# QLang

**The Qubit Language**

# Team

**Christopher Campbell  (System Architect)**
**Sankalpa Khadka (Language Guru)**
**Winnie Narang (Verification and Validation)**
**Jonathan Wong (Manager)**
**Clément Canonne (LaTex)**

# Introduction

**Quantum Computing**

- **Computing using principle of Quantum Mechanics.**
- **Simple analogies with Classical Computing.**
  - **Bits – 101 -> Qubits (vectors) - |101>**
  - **Gates – AND, OR, etc. -> Unitary Matrices – H , X, Y, Z**

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

# Motivations

Design language to perform quantum computation and simulate quantum algorithm through

- Simple and intuitive syntax
- Leverage well-known and elegant Dirac notation for qubit representation.

  <01101| (bra) or |1010> (ket)
- Significantly reduces the complexity of dealing with matrices and their associated operation such as tensor product.

  |0> @ |1>
- Provide comprehensive set of operators for quantum computation.

# Result: QLang

```
def apply(mat x) : mat result {
    mat y;
    y = |0>;
    result = y*x;
}

def compute() : mat final_result{
    mat x;
    x = [(1,1)(1,-1)];
    final_result = apply(x);
}
```

# Brief Tutorial

**Types**

- int (integers): 17, 0, -3489
- float (floating point): 24.2, -3., 17.006
- comp (complex): C(7.4 + 8.1I)
- mat (matrix): [(1,2,3)(4,5,6)] (gates) , |1101> (qubits)

**Operators (All arithmetic operations + Matrix Operations)**

- multiplication , H * X, H * |001>, <010|*|010>
- Tensor Product, H @ X, |001> @ |10>
- norm, norm(|010>)
- transpose,  trans(H)
- adjoint, adj(Z)
- conjugate, conj(C(4.+5.7I))

# Brief Tutorial

## Control-Flow/Loops

- **If-else**

  ```
  if (norm(A) eq 1){  output = 5;  }
  ```

- **While loop**

  ```
  while (i < 5){ print(i);    i= i+1;}
  ```

- **For Loop**

  ```
  for (i from 0 to 10 by 2){ print(i); }
  ```

# Brief Tutorial

## Built-In Variables and Functions

**Variables**

- H – Hadamard gate
- X – Pauli X
- Y – Pauli Y
- IDT – Identity Matrix (2x2)
- e, pi – the numbers e and pi

**Functions**

- print(val) – prints val (takes any type)
- printq(qubit) – prints a matrix in Dirac notation if possible
- rows(matrix) – returns number of rows in a matrix
- cols(matrix) – returns number of columns in a matrix
- elem(matrix, row, col) – returns the element given by [row,col]

# Brief Tutorial

Function name parameter Return type Return variable

```
def apply(mat x) : mat result {
    mat y;
    y = |0>;
    result = y*x;
}
```

Function name

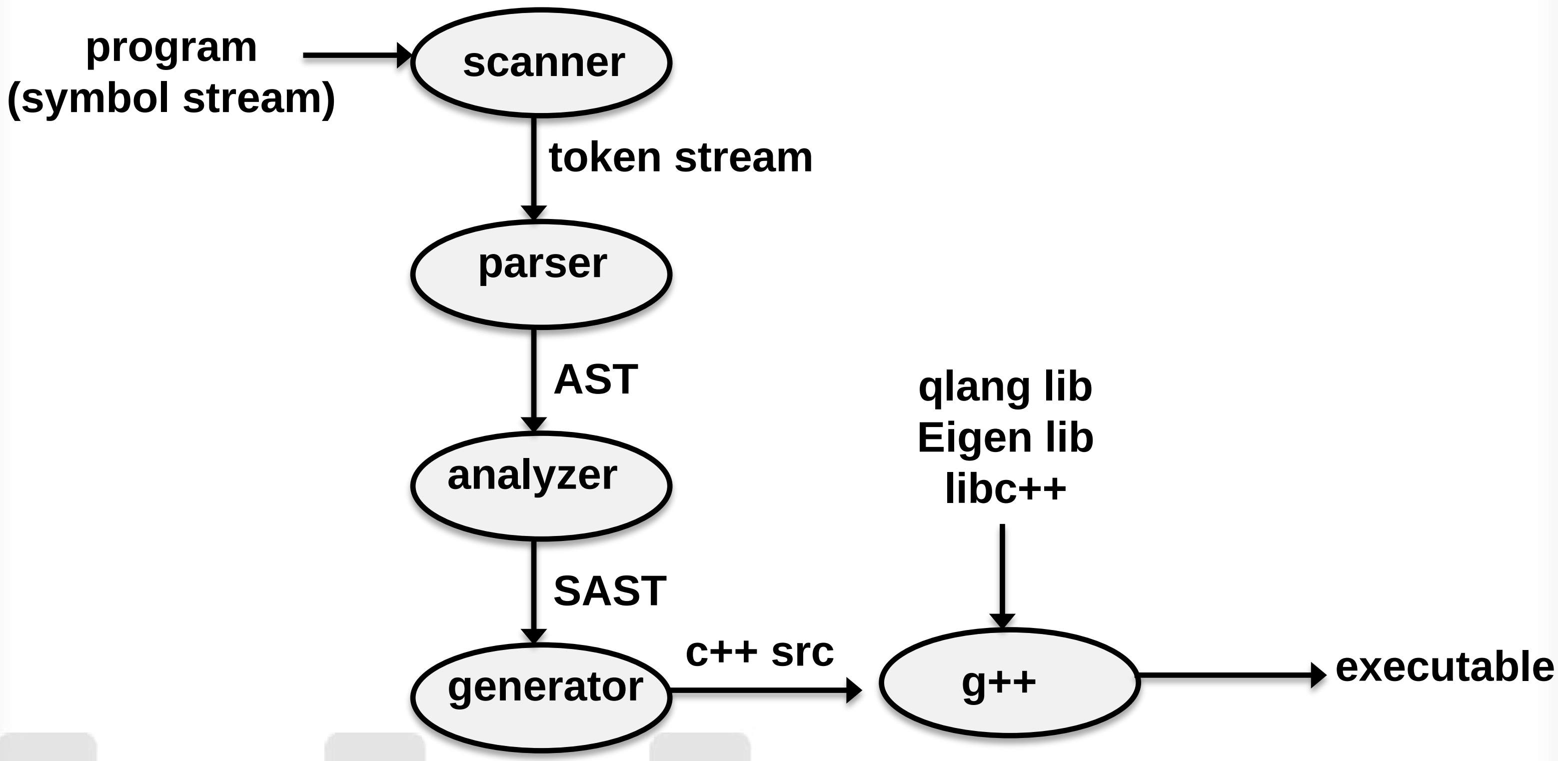
Main Execution function

Output variable which prints

```
def compute() : mat final_result{
    mat x;
    x = [(1,1)(1,-1)];
    final_result = apply(x);
}
```

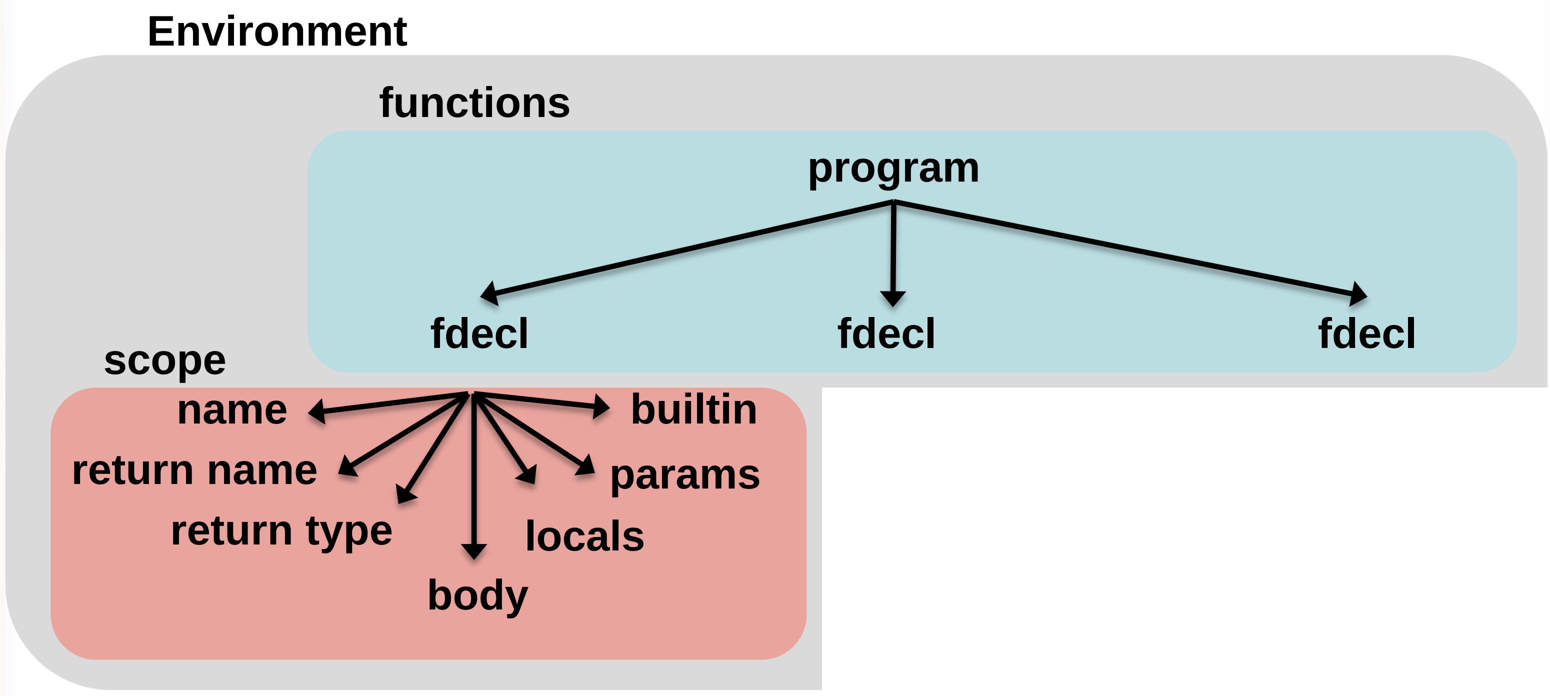
# Implementation

## Design



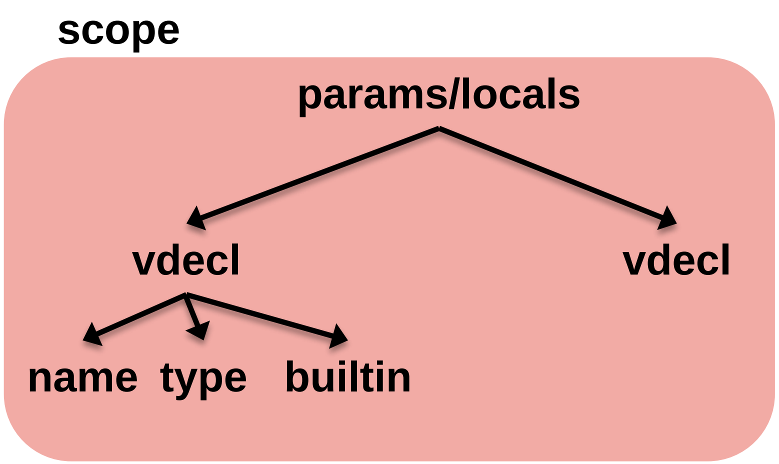
# Implementation

## Structure



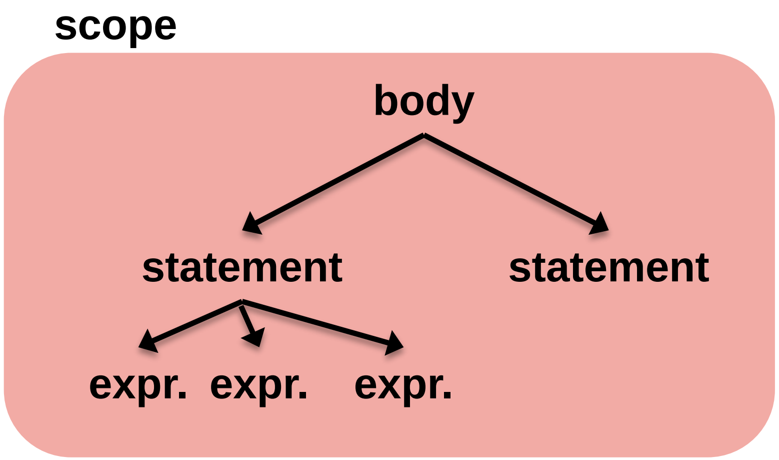
# Implementation

## Structure



# Implementation

## Structure



# Implementation

## Details

function name

formal params

return type

return name

```
def x2(int a) : comp  result {
    result = a * 2;
}
```

automatically returned

```
def compute() : comp  final_result {
    int a;
    a = 3;
    final_result = x2(a);
}
```

automatically printed

```
#include <iostream>
#include <complex>
#include <cmath>
#include <Eigen/Dense>
#include <qlang>
using namespace Eigen;
using namespace std;

MatrixXcf test_add (MatrixXcf x )
{
     MatrixXcf y;
     MatrixXcf result;

     y = genQubit("01",1);
     result = x + y;
    return result;
}

int main ()
{
    MatrixXcf x;
    MatrixXcf final_result;
    x = genQubit("10",1);
    final_result = test_add(x);
    std::cout << final_result << endl;
    return 0;
}
```

# Implementation

## Analyzer Exceptions

```
let binop_error t = match t with
    Ast.Add -> raise (Except("Invalid use of binop: 'expr + expr'"))
  | Ast.Sub -> raise (Except("Invalid use of binop: 'expr - expr'"))
  | Ast.Mult -> raise (Except("Invalid use of binop: expr * expr'"))
  | Ast.Div -> raise (Except("Invalid use of binop: 'expr / expr'"))
  | Ast.Mod -> raise (Except("Invalid use of binop: 'expr % expr'"))
  | Ast.Expn -> raise (Except("Invalid use of binop: 'expr ^ expr'"))
  | Ast.Or -> raise (Except("Invalid use of binop: 'expr or expr'"))
  | Ast.And -> raise (Except("Invalid use of binop: 'expr and expr'"))
  | Ast.Xor -> raise (Except("Invalid use of binop: 'expr xor expr'"))
  | Ast.Tens -> raise (Except("Invalid use of binop: 'expr @ expr'"))
  | Ast.Eq -> raise (Except("Invalid use of binop: 'expr eq expr'"))
  | Ast.Neq -> raise (Except("Invalid use of binop: 'expr neq expr'"))
  | Ast.Lt -> raise (Except("Invalid use of binop: 'expr lt expr'"))
  | Ast.Gt -> raise (Except("Invalid use of binop: 'expr gt expr'"))
  | Ast.Leq -> raise (Except("Invalid use of binop: 'expr leq expr'"))
  | Ast.Geq -> raise (Except("Invalid use of binop: 'expr geq expr'"))
```

# Testing and Verification

- Semantic testing
  - Check for incorrect syntax or logical errors.
- Code generation testing
  - For syntactically correct code, generate equivalent C++ code.

- Test phases
  - Unit testing
  - Integration testing
  - System testing

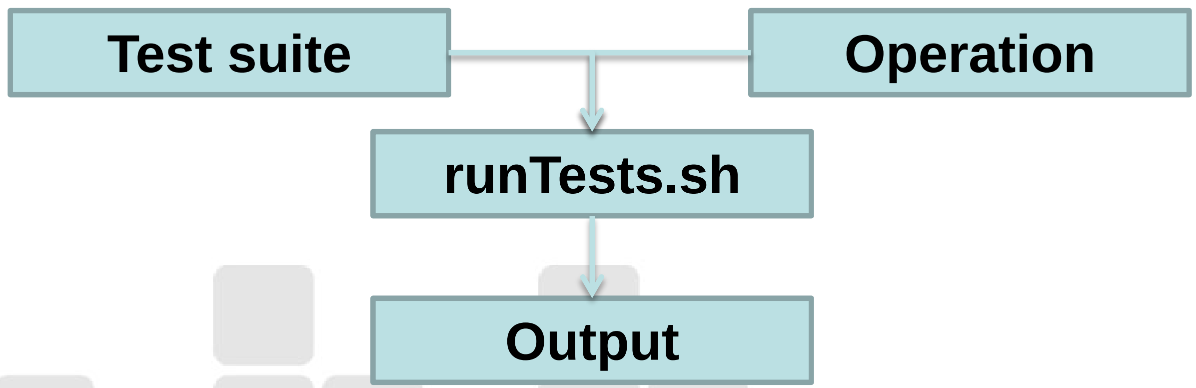
# Testing and Verification

## Test Suites

- SemanticSuccess
- SemanticFailures

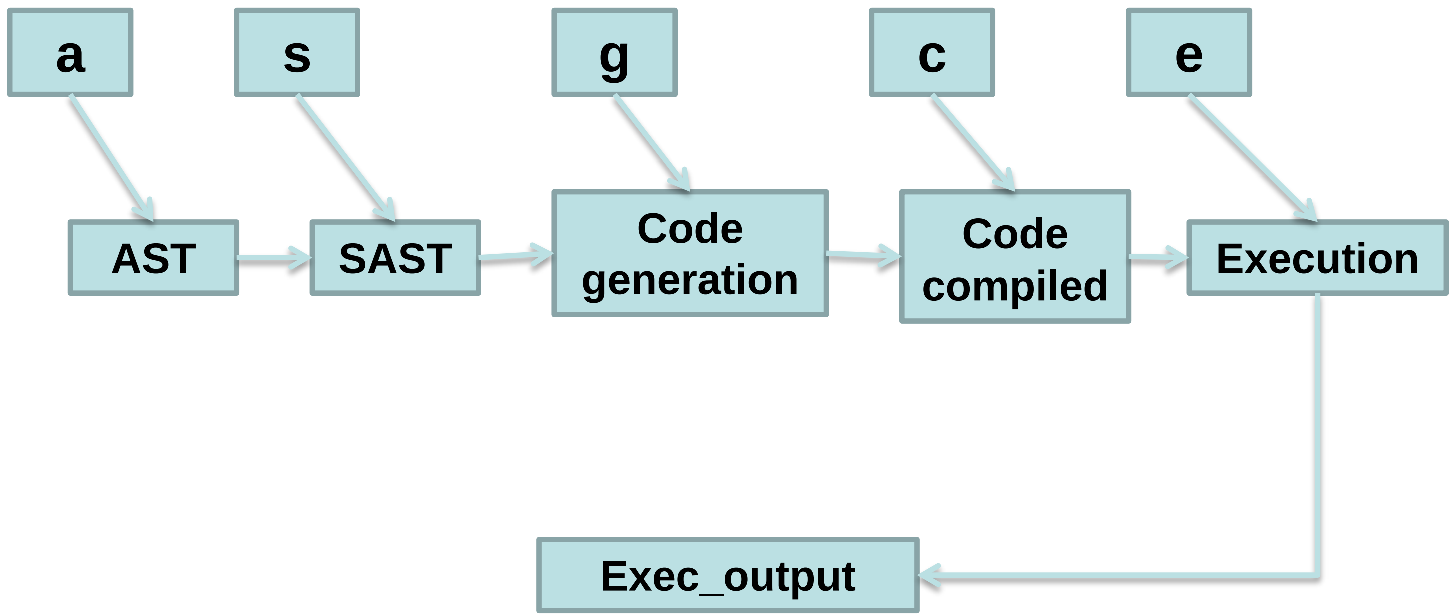
## Automation

- One universal script to do it all



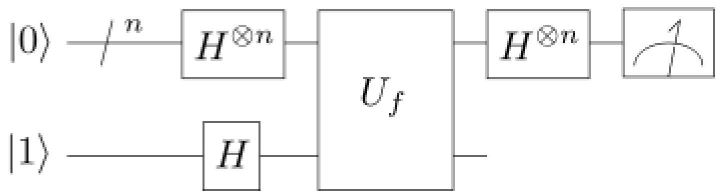
# Testing and Verification

## Workflow



# Demo

## Deutsh Algorithm




**10.1.3 Problem 3**

Consider the circuit and show the probabilities of outcome 0 where $|\Psi_{in}\rangle = |1\rangle$

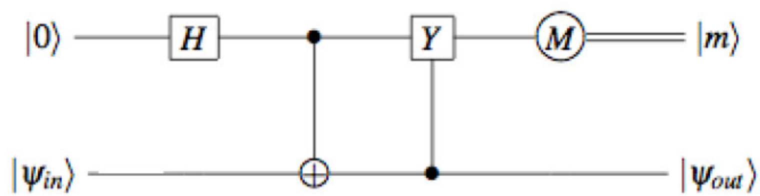



Figure 2: Quantum Circuit