parameters Parameters are bound using a symbol tuple that is identical to the left-hand side of a variable declaration statement. The type of the parameter tuple is implicit. Type annotations are not supported; if type inference fails, you may need to create a top-level callable declaration and use partial application instead.

Mutable capture variables Mutable variables cannot be captured. If you only need to capture the value of a mutable variable at the instant the lambda expression is created, you can create an immutable copy:

```
// ERROR: 'variable' cannot be captured.
mutable variable = 1;
let f = () → variable;

// OK.
let value = variable;
let g = () → value;
```

Characteristics The characteristics of an anonymous operation are inferred based on the body of the lambda expression. For example:

```
// Has type Unit ⇒ Unit is Adj + Ctl because X is known to be Adj + Ctl.
() ⇒ X(q)


// Has type Unit ⇒ Result because M is neither Adj nor Ctl.
() ⇒ M(q)
```

Because of limitations in characteristics inference, this is based only on type information known at the point where the lambda expression occurs. For example:

```
let foo = op ⇒ op(q);
foo(X);
```

`foo` is inferred to have the following type based on both the body of the lambda and the type of `X`:

```
(Qubit ⇒ Unit is Adj + Ctl) ⇒ Unit
```

But the most specific type that `foo` could have is:

```
(Qubit ⇒ Unit is Adj + Ctl) ⇒ Unit is Adj + Ctl
```

If you need different characteristics for an operation lambda than what was inferred, you will need to create a top-level operation declaration instead.

Partial application Partial application is a convenient shorthand for applying some, but not all, of a callable's arguments. The syntax is the same as a call expression, but unapplied arguments are replaced with _. Conceptually, partial application is equivalent to a lambda expression that captures the applied arguments and takes in the unapplied arguments as parameters.

For example, given that `f` is a function and `o` is an operation, and the captured variable `x` is immutable:

Partial application	Lambda expression
`f(x, _)`	`a → f(x, a)`
`o(x, _)`	`a ⇒ o(x, a)`
`f(_, (1, _))`	`(a, b) → f(a, (1, b))[^1]`
`f((_, _, x), (1, _))`	`((a, b), c) → f((a, b, x), (1, c))`

Mutable capture variables Unlike lambda expressions, partial application can automatically capture a copy of the value of a mutable variable:

```
mutable variable = 1;
let f = Foo(variable, _);
```

This is equivalent to the following lambda expression:

```
mutable variable = 1;
let value = variable;
let f = x → Foo(value, x);
```

---

[^1]: The parameter tuple is strictly written `(a, (b))`, but `(b)` is equivalent to `b`.

Functor application

Functors are factories that allow you to access particular specialization implementations of a callable. Q# currently supports two functors; the `Adjoint` and the `Controlled`, both of which can be applied to operations that provide the necessary specializations.

The `Controlled` and `Adjoint` functors commute; if `ApplyUnitary` is an operation that supports both functors, then there is no difference between `Controlled Adjoint ApplyUnitary` and `Adjoint Controlled ApplyUnitary`. Both have the same type and, upon invocation, execute the implementation defined for the `controlled adjoint` specialization.

Adjoint functor If the operation `ApplyUnitary` defines a unitary transformation U of the quantum state, `Adjoint ApplyUnitary` accesses the implementation of $U\dagger$. The `Adjoint` functor is its own inverse, since $(U\dagger)\dagger = U$ by definition. For example, `Adjoint Adjoint ApplyUnitary` is the same as `ApplyUnitary`.

The expression `Adjoint ApplyUnitary` is an operation of the same type as `ApplyUnitary`; it has the same argument and return type and supports the same functors. Like any operation, it can be invoked with an argument of suitable type. The following expression applies the adjoint specialization of `ApplyUnitary` to an argument `arg`:

```
Adjoint ApplyUnitary(arg)
```

Controlled functor For an operation `ApplyUnitary` that defines a unitary transformation U of the quantum state, `Controlled ApplyUnitary` accesses the implementation that applies U conditional on all qubits in an array of control qubits being in the $|1\rangle$ state.

The expression `Controlled ApplyUnitary` is an operation with the same return type and operation characteristics as `ApplyUnitary`, meaning it supports the same functors. It takes an argument of type `(Qubit[], <TIn>)`, where `<TIn>` should be replaced with the argument type of `ApplyUnitary`, taking singleton tuple equivalence into account.

Operation	Argument Type	Controlled Argument Type
X	`Qubit`	`(Qubit[], Qubit)`
SWAP	`(Qubit, Qubit)`	`(Qubit[], (Qubit, Qubit))`
ApplyQFT	`LittleEndian`	`(Qubit[], LittleEndian)`

Concretely, if `cs` contains an array of qubits, `q1` and `q2` are two qubits, and the operation `SWAP` is as defined here, then the following expression exchanges the state of `q1` and `q2` if all qubits in `cs` are in the $|1\rangle$ state:

```
Controlled SWAP(cs, (q1, q2))
```

> [!NOTE] Conditionally applying an operation based on the control qubits being in a state other than a zero-state may be achieved by applying the appropriate adjointable transformation to the control qubits before invocation, and applying the inverses after. Conditioning the transformation on all control qubits being in the $|0\rangle$ state, for example, can be achieved by applying the `X` operation before and after. This can be conveniently expressed using a conjugation. Nonetheless,

the verbosity of such a construct may merit additional support for a more compact syntax in the future.

Item access

Q# supports item access for array items and for items in user defined types. In both cases, the access is read-only; the value cannot be changed without creating a new instance using a copy-and-update expression.

Array item access and array slicing Given an array expression and an expression of type `Int` or `Range`, a new expression may be formed using the array item access operator consisting of `[` and `]`.

If the expression inside the brackets is of type `Int`, then the new expression contains the array item at that index. For example, if `arr` is of type `Double[]` and contains five or more items, then `arr[4]` is an expression of type `Double`.

If the expression inside the brackets is of type `Range`, then the new expression contains an array of all the items indexed by the specified `Range`. If the `Range` is empty, then the resulting array is empty. For example,

```
let arr = [10, 11, 36, 49];
let ten = arr[0]; // contains the value 10
let odds = arr[1..2..4]; // contains the value [11, 49]
let reverse = arr[...-1...]; // contains the value [49, 36, 11, 10]
```

In the last line of the example, the start and end value of the range have been omitted for convenience. For more information, see Contextual expressions.

If the array expression is not a simple identifier, it must be enclosed in parentheses in order to extract an item or a slice. For instance, if `arr1` and `arr2` are both arrays of integers, an item from the concatenation would be expressed as `(arr1 + arr2)[13]`. For more information, see Precedence and associativity.

All arrays in Q# are zero-based, that is, the first element of an array `arr` is always `arr[0]`. An exception is thrown at runtime if the index or one of the indices used for slicing is outside the bounds of the array, for example, if it is less than zero or larger or equal to the length of the array.

Item access for user-defined types (For more information about how to define custom types containing one or more named or anonymous items, see Type declarations).

The contained items can be accessed via their name or by deconstruction, illustrated by the following statements that may be used as part of a operation or function implementation:

```
    let complex = Complex(1.,0.); // create a value of type Complex
    let (re, _) = complex!;       // item access via deconstruction
    let im = complex::Imaginary;  // item access via name
```

The item access operator (`::`) retrieves named items, as illustrated by the following example:

```
newtype TwoStrings = (str1: String, str2: String);

operation LinkTwoStrings(str : TwoStrings) : String {
    let s1 = str::str1;
    let s2 = str::str2:
    return s1 + s2;
}
```

While named items can be accessed by their name or via deconstruction, anonymous items can only be accessed by the latter. Since deconstruction relies on all of the contained items, the usage anonymous items is discourage when these items need to be accessed outside the compilation unit in which the type is defined.

Access via deconstruction makes use of the unwrap operator (`!`). The unwrap operator returns a tuple of all contained items, where the shape of the tuple matches the one defined in the declaration, and a single item tuple is equivalent to the item itself (see this section).

For example, for a value `nested` of type `Nested` that is defined as follows

```
newtype Nested = (Double, (ItemName : Int, String));
```

the expression `nested!` return a value of type `(Double, (Int, String))`.

The `!` operator has lower precedence than both item access operators, but higher precedence than any other operator. For a complete list of precedences, see Precedence and associativity.

Contextual and omitted expressions

Contextual expressions are expressions that are only valid in certain contexts, such as the use of item names in copy-and-update expressions without having to qualify them.

Expressions can be *omitted* when they can be inferred and automatically inserted by the compiler, for example, in the case of evaluate-and-reassign statements.

Open-ended ranges are another example that apply to both contextual and omitted expressions. They are are valid only within a certain context, and the compiler translates them into normal `Range` expressions during compilation by inferring suitable boundaries.

A value of type `Range` generates a sequence of integers, specified by a start value, a step value (optional), and an end value. For example, the `Range` literal expression `1 .. 3` generates the sequence 1,2,3. Likewise, the expression `3 .. -1 .. 1` generates the sequence 3,2,1. You can also use ranges to create a new array from an existing one by slicing, for example:

```
    let arr = [1,2,3,4];
    let slice1 = arr[1..2..4];  // contains [2,4]
    let slice2 = arr[2..-1..0]; // contains [3,2,1]
```

You cannot define an infinite range in Q#; the start and end values always need to be specified. The only exception is when you use a `Range` to slice an array. In that case, the start or end values of the range can reasonably be inferred by the compiler.

In the previous array slicing examples, it is reasonable for the compiler to assume that the intended range end should be the index of the last item in the array if the step size is positive. If the step size is negative, then the range end likely should be the index of the first item in the array, `0`. The converse holds for the start of the range.

To summarize, if you omit the range start value, the inferred start value

- is zero if no step is specified or the specified step is positive.
- is the length of the array minus one if the specified step is negative.

If you omit the range end value, the inferred end value

- is the length of the array minus one if no step is specified or the specified step is positive.
- is zero if the specified step is negative.

Q# hence allows the use of open-ended ranges within array slicing expressions, for example:

```
let arr = [1,2,3,4,5,6];
let slice1  = arr[3...];      // slice1 is [4,5,6];
let slice2  = arr[0..2...];   // slice2 is [1,3,5];
let slice3  = arr[...2];      // slice3 is [1,2,3];
let slice4  = arr[...2..3];   // slice4 is [1,3];
let slice5  = arr[...2...];   // slice5 is [1,3,5];
```

```
let slice7  = arr[4..-2...];  // slice7 is [5,3,1];
let slice8  = arr[...-1..3];  // slice8 is [6,5,4];
let slice9  = arr[...-1...];  // slice9 is [6,5,4,3,2,1];
let slice10 = arr[...];       // slice10 is [1,2,3,4,5,6];
```

Since the determination of whether the range step is positive or negative happens at runtime, the compiler inserts a suitable expression that will be evaluated at runtime. For omitted end values, the inserted expression is `step < 0 ? 0 | Length(arr)-1`, and for omitted start values it is `step < 0 ? Length(arr)-1 | 0`, where `step` is the expression given for the range step, or `1` if no step is specified.

Literals

Unit literal

The only existing literal for the `Unit` type is the value `()`.

The `Unit` value is commonly used as an argument to callables, either because no other arguments need to be passed or to delay execution. It is also used as return value when no other value needs to be returned, which is the case for unitary operations, that is, operations that support the `Adjoint` and/or the `Controlled` functor.

Int literals

Value literals for the `Int` type can be expressed in binary, octal, decimal, or hexadecimal representation. Literals expressed in binary are prefixed with `0b`, with `0o` for octal, and with `0x` for hexadecimal. There is no prefix for the commonly used decimal representation.

Representation	Value Literal
Binary	`0b101010`
Octal	`0o52`
Decimal	`42`
Hexadecimal	`0x2a`

BigInt literals

Value literals for the `BigInt` type are always postfixed with `L` and can be expressed in binary, octal, decimal, or hexadecimal representation. Literals expressed in binary are prefixed with `0b`, with `0o` for octal, and with `0x` for hexadecimal. There is no prefix for the commonly used decimal representation.

Representation	Value Literal
Binary	`0b101010L`
Octal	`0o52L`
Decimal	`42L`
Hexadecimal	`0x2aL`

Double literals

Value literals for the `Double` type can be expressed in standard or scientific notation.

Representation	Value Literal
Standard	`0.1973269804`

Representation	Value Literal
Scientific	`1.973269804e-1`

If nothing follows after the decimal point, then the digit after the decimal point may be omitted. For example, `1.` is a valid `Double` literal and the same as `1.0`. Similarly, if the digits before the decimal point are all zero, then they may be omitted. For example, `.1` is a valid `Double` literal and the same as `0.1`.

Bool literals

Existing literals for the `Bool` type are `true` and `false`.

String literals

A value literal for the `String` type is an arbitrary length sequence of Unicode characters enclosed in double quotes. Inside of a string, the back-slash character `\` may be used to escape a double quote character, and to insert a new-line as `\n`, a carriage return as `\r`, and a tab as `\t`.

The following are examples for valid string literals:

```
"This is a simple string."
"\"This is a more complex string.\", she said.\n"
```

Q# also supports *interpolated strings*. An interpolated string is a string literal that may contain any number of interpolation expressions. These expressions can be of arbitrary types. Upon construction, the expressions are evaluated and their `String` representation is inserted at the corresponding location within the defined literal. Interpolation is enabled by prepending the special character `$` directly before the initial quote, with no white space between them.

For instance, if `res` is an expression that evaluates to `1`, then the second sentence in the following `String` literal displays "The result was 1.":

```
$"This is an interpolated string. The result was {res}."
```

Qubit literals

No literals for the `Qubit` type exist, since quantum memory is managed by the runtime. Values of type `Qubit` can hence only be obtained via allocation.

Values of type `Qubit` represent an opaque identifier by which a quantum bit, or *qubit*, can be addressed. The only operator they support is equality comparison. For more information on the `Qubit` data type, See Qubits.

Result literals

Existing literals for the `Result` type are `Zero` and `One`.

Values of type `Result` represent the result of a binary quantum measurement. `Zero` indicates a projection onto the +1 eigenspace, `One` indicates a projection onto the -1 eigenspace.

Pauli literals

Existing literals for the `Pauli` type are `PauliI`, `PauliX`, `PauliY`, and `PauliZ`.

Values of type `Pauli` represent one of the four single-qubit Pauli matrices, with `PauliI` representing the identity. Values of type `Pauli` are commonly used to denote the axis for rotations and to specify with respect to which basis to measure.

Range literals

Value literals for the `Range` type are expressions of the form `start .. step .. stop`, where `start`, `step`, and `end` are expressions of type `Int`. If the step size is one, it may be omitted. For example, `start .. stop` is a valid `Range` literal and the same as `start .. 1 .. stop`.

Values of type `Range` represent a sequence of integers, where the first element in the sequence is `start`, and subsequent elements are obtained by adding `step` to the previous one, until `stop` is passed. `Range` values are inclusive at both ends, that is, the last element of the range is `stop` if the difference between `start` and `stop` is a multiple of `step`. A range may be empty if, for instance, `step` is positive and `stop < start`.

The following are examples for valid `Range` literals:

- `1 .. 3` is the range 1, 2, 3.
- `2 .. 2 .. 5` is the range 2, 4.
- `2 .. 2 .. 6` is the range 2, 4, 6.
- `6 .. -2 .. 2` is the range 6, 4, 2.
- `2 .. -2 .. 1` is the range 2.
- `2 .. 1` is the empty range.

For more information, see Contextual expressions.

Array literals

An array literal is a sequence of one or more expressions, separated by commas and enclosed in brackets `[` and `]`; for example, `[1,2,3]`. All expressions must have a common base type, which is the item type of the array.

Arrays of arbitrary length, and in particular empty arrays, may be created using a new array expression. Such an expression is of the form `new <ItemType>[expr]`, where `expr` can be any expression of type `Int` and `<ItemType>` is to be replaced by the type of the array items.

For instance, `new Int[10]` creates an array of integers containing ten items. The length of an array can be queried with the function `Length`. It is defined in the automatically opened namespace `Microsoft.Quantum.Core` and returns an `Int` value.

All items in the created array are set to the default value of the item type. Arrays containing qubits or callables must be properly initialized with non-default values before their elements may be safely used. Suitable initialization routines can be found in the `Microsoft.Quantum.Arrays` namespace.

Tuple literals

A tuple literal is a sequence of one or more expressions of any type, separated by commas and enclosed in parentheses `(` and `)`. The type of the tuple includes the information about each item type.

Value Literal	Type
`("Id", 0, 1.)`	`(String, Int, Double)`
`(PauliX,(3,1))`	`(Pauli, (Int, Int))`

Tuples containing a single item are treated as identical to the item itself, both in type and value, which is called singleton tuple equivalence.

Tuples are used to bundle values together into a single value, making it easier to pass them around. This makes it possible for every callable to take exactly one input and return exactly one output.

Literals for user-defined types

Values of a user-defined type are constructed by invoking their constructor. A default constructor is automatically generated when declaring the type. It is currently not possible to define custom constructors.

For instance, if `IntPair` has two items of type `Int`, then `IntPair(2, 3)` creates a new instance by invoking the default constructor.

Operation and function literals

Anonymous operations and functions can be created using a lambda expression.

Default values

Type	Default
`Unit`	`()`
`Int`	`0`
`BigInt`	`0L`
`Double`	`0.0`
`Bool`	`false`
`String`	`""`
`Qubit`	*invalid qubit*
`Result`	`Zero`
`Pauli`	`PauliI`
`Range`	empty range
Array	empty array
Tuple	all items are set to default values
User-defined type	all items are set to default values
Operation	*invalid operation*
Function	*invalid function*

For qubits and callables, the default is an invalid reference that cannot be used without causing a runtime error.

Type System

With the focus for quantum algorithm being more towards what should be achieved rather than on a problem representation in terms of data structures, taking a more functional perspective on language design is a natural choice. At the same time, the type system is a powerful mechanism that can be leveraged for program analysis and other compile-time checks that facilitate formulating robust code.

All in all, the Q# type system is fairly minimalist, in the sense that there isn't an explicit notion of classes or interfaces as one might be used to from classical languages like C# or Java. We also take a somewhat pragmatic approach making incremental progress, such that certain construct are not yet fully integrated into the type system. An example are functors, which can be used within expressions but don't yet have a representation in the type system. Correspondingly, they cannot currently be assigned or passed as arguments, similar as it is the case for type parametrized callables. We expect to make incremental progress in extending the type system to be more complete, and balance immediate needs with longer-term plans.

Available Types

All types in Q# are immutable.

Type	Description
`Unit`	Represents a singleton type whose only value is `()`.
`Int`	Represents a 64-bit signed integer. Values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
`BigInt`	Represents signed integer values of any size.
`Double`	Represents a double-precision 64-bit floating-point number. Values range from -1.79769313486232e308 to 1.79769313486232e308 as well as NaN (not a number).
`Bool`	Represents Boolean values. Possible values are `true` or `false`.
`String`	Represents text as values that consist of a sequence of UTF-16 code units.
`Qubit`	Represents an opaque identifier by which virtual quantum memory can be addressed. Values of type `Qubit` are instantiated via allocation.
`Result`	Represents the result of a projective measurement onto the eigenspaces of a quantum operator with eigenvalues ± 1. Possible values are `Zero` or `One`.
`Pauli`	Represents a single-qubit Pauli matrix. Possible values are `PauliI`, `PauliX`, `PauliY`, or `PauliZ`.
`Range`	Represents an ordered sequence of equally spaced `Int` values. Values may represent sequences in ascending or descending order.
Array	Represents values that each contain a sequence of values of the same type.
Tuple	Represents values that each contain a fixed number of items of different types. Tuples containing a single element are equivalent to the element they contain.
User defined type	Represents a user defined type consisting of named and anonymous items of different types. Values are instantiated by invoking the constructor.
Operation	Represents a non-deterministic callable that takes one (possibly tuple-valued) input argument returns one (possibly tuple-valued) output. Calls to operation values may have side effects and the output may vary for each call even when invoked with the same argument.
Function	Represents a deterministic callable that takes one (possibly tuple-valued) input argument returns one (possibly tuple-valued) output. Calls to function values do not have side effects and the output is will always be the same given the same input.

Operations and functions

As elaborated in more detail in the description of the qubit data type, quantum computations are executed in the form of side effects of operations that are natively supported on the targeted quantum processor. These are, in fact, the only side effects in Q#. Since all types are immutable, there are no side effects that impact a value that is explicitly represented in Q#. Hence, as long as an implementation of a certain callable does not directly or indirectly call any of these natively implemented operations, its execution always produces the same output, given the same input.

Q# allows you to explicitly split out such purely deterministic computations into *functions*. Since the set of natively supported instructions is not fixed and built into the language itself, but rather fully configurable and expressed as a Q# library, determinism is guaranteed by requiring that functions can only call other functions and cannot call any operations. Additionally, native instructions that are not deterministic, that is, because they impact the quantum state, are represented as operations. With these two restrictions, functions can be evaluated as soon as their input value is known, and, in principle, never need to be evaluated more than once for the same input.

Q# therefore distinguishes between two types of callables: operations and functions. All callables take a single argument (potentially tuple-valued) as input and produce a single value (tuple) as output. Syntactically, the operation type is expressed as `<TIn> ⇒ <TOut> is <Char>`, where `<TIn>` is to be replaced by the argument type, `<TOut>` is to be replaced by the return type, and `<Char>` is to be replaced by the operation characteristics. If no characteristics need to be specified, the syntax simplifies to `<TIn> ⇒ <TOut>`. Similarly, function types are expressed as `<TIn> → <TOut>`.

Aside from this determinism guarantee, there is little difference between operations and functions. Both are first-class values that can be passed around freely; they can be used as return values or arguments to other callables, as shown in the following example:

```
function Pow<'T>(op : 'T ⇒ Unit, pow : Int) : 'T ⇒ Unit {
    return PowImpl(op, pow, _);
}
```

Both can be instantiated based on a type-parametrized definition, for example, the type parametrized function `Pow` above, and they can be partially applied as done in the `return` statement in the example.

Operation characteristics

In addition to the information about input and output type, the operation type contains information about the characteristics of an operation. This information, for example, describes what functors are supported by the operation. Additionally, the internal representation also contains optimization-relevant information that is inferred by the compiler.

The characteristics of an operation are a set of predefined and built-in labels. They are expressed in the form of a special expression that is part of the type signature. The expression consists either of one of the predefined sets of labels, or of a combination of characteristics expressions via a supported binary operator.

There are two predefined sets, `Adj` and `Ctl`.

- `Adj` is the set that contains a single label indicating that an operation is adjointable, meaning it supports the `Adjoint` functor and the applied quantum transformation can be ”undone”, that is, it can be inverted.
- `Ctl` is the set that contains a single label indicating that an operation is controllable, meaning it supports the `Controlled` functor and its execution can be conditioned on the state of other qubits.

The two operators that are supported as part of characteristics expressions are the set union `+` and the set intersection `*`. In EBNF,

```
predefined = "Adj" | "Ctl";
characteristics = predefined
    | "(", characteristics, ")"
    | characteristics ("+"|"*") characteristics;
```

As one would expect, `*` has higher precedence than `+` and both are left-associative. The type of a unitary operation, for example, is expressed as `<TIn> ⇒ <TOut> is Adj + Ctl`, where `<TIn>` should be replaced with the type of the operation argument, and `<TOut>` replaced with the type of the returned value.

> [!NOTE] Indicating the characteristics of an operation in this form has two major advantages; for one, new labels can be introduced without having exponentially many language keywords for all combinations of labels. Perhaps more importantly, using expressions to indicate the characteristics of an operation also supports parameterizations over operation characteristics in the future.

Quantum-specific data types

This topic describes the `Qubit` type, along with two other types that are somewhat specific to the quantum domain: `Pauli` and `Result`.

Qubit

Q# treats qubits as opaque items that can be passed to both functions and operations, but can only be interacted with by passing them to instructions that are native to the targeted quantum processor. Such instructions are always defined in the form of operations, since their intent is to modify the quantum state. The restriction that functions cannot modify the quantum state, despite the fact that qubits can be passed as

input arguments, is enforced by the requiring that functions can only call other functions, and cannot call operations.

The Q# libraries are compiled against a standard set of intrinsic operations, meaning operations which have no definition for their implementation within the language. Upon targeting, the implementations that express them in terms of the instructions that are native to the execution target are linked in by the compiler. A Q# program thus combines these operations as defined by a target machine to create new, higher-level operations to express quantum computation. In this way, Q# makes it very easy to express the logic underlying quantum and hybrid quantum-classical algorithms, while also being very general with respect to the structure of a target machine and its realization of quantum state.

Within Q# itself, there is no type or construct in Q# that represents the quantum state. Instead, a qubit represents the smallest addressable physical unit in a quantum computer. As such, a qubit is a long-lived item, so Q# has no need for linear types. Hence, we do not explicitly refer to the state within Q#, but rather describe how the state is transformed by the program, for example, via the application of operations such as `X` and `H`. Similar to how a graphics shader program accumulates a description of transformations to each vertex, a quantum program in Q# accumulates transformations to quantum states, represented as entirely opaque references to the internal structure of a target machine.

A Q# program has no ability to introspect into the state of a qubit, and thus is entirely agnostic about what a quantum state is or on how it is realized. Rather, a program can call operations such as `Measure` to learn information about the quantum state of the computation.

Pauli

Values of type `Pauli` specify a single-qubit Pauli operator; the possibilities are `PauliI`, `PauliX`, `PauliY`, and `PauliZ`. `Pauli` values are used primarily to specify the basis for a quantum measurement.

Result

The `Result` type specifies the result of a quantum measurement. Q# mirrors most quantum hardware by providing measurements in products of single-qubit Pauli operators; a `Result` of `Zero` indicates that the +1 eigenvalue was measured, and a `Result` of `One` indicates that the -1 eigenvalue was measured. That is, Q# represents eigenvalues by the power to which -1 is raised. This convention is more common in the quantum algorithms community, as it maps more closely to classical bits.

Immutability

All types in Q# are *value types*. Q# does not have a concept of a reference or pointer. Instead, it allows you to reassign a new value to a previously declared variable via a `set` statement. For example, there is no distinction in behavior between reassignments for variables of type `Int` or variables of type `Int[]`. Consider the following sequence of statements:

```
mutable arr1 = new Int[3];
let arr2 = arr1;
set arr1 w/= 0 <- 3;
```

The first statement instantiates a new array of integers `[0,0,0]` and assigns it to `arr1`. The next statement assigns that value to a variable with name `arr2`. The last statement creates a new array instance based on `arr1` with the same values except for the value at index 0 which is set to 3. The newly created array is then assigned to the variable `arr1`. The last line makes use of the abbreviated syntax for evaluate-and-reassign statements, and could equivalently have been written as `set arr1 = arr1 w/ 0 ← 1;`. After running the three statements, `arr1` will contain the value `[3,0,0]` while `arr2` remains unchanged and contains the value `[0,0,0]`.

Q# clearly thus distinguishes the mutability of a handle and the behavior of a type. Mutability within Q# is a concept that applies to a *symbol* rather than a type or value; it applies to the handle that allows you to

access a value rather than to the value itself. It is *not* represented in the type system, implicitly or explicitly.

Of course, this is merely a description of the formally defined behavior; under the hood, the actual implementation uses a reference counting scheme to avoid copying memory as much as possible. The modification is specifically done in place as long as there is only one currently valid handle that accesses a certain value.

Singleton tuple equivalence

To avoid any ambiguity between tuples and parentheses that group sub-expressions, a tuple with a single element is considered to be equivalent to the contained item, including its type. For example, the types `Int`, `(Int)`, and `((Int))` are treated as being identical. The same holds true for the values `5`, `(5)` and `(((5)))`, or for `(5, (6))` and `(5, 6)`. This equivalence applies for all purposes, including assignment. Since there is no dynamic dispatch or reflection in Q# and all types in Q# are resolvable at compile-time, singleton tuple equivalence can be readily implemented during compilation.

Subtyping and variance

Q# supports only a few conversion mechanisms. Implicit conversions can happen only when applying binary operators, evaluating conditional expressions, or constructing an array literal. In these cases, a common supertype is determined and the necessary conversions are performed automatically. Aside from such implicit conversions, explicit conversions via function calls are possible and often necessary.

Currently, the only subtyping relation that exists applies to operations. Intuitively, it makes sense that one should be allowed to substitute an operation that supports more than the required set of functors. Concretely, for any two concrete types `TIn` and `TOut`, the subtyping relation is

```
(TIn => TOut) :>
(TIn => TOut is Adj), (TIn => TOut is Ctl) :>
(TIn => TOut is Adj + Ctl)
```

where `A :> B` indicates that `B` is a subtype of `A`. Phrased differently, `B` is more restrictive than `A` such that a value of type `B` can be used wherever a value of type `A` is required. If a callable relies on an argument (item) of being of type `A`, then an argument of type `B` can safely be substituted since if provides all the necessary capabilities.

This kind of polymorphism extends to tuples in that a tuple of type `B` is a subtype of a tuple type `A` if it contains the same number of items and the type of each item is a subtype of the corresponding item type in `A`. This is known as *depth subtyping*. There is currently no support for *width subtyping*, that is, there is no subtype relation between any two user-defined types or a user-defined type and any built-in type. The existence of the `unwrap` operator, which allows you to extract a tuple containing all named and anonymous items, prevents this.

> [!NOTE] In regards to callables, if a callable processes an argument of type `A`, then it is also capable of processing an argument of type `B`. If a callable is passed as an argument to another callable, then it has to be capable of processing anything that the type signature may require. This means that if the callable needs to be able to process an argument of type `B`, any callable that is capable of processing a more general argument of type `A` can be passed safely. Conversely, we expect that if we require that the passed callable returns a value of type `A`, then the promise to return a value of type `B` is sufficient, since that value will provide all necessary capabilities.

The operation or function type is *contravariant* in its argument type and *covariant* in its return type. `A :> B` hence implies that for any concrete type `T1`,

```
(B → T1) :> (A → T1), and
(T1 → A) :> (T1 → B)
```

where → here can mean either a function or operation, and we omit any annotations for characteristics. Substituting `A` with `(B → T2)` and `(T2 → A)` respectively, and substituting `B` with `(A → T2)` and `(T2 → B)` respectively, leads to the conclusion that, for any concrete type `T2`,

```
((A → T2) → T1) :> ((B → T2) → T1), and
((T2 → B) → T1) :> ((T2 → A) → T1), and
(T1 → (B → T2)) :> (T1 → (A → T2)), and
(T1 → (T2 → A)) :> (T1 → (T2 → B))
```

By induction, it follows that every additional indirection reverses the variance of the argument type, and leaves the variance of the return type unchanged.

> [!NOTE] This also makes it clear what the variance behavior of arrays needs to be; retrieving items via an item access operator corresponds to invoking a function of type `(Int → TItem)`, where `TItem` is the type of the elements in the array. Since this function is implicitly passed when passing an array, it follows that arrays need to be covariant in their item type. The same considerations also hold for tuples, which are immutable and thus covariant with respect to each item type. If arrays weren't immutable, the existence of a construct that would allow you to set items in an array, and thus take an argument of type `TItem`, would imply that arrays also need to be contravariant. The only option for data types that support getting and setting items is hence to be *invariant*, meaning there is no subtyping relation whatsoever; `B[]` is *not* a subtype of `A[]` even if `B` is a subtype of `A`. Despite the fact that arrays in Q# are immutable, they are invariant rather than covariant. This means, for example, that a value of type `(Qubit ⇒ Unit is Adj)[]` cannot be passed to a callable that requires an argument of type `(Qubit ⇒ Unit)[]`. Keeping arrays invariant allows for more flexibility related to how arrays are handled and optimized in the runtime, but it may be possible to revise that in the future.

Type parameterizations

Q# supports type-parameterized operations and functions. The Q# standard libraries make heavy use of type-parametrized callables to provide a host of useful abstractions, including functions like `Mapped` and `Fold` that are familiar from functional languages.

To motivate the concept of type parameterizations, consider the example of the function `Mapped`, which applies a given function to each value in an array and returns a new array with the computed values. This functionality can be perfectly described without specifying the item types of the input and output arrays. Since the exact types do not change the implementation of the function `Mapped`, it makes sense that it should be possible to define this implementation for arbitrary item types; we want to define a *factory* or *template* that, given the concrete types for the items in the input and output array, returns the corresponding function implementation. This notion is formalized in the form of type parameters.

Concretization

Any operation or function declaration may specify one or more type parameters that can be used as the types, or part of the types, of the callable's input or output, or both. The exceptions are entry points, which must be concrete and cannot be type-parametrized. Type parameter names start with a tick (') and may appear multiple times in the input and output types. All arguments that correspond to the same type parameter in the callable signature must be of the same type.

A type-parametrized callable needs to be concretized, that is, it must be provided with the necessary type arguments before it can be assigned or passed as argument, such that all type parameters can be replaced with concrete types. A type is considered to be concrete if it is one of the built-in types, a user-defined type, or if it is concrete within the current scope. The following example illustrates what it means for a type to be concrete within the current scope, and is explained in more detail below:

```
function Mapped<'T1, 'T2> (
    mapper : 'T1 → 'T2,
    array : 'T1[]
) : 'T2[] {

    mutable mapped = new 'T2[Length(array)];
```

```
        for (i in IndexRange(array)) {
            set mapped w/= i <- mapper(array[i]);
        }
        return mapped;
    }

    function AllCControlled<'T3> (
        ops : ('T3 ⇒ Unit)[]
    ) : ((Bool,'T3) ⇒ Unit)[] {

        return Mapped(CControlled<'T3>, ops);
    }
```

The function `CControlled` is defined in the `Microsoft.Quantum.Canon` namespace. It takes an operation `op` of type `'TIn ⇒ Unit` as argument and returns a new operation of type `(Bool, 'TIn) ⇒ Unit` that applies the original operation, provided a classical bit (of type `Bool`) is set to *true*; this is often referred to as the classically controlled version of `op`.

The function `Mapped` takes an array of an arbitrary item type `'T1` as argument, applies the given `mapper` function to each item, and returns a new array of type `'T2[]` containing the mapped items. It is defined in the `Microsoft.Quantum.Array` namespace. For the purpose of the example, the type parameters are numbered to avoid making the discussion more confusing by giving the type parameters in both functions the same name. This is not necessary; type parameters for different callables may have the same name, and the chosen name is only visible and relevant within the definition of that callable.

The function `AllCControlled` takes an array of operations and returns a new array containing the classically controlled versions of these operations. The call of `Mapped` resolves its type parameter `'T1` to `'T3 ⇒ Unit`, and its type parameter `'T2` to `(Bool,'T3) ⇒ Unit`. The resolving type arguments are inferred by the compiler based on the type of the given argument. We say that they are *implicitly* defined by the argument of the call expression. Type arguments can also be specified explicitly, as is done for `CControlled` in the same line. The explicit concretization `CControlled<'T3>` is necessary when the type arguments cannot be inferred.

The type `'T3` is concrete within the context of `AllCControlled`, since it is known for each *invocation* of `AllCControlled`. That means that as soon as the entry point of the program - which cannot be type-parametrized - is known, so is the concrete type `'T3` for each call to `AllCControlled`, such that a suitable implementation for that particular type resolution can be generated. Once the entry point to a program is known, all usages of type parameters can be eliminated at compile-time. We refer to this process as *monomorphization*.

Some restrictions are needed to ensure that this can indeed be done at compile-time as opposed to only at run time.

Restrictions

Consider the following example:

```
    operation Foo<'TArg> (
        op : 'TArg ⇒ Unit,
        arg : 'TArg
    ) : Unit {

        let cbit = RandomInt(2) == 0;
        Foo(CControlled(op), (cbit, arg));
    }
```

Ignoring that an invocation of `Foo` will result in an infinite loop, it serves for the purpose of illustration. `Foo` invokes itself with the classically controlled version of the original operation `op` that has been passed in, as

well as a tuple containing a random classical bit in addition to the original argument.

For each iteration in the recursion, the type parameter `'TArg` of the next call is resolved to `(Bool, 'TArg)`, where `'TArg` is the type parameter of the current call. Concretely, suppose `Foo` is invoked with the operation `H` and an argument `arg` of type `Qubit`. `Foo` will then invoke itself with a type argument `(Bool, Qubit)`, which will then invoke `Foo` with a type argument `(Bool, (Bool, Qubit))`, and so on. Clearly, in this case `Foo` cannot be monomorphized at compile-time.

Additional restrictions apply to cycles in the call graph that involve only type-parametrized callables. Each callable needs to be invoked with the same set of type arguments after traversing the cycle.

> [!NOTE] It would be possible to be less restrictive and require that for each callable in the cycle, there is a finite number of cycles after which it is invoked with the original set of type arguments, such as the case for the following function:
>
> ```
> function Bar<'T1,'T2,'T3>(a1:'T1, a2:'T2, a3:'T3) : Unit{
>     Bar<'T2,'T3,'T1>(a2, a3, a1);
> }
> ```
>
> For simplicity, the more restrictive requirement is enforced. Note that for cycles that involve at least one concrete callable without any type parameter, such a callable will ensure that the type-parametrized callables within that cycle are always called with a fixed set of type arguments.

Type inference

Q#'s type inference algorithm is based on inference algorithms designed for the Hindley-Milner type system. While top-level callables must be declared with explicit type annotations, most types used within a callable can be inferred. For example, given these callables:

```
function Length<'a>(xs : 'a[]) : Int
function Mapped<'a, 'b>(f : 'a → 'b, xs : 'a[]) : 'b[]
```

and this expression:

```
Mapped(Length, [[], ["a"], ["b", "c"]])
```

then the type argument to `Length` is inferred to be `Length<String[]>`, and the type arguments to `Mapped` are inferred to be `Mapped<String[], Int>`. It is not required to write these types explicitly.

Ambiguous types

Sometimes there is not one single principal type that can be inferred for a type variable. In these cases, type inference fails with an error referring to an ambiguous type. For example, change the previous example slightly:

```
Mapped(Length, [[]])
```

What is the type of `[[]]`? In some type systems, it's possible to give it the type `∀a. a[][]`, but this is not supported in Q#. A concrete type is required, but there are an infinite number of types that work: `String[][]`, `(Int, Int)[][]`, `Double[][][]`, and so on. You must explicitly say which type you meant.

There are multiple ways to do this, depending on the situation. For example:

1. Call `Length` with a type argument.

   ```
   Mapped(Length<String>, [[]])
   ```

2. Call `Mapped` with its first type argument. (The _ for its second type argument means that it should still be inferred.)

```
Mapped<String[], _>(Length, [[]])
```

3. Replace the empty array literal with an explicitly-typed call to a library function.

```
Mapped<String[], _>(Length, [EmptyArray<String>()])
```

Grammar

A reference implementation of the Q# grammar is available in the ANTLR4 format. The grammar source files are listed below:

- **QSharpLexer.g4** describes the lexical structure of Q#.
- **QSharpParser.g4** describes the syntax of Q#.

Target language

The ANTLR grammar contains some embedded C# code. It looks like this (enclosed in curly braces):

```
BraceRight : '}' { if (ModeStack.Count > 0) PopMode(); };
```

If you want to generate a parser for a target language other than C#, you will need to change these code snippets.