Q#基本文法

OpenQL勉強会
量子プログラミング言語Q#で量子アルゴリズムを学ぶ3（ハンズオン）

@tanaka_733

最近のQ#

Q# 0.9（まで）リリース
- https://docs.microsoft.com/en-us/quantum/relnotes/?view=qsharp-preview
- 0.6で名前空間の変更など大幅な変更あり
- .NET Core 3対応

Azure Quantum発表
- https://news.microsoft.com/ja-jp/2019/11/08/191108-announcing-microsoft-azure-quantum/
- https://azure.microsoft.com/ja-jp/services/quantum/

Quantum Development Kit OSS化
- コンパイラー
  - https://github.com/microsoft/qsharp-compiler/
- Runtime
  - https://github.com/microsoft/qsharp-runtime

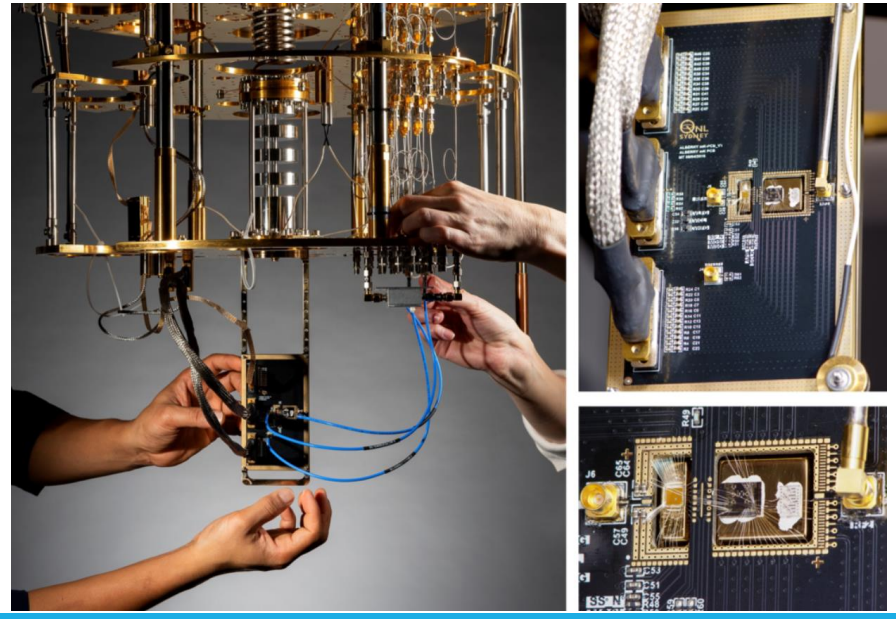
Azure Quantum

進行中のプロジェクト

- 量子コンピューターでも解読不能な公開鍵暗号アルゴリズムとプロトコル
- オープンソースの Quantum Development Kit
- 現時点では、3 線式、極低温 CMOS 設計、1 平方センチメートルのチップをおよそ絶対零度で稼働させるだけで、最大 50,000 キュービットを制御可能

成果

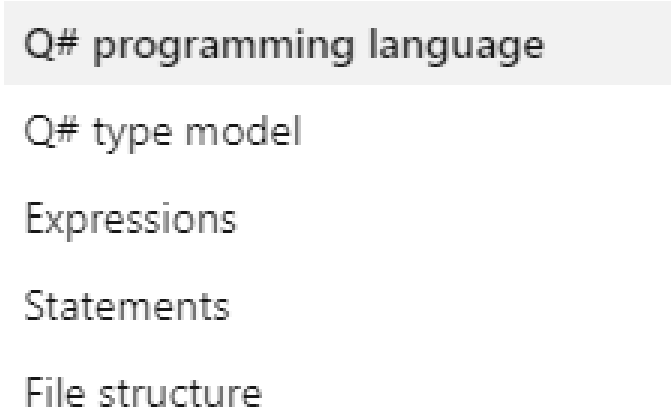
- Case Western Reserve University によるMRIスキャンにかかる時間を1/3に（量子シミュレーター）
- OTI Lumionics によるODELの発光材料であるAlq3のシミュレーション（現状は大規模古典コンピューターすら不要な規模だが、より規模を大きくすることを検討）
- 1QBit と IonQによるAzure Quantumのデモ



Q#基礎文法

ドキュメント

- Q# Programming language
  - https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview
- References > Q# Libraries
  - https://docs.microsoft.com/en-us/qsharp/api/qsharp/microsoft.quantum.convert?view=qsharp-preview



おさらい Q#プロジェクトの構造

.NET Coreプロジェクトにライブラリを追加
- Q#コンパイラも.NET Coreのライブラリとして動作
- Q#コードはC#コードにトランスパイルされる

エントリはC#コード
- いわゆるMainメソッド
- 古典的コンピューターでの処理に対応

Q#コードはC#からクラスとして参照する
- 1Q# operationが1クラスとして参照できる
- 実行先（実機へのドライバー、シミュレーター）を指定する

量子ビットの割り当て

using, borrowing で割り当てる
usingは新規量子ビット割り当てで状態が0で割り当てられ、0にして戻す
borrowingは一時的な割り当てで、不定な状態で割り当てられ、同じ状態にして戻す

```
operation QubitExample1(bits: Int): Unit {
    using (q = Qubit()) {
        // ...
    }
    using ((ancilla, qubits) = (Qubit(), Qubit[bits * 2 + 3])) {
        // ancilla はQubit
        // qubitsはQubit[]
    }

    borrowing (q = Qubit()) {
        // ...
        //Measureなど測定はNG
    }
    borrowing ((ancilla, qubits) = (Qubit(), Qubit[bits * 2 + 3])) {
        // ...
    }
}
```

シグネチャとタプル、変数

メソッドの引数は、変数名: 型名
返り値は任意の型で複数値戻す場合はタプル、void相当はUnit。

不変な変数はletで宣言
可変な変数はmutableで宣言し、setで更新

```
operation MeasureTwice(q1: Qubit, p1: Pauli, q2: Qubit, p2: Pauli): (Result, Result) {
    return (Measure([p1], [q1]), Measure([p2], [q2]));
}

operation VariableExample(): Unit {
    let (i, f) = (5, 0.1); // i is bound to 5 and f to 0.1
    mutable (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 rebound to (5,6), y is rebound to [8]
    using((q1, q2) = (Qubit(), Qubit())) {
        let (r1, r2) = MeasureTwice(q1, PauliX, q2, PauliY);
    }
}
```

ユーザー定義型とタプル

複数の型を組み合わせてユーザー定義型をタプルとして定義できる

変数名を定義でき::で参照できる

ユーザー定義型を含めたタプルは!演算子でアンラップ(unwrap)しばらばらの変数に割り当てられる

```
newtype Complex = (Re : Double, Im : Double);

function Addition (c1 : Complex, c2 : Complex) : Complex {
    return Complex(c1::Re + c2::Re, c1::Im + c2::Im);
}

function PrintMsg (value : Complex) : Unit {
    let (re, im) = value!; //unwrap
    Message ($"Re:{re}, Im:{im}");
}
```

Range（範囲）型リテラル

数値の範囲をRange型で表現でき、..を使ってリテラル表現できる

開始..ステップ..終了で、ステップ省略時は1もしくは-1、昇順、降順ともに可能

```
operation Ranges(): Unit {
    Message("Range: 1..3");
    for (i in 1..3) {
        Message($"{i}");
    }
    Message("Range: 2..2..5");
    for (i in 2..2..5) {
        Message($"{i}");
    }
    Message("Range: 2..2..6");
    for (i in 2..2..6) {
        Message($"{i}");
    }
    Message("Range: 6..-2..2");
    for (i in 6..-2..2) {
        Message($"{i}");
    }
    Message("Range: 2..1");
    for (i in 2..1) {
        Message($"{i}");
    }
    Message("Range: 2..6..7");
    for (i in 2..6..7) {
        Message($"{i}");
    }
    Message("Range: 2..2..1");
    for (i in 2..2..1) {
        Message($"{i}");
    }
}
```

数値型とリテラル

Int: 64bit符号付整数

BigInt: 任意サイズの符号付整数。リテラルでは末尾にLをつける。C#のBigInteger相当。

Double: 倍精度浮動小数。リテラルでは末尾にDをつける。指数表記可能(4e-7)

リテラル表現では、0bで2進表記、0xで16進表記が可能

暗黙的な型変換がないため、異なる型同士の四則演算は不可

```
function NumericExpressions() : Unit {
    let zero = 0;
    let hex = 0xdeadbeaf;
    let bigZero = 0L;
    let bigHex = 0x123456789abcdef123456789abcdefL;
    let bigOne = bigZero + 1L;
    //let bigSum = zero + bigHex; //compile error
    let bigSum = IntAsBigInt(zero) + bigHex;

    let module = 4 % 3;
    let power = 4 ^ 3;
    Message($"4^3 = {power}");
    Message($"4.2^3.2 = {powerDouble}");
    let bitwiseAnd = 0b101 &&& 0b111;
    Message($"0b101 AND 0b111 = {bitwiseAnd}");
    let bitwiseOr = 0b101 ||| 0b111;
    Message($"0b101 OR 0b111 = {bitwiseOr}");
    let bitwiseXor = 0b101 ^^^ 0b111;
    Message($"0b101 XOR 0b111 = {bitwiseXor}");

    let leftShift = 1 <<< 3;
    Message($"1 left shift 3 = {leftShift}");
    let rightShift = 0b1000 >>> 3;
    Message($"0b1000 right shift 3 = {rightShift}");
}
```

Immutable variable zero
Type: Int

部分適用

関数を部分適用できる。

部分適用後に引数として指定される変数は_で指定する

```
function AddBuilder(num: Int): (Int -> Int){
    return Add(_, num);
}

function Add(num1: Int, num2: Int): Int {
    return num1 + num2;
}

function AddBuilderExample(): Unit {
    let op = AddBuilder(4);
    let res1 = op(3);

    //変数に格納せず実行
    let re2 = (AddBuilder(5))(4);
}
```

```
operation Op<'T1>(value1: 'T1, qubit: Qubit, value2: 'T1): Unit is Adj {

}

operation OpExample(): Unit {
    using(qb = Qubit()) {
        let f1 = Op<Int>(_, qb, _); // f1 has type ((Int,Int) => Unit is Ad
        let f2 = Op(5, qb, _);      // f2 has type (Int => Unit is Adj)
        //let f3 = Op(_,qb, _);       // f3 generates a compilation error
    }
}
```

Forループ

For文は配列もしくはRangeを列挙する。
ループ変数はletやimmutable宣言不要。(ループごとにletで宣言されるのと同等)
列挙対象がタプルの場合、分解して受けることが可能

```
//制御構文
operation ForLoopExample(): Unit {
    using(qubits = Qubit[5]) {
        for (qb in qubits) {
            H(qb);
        }

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

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

Repeat Until Success

「出るまで回せば当たる」構文

確率的な操作に対して、
都合のよい結果が得られるまでループする

```
//https://arxiv.org/abs/1311.1074
operation RepeatUntilSuccessExample(target: Qubit): Unit {
    using (anc = Qubit()) {
        repeat {
            H(anc);
            T(anc);
            CNOT(target,anc);
            H(anc);
            Adjoint T(anc);
            H(anc);
            T(anc);
            H(anc);
            CNOT(target,anc);
            T(anc);
            Z(target);
            H(anc);
            let result = M(anc);
        }
        until (result == Zero)
        fixup {
            //do nothing in this case
        }
    }
}
```

within~apply

前処理、処理、前処理のAdjoint といったパターンの処理を行う構文

下の2つが等価になる

```
operation ApplyWith<'T>(
    outerOperation : ('T => Unit is Adj),
    innerOperation : ('T => Unit),
    target : 'T)
: Unit {

    outerOperation(target);
    innerOperation(target);
    Adjoint outerOperation(target);
}
```

```
operation ApplyWithEx<'T>(
    outerOperation : ('T => Unit is Adj),
    innerOperation : ('T => Unit),
    target : 'T)
: Unit {

    within{
        outerOperation(target);
    }
    apply {
        innerOperation(target);
    }
}
```

クラスライブラリ

以下の名前空間に属する関数は古典的プログラミングでも使われるようなクラスライブラリ

Microsoft.Quantum.Bitwise ビット演算

Microsoft.Quantum.Convert 型変換。Q#には暗黙的型変換がないので重要。

Microsoft.Quantum.Math 算術関数

Microsoft.Quantum.Arrays 配列操作

Microsoft.Quantum.Logical ブール演算

配列操作

Q#では多次元配列ではなく配列の配列のみ。

配列の要素を書き換えて新しい配列を生成するために
copy and update (w/= *index* <- *値*) 操作がある

配列から部分配列を得るのにRange型を適用する

```
function JaggedArrayExample(): Unit {
    let N = 4;
    mutable multiplicationTable = new Int[][N];
    for (i in 1..N) {

        mutable row = new Int[i];
        for (j in 1..i) {
            set row w/= j-1 <- i * j;
        }
        set multiplicationTable w/= i-1 <- row;
    }
}
```

C#でいうLINQに相当するデータ配列操作関数が用意されている

サンプルのArrays.qs参照

次回予定

測定その2

- 通常の測定（Joint Measurementではない）のさらなる応用
- Joint Measurementのさらなるパターン