Quark
The Team

- Daria Jung
- Jamis Johnson
- Linxi Fan (Jim)
- Parthiban Loganathan
Why Quark?

Quantum computing has the potential to become a reality in the next few decades. We're thinking ahead of the curve and have developed a language that makes it easy to build quantum circuits, which consist of quantum gates and quantum registers holding qubits.
Quantum Computing will allow us to:

- Factorize large integers in polynomial time (Shor's algorithm)
- Search unsorted database in sublinear time (Grover's Search)
- Build the Infinite Improbability Drive and solve intergalactic travel
What is Quark?

"QUantum Analysis and Realization Kit"

A high-level language for quantum computing that encapsulates mathematical operations and quantum computing specific components like quantum registers.

A futuristic compiler on your laptop.
Features

- Easy-to-use, high-level language influenced by MATLAB and Python
- Useful built-in data types for fractions and complex numbers
- Support for matrices and matrix operations
- Quantum registers and ability to query them
- Built-in quantum gate functions
- Imports
- Informative semantic error messages
- Cross-platform
How did we do it?

Compiler flow:

- Preprocessor
- Scanner
- Parser
- AST
- Semantic Checker
- SAST
- Code Generator
- OS-aware g++ invocation
- Quantum Simulator (Quark++)
Preprocessor

• Resolves import statements before the scanner and parser stages
• Recursively finds all imports and prepends them to the file
• Handles cyclic and repetitive imports
Scanner
Based on MicroC

All the usual tokens + specific ones for

- fractions: \( \frac{1}{2} \)
- complex numbers: \( i(3, 4) \)
  - \( i \) can still be used as a variable, not a function
- matrix operations: \( \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \), \( A', A ** B \)
- quantum registers and querying: \( q \text{reg}, \langle 10, 1 \rangle, q \text{ ? [1:5], q ?'} 3 \)
Parser

Grammar was developed incrementally

- Quantum registers query
- Matrix and high dimensional array literals
- Membership
- Fractions, complex numbers
- Pythonic for-loops

Pacman Parsing
Some example rules

```plaintext
expr:
  ...
  /* Query */
  | expr ? expr
  | expr ? [ : expr ]
  | expr ? [expr : expr ]
  ...
  /* Membership testing with keyword 'in' */
  | expr in expr
  ...
  /* literals */
  | expr $ expr
  | [ | matrix_row_list | ]
  | i( expr , expr )
  | < expr , expr >

iterator:
  | ident in [range]
  | datatype ident in [range]
  | datatype ident in expr
```
Lexical and syntactical analysis complete
Now we need semantic checks

Valid syntax doesn't always make sense

The importance of semantic checks in real life

“I’m sorry” and “my bad” mean the same thing...

Unless you are at a funeral.
Semantic Checker

- StrMap hashtables
  - Variable table
    ```
    type var_info = {
      v_type: A.datatype;
      v_depth: int; (* how deep in scope *)
    }
    ```

- Function table
  ```
  type func_info = {
    f_args: A.datatype list;
    f_return: A.datatype;
    f_defined: bool; (* for forward declaration *)
  }
  ```
Semantic Checker

- Environment struct

```plaintext
type environment = {
    var_table: var_info StrMap.t;
    func_table: func_info StrMap.t;
    (* current function name waiting for 'return' *)
    (* if "", we are not inside any function *)
    func_current: string;
    depth: int;
    is_returned: bool;
    in_loop: bool; (* check break/continue validity *)
}
```
## Semantic Checker

- From AST to SAST

```plaintext
<table>
<thead>
<tr>
<th>expr =</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binop of expr * binop * expr</td>
</tr>
<tr>
<td>AssignOp of lvalue * binop * expr</td>
</tr>
<tr>
<td>QueryOp of expr * queryop * expr * expr</td>
</tr>
<tr>
<td>Unop of unop * expr</td>
</tr>
<tr>
<td>PostOp of lvalue * postop</td>
</tr>
<tr>
<td>Assign of lvalue * expr</td>
</tr>
<tr>
<td>IntLit of string</td>
</tr>
<tr>
<td>BoolLit of string</td>
</tr>
<tr>
<td>FractionLit of expr * expr</td>
</tr>
<tr>
<td>QRegLit of expr * expr</td>
</tr>
<tr>
<td>FloatLit of string</td>
</tr>
<tr>
<td>StringLit of string</td>
</tr>
<tr>
<td>ArrayLit of expr list</td>
</tr>
<tr>
<td>ArrayCtor of datatype * expr</td>
</tr>
<tr>
<td>MatrixLit of expr list list</td>
</tr>
<tr>
<td>MatrixCtor of datatype * expr * expr</td>
</tr>
</tbody>
</table>

and expr =

| Binop of expr * A.binop * expr * op_tag |
| QueryOp of expr * A.queryop * expr * expr * expr |
| Unop of A.unop * expr * op_tag |
| PostOp of lvalue * A.postop |
| Assign of lvalue * expr |
| IntLit of string |
| BoolLit of string |
| FloatLit of string |
| StringLit of string |
| FractionLit of expr * expr |
| QRegLit of expr * expr |
| ComplexLit of expr * expr |
| ArrayLit of A.datatype * expr list |
| ArrayCtor of A.datatype * expr (* int size of new a) |
| MatrixLit of A.datatype * expr list int (* column) |
| MatrixCtor of A.datatype * expr * expr (* int, int) |
```
Semantic Checker

- Traverse AST recursively to produce SAST

```plaintext
A.ComplexLit(real_ex, im_ex) ->
  let env, s_real_ex, real_type = gen_s_expr env real_ex in
  let env, s_im_ex, im_type = gen_s_expr env im_ex in
  match real_type, im_type with
  | A.DataType(T.Int), A.DataType(T.Int) ->
  | A.DataType(T.Int), A.DataType(T.Float) ->
  | A.DataType(T.Float), A.DataType(T.Int) ->
  | A.DataType(T.Float), A.DataType(T.Float) ->
    env, S.ComplexLit(s_real_ex, s_im_ex), A.DataType(T.Complex)
  | _ -> failwith @@ compound_type_err_msg "complex" real_type im_type
)```
Semantic Checker

- Tag the SAST with op_tag constants to facilitate code generation

```plaintext
(* tag what operator is actually used in C++ *)
type op_tag =
  | OpVerbatim  (* no change to the operator *)
  | CastComplex1 (* cast the first arg to complex *)
  | CastComplex2 (* cast the second arg to complex *)
  | CastFraction1 (* cast the first arg to fraction *)
  | CastFraction2 (* cast the second arg to fraction *)
  | OpFloatComparison (* equality/inequality with tolerance *)
  | OpArrayConcat
  | OpStringConcat
  | OpMatrixKronecker
  | OpMatrixTranspose
  | OpQuerySingleBit (* measure only a single bit, not a range *)
```
Semantic Checker

- Tag the SAST with op_tag constants to facilitate code generation

```ml
let binop_math op type1 type2 =
let notmod = op <> A.Mod in
let notmodpow = notmod && op <> A.Pow in
match type1, type2 with
| T.Float, T.Int
| T.Int, T.Float
| T.Float, T.Float when notmod ->
  T.Float, S.OpVerbatim
| T.Int, T.Int ->
  T.Int, S.OpVerbatim
| T.Float, T.Complex
| T.Int, T.Complex when notmod ->
  T.Complex, S.CastComplex1
| T.Complex, T.Float
| T.Complex, T.Int when notmod ->
  T.Complex, S.CastComplex2
| T.Complex, T.Complex when notmod ->
  T.Complex, S.OpVerbatim
| T.Int, T.Fraction when notmodpow ->
  T.Fraction, S.CastFraction1
| T.Fraction, T.Int when notmodpow ->
  T.Fraction, S.CastFraction2
```
Semantic Checker

- Separate source file for built-in functions (e.g. quantum gates)
- Can be overridden by users
- `print()` and `print_noline()` support any number of args of any type

```c
"phase_scale" -> [qreg; f; i], void
"phase_shift" -> [qreg; f; i], void
(* multi-bit gates *)
"generic_1gate" -> [qreg; cx_mat; i], void
"generic_2gate" -> [qreg; cx_mat; i; i], void
"generic_ngate" -> [qreg; cx_mat; AArrayType(i)], void
(* control gates *)
"cnot" -> [qreg; i; i], void
"toffoli" -> [qreg; i; i; i], void
"control_phase_shift" -> [qreg; f; i; i], void
"ncnot" -> [qreg; AArrayType(i); i], void
```
Semantic Checker

Error messages

- "A function is confused with a variable: u"
- "Function foo() is forward declared, but called without definition"
- "If statement predicate must be bool, but fraction provided"
- "Array style for-loop must operate on array type, not complex[]"
- "Matrix element unsupported: string"
- "Incompatible operands for **: string -.- fraction"
- "All rows in a matrix must have the same length"
Code Generation
Code Generation

- Recursively walks the SAST to generate a string of valid C++ program
- The generated string, concatenated with a header string, should compile with the simulator and Eigen library

```cpp
let header_code = "#include "qureg.h"\n" ^
"#include "qumat.h"\n" ^
"#include "qugate.h"\n" ^
"#include "quarklang.h"\n\n" ^
"using namespace Qumat;\n" ^
"using namespace Qugate;\n\n"
```

- No exception should be thrown at this stage
Code Generation

Type Mapping

- int → C++ int64_t
- float → C++ primitive float
- string → C++ std::string
- complex → C++ std::complex<float>
- arrays → C++ std::vector<> 
- matrices → Eigen::Matrix<float, Dynamic, Dynamic>
- fraction → Quark++ Frac class
- qreg → Quark++ Qureg class
Code Generation

Op Tag

| S.OpVerbatim -> |
| if op = A.Pow then (* special: not infix! *) |
| two_arg "pow" expr1_code expr2_code |

| else |
| parenthesize expr1_code op expr2_code |

| S.CastComplex1 -> |
| parenthesize (cast_complex expr1_code) op expr2_code |

| S.CastComplex2 -> |
| parenthesize expr1_code op (cast_complex expr2_code) |

| S.CastFraction1 -> |
| parenthesize (cast_fraction expr1_code) op expr2_code |

| S.CastFraction2 -> |
| parenthesize expr1_code op (cast_fraction expr2_code) |

| S.OpArrayConcat -> |
| two_arg "concat_vector" expr1_code expr2_code |

| S.OpStringConcat -> |
| parenthesize expr1_code A.Add expr2_code |

| S.OpMatrixKronecker -> |
| two_arg "kronecker_mat" expr1_code expr2_code |

| S.OpFloatComparison -> |
| let equal_func = if op = A.Eq then |
| "equal_tolerance" else "unequal_tolerance" |
Code Generation

Pythonic for-loop

- `[len(a) : 0 : step(x)]` the step size can be negative
- Whether `step(x)` is negative or not can only be determined at runtime
- We use system generated temp variables to handle this.
  Always prefixed with "_QUARK_" and followed by a string of 10 random chars.
**Code Generation**

**Pythonic for-loop**

```python
def int step:
    return 2 - 4;

def int main:
    for int i in [10 : 0: step()]:
        print(i);
    return 0;
```

```cpp
#include "qureg.h"
#include "qumat.h"
#include "qugate.h"
#include "quarklang.h"

using namespace Qumat;
using namespace Qugate;

int64_t step()
{
    return (2 - 4);
} // end step()

int main()
{
    int64_t _QUARK_5H0aq5mw6x = 0;
    int64_t _QUARK_v3YH001B0h = step();
    int64_t _QUARK_l03AMaXh6u = _QUARK_v3YH001B0h > 0 ? 1 : -1;
    for (int64_t i = 10;
         _QUARK_l03AMaXh6u * i < _QUARK_l03AMaXh6u * 0;
         i += _QUARK_v3YH001B0h){
        std::cout << std::boolalpha
                  << std::setprecision(6) << i << std::endl;
    } // end for
    return 0;
} // end main()
```
Code Generation

More examples

(4 if i(9) != i(9, 0) else 3) + (3$7 if "sh" == "sh" else 8$19);

(Frac((unequal_tolerance(std::complex<float>>(9, 0.0), std::complex<float>>(9, 0)) ? 4 : 3), 1) + ((std::string("sh") == std::string("sh")) ? Frac(3, 7) : Frac(8, 19)));
Code Generation

More examples

```
complex[][[] marray = [
    || i(2), i(-1); i(PI/2), i(0, -E); i(.2), i(.5) ||],
    || i(0), i(PI**2); i(.1), i(0); i(3), i(.5) ||
];
marray[0] + complex[][ 2, 3 ||];
```

```
vector<Matrix<
std::complex<float>, Dynamic, Dynamic>> marray = vector<
Matrix<
std::complex<float>, Dynamic, Dynamic>>{ matrix_literal(2, vector<std::complex<float>>{ std::complex<float>(2, 0.0), std::complex<float>(-1, 0.0), std::com
plex<float>((3.141592653589793 / 2), 0.0), std::complex<float>(0, (-2.718281828459045)), std::complex<float>(.2, 0.0), std::complex<float>(.5, 0.0) })}, matrix_literal(2, vector<std::complex<float>>{ std::complex<float>(0, 0.0), std::complex<float>(pow(3.141592653589793, 2), 0.0), std::complex<float>((.1, 0.0), std::complex<float>(0, 0.0), std::complex<float>(3, 0.0), std::complex<float>(.5, 0.0) }) })
};

(marray[0] + Matrix<std::complex<float>, Dynamic, Dynamic>::Zero(2, 3));
```
Simulator: Quark++
Simulator: Quark++

- Written over the summer. Built from scratch except for the Eigen matrix library.
- Features optimized C++11 code for quantum register manipulation and quantum gates/operations.
- Can be used as a standalone library for any quantum computing education or research project.
- Minor modification to accommodate the Quark language.
User Interface

- Command line args
  - `-s`: source
  - `-c`: generated.cpp
  - `-o`: executable
  - `-sc`, `-sco`
  - `-static`

- Precompiled dynamic/static libraries
- Minimal user effort to install dependencies
- OS aware. Supports all major OSes
Let's look at some code

A simple Hello World

def int main:
    { print("Hello, Ground!");
    return 0;
}

It was unfortunately a very short hello for our whale friend
Defining types

```c
int i = 4;
float f = 2.0;
bool b = true;
string s = "So Long, and Thanks for All the Fish";
string[] arr = ["Ford", "Prefect", "Zaphod", "Beeblebrox"];  
int[][] arr2 = [[1,2,3],[4,5,6]];

fraction f = 84/2;
complex c = i(5.0, 7.0);
float[][] = [[1.0, 2.1; 3.2, 46.1]];

qreg q = <\ 42, 0 |>;
```
Special operations

% FRACTIONS
frac foo = 2/3;
~foo; % 3/2
int i = 5;
i > foo; % true

% COMPLEX NUMBERS
complex cnum = i(3.0, 1);
real(cnum); % 3.0
imag(cnum); % 1
complex cnum2 = i(9) % this gives us i(9, 0)

% MATRICES
float[,] mat = [| 1.2, 3.4; 5.6, 7.8 |];
mat[2, 1];
mat'; % transpose matrix

% QUANTUM REGISTERS
greg q = <|10, 3|>
hadamard(q);
q ? [2:10]; % measures qubit 2 to 10
Control flow

```python
if x > 0:
    print("positive");
elif x < 0:
    print("negative");
else:
    print("zero");

while x > 42: {
    print(x);
    x = x - 1;
}

arr = [1, 2, 3];
for i in arr:
    print i;

i;
for i in [1:10]
for i in [1:10:2]
```
Imports

```python
import ../lib/mylib1;
import ../lib/herlib2;
import imported_file;

def int main:
    { return imported_file.function(5); }
```

So Fancy!
def int gcd: int x, int y
{
    while y != 0:
    {
        int r = x mod y;
        x = y;
        y = r;
    }
    return x;
}

def int main:
{
    % prints the greatest common divisor of 10 and 20
    print(gcd(10, 20));
    return 0;
}
Quantum Computing Demo

Time

FROODY

Hang on to your Towel!

Let's see Shor's algorithm and Grover's Search in action! Real quantum computing programs running on a not-so-real quantum computer (our simulator)
What did we learn?
Start early!!!
OCaml:

[oh-kam-uh l]

Mostly harmless
Interacting with other homo sapiens

- Group projects are painful (more so than Vogon poetry)
- Allocating work strategically avoids bottlenecks in pipeline
- Better communication saves time and headaches
- Dictatorship > Democracy when it comes to software

Vogon Poetry

Vogon Poetry is so awful that even the Sarkopsi of Burphon XII, whose religion strictly forbids the taking of one's life, consider suicide a preferable alternative to a Vogon poetry reading.