Quark

*Quantum Analysis and Realization Kit Language*

<table>
<thead>
<tr>
<th>Name</th>
<th>UNI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daria Jung</td>
<td>djj2115</td>
</tr>
<tr>
<td>Jamis Johnson</td>
<td>jmj2180</td>
</tr>
<tr>
<td>Linxi Fan (Jim)</td>
<td>lf2422</td>
</tr>
<tr>
<td>Parthiban Loganathan</td>
<td>pl2487</td>
</tr>
</tbody>
</table>

December 15, 2014
Slides

Find our slides at the following URL: http://slides.com/quarklang/quark
Contents

**Quark**
**Slides**
**Contents**
**Introduction**
  - History
  - Language

**Tutorial**
  - Environment Installation
  - Compiling and Running Quark Programs
  - Essential Syntax
  - Control Flow
  - Declaration
  - Matrices
  - Quantum Registers

**Language Reference Manual**
  - Lexical Conventions
  - Comments
  - Whitespace
  - Tokens
  - Identifiers
  - Keywords
    - Reserved Prefix
    - Pseudo-reserved
    - Mathematical Constants
  - Punctuation
  - Escape Sequences
**Data Types**
  - int
  - float
  - fraction
  - Note on int, float, and fraction
  - bool
  - complex
  - string
  - qreg
  - matrix
  - array
  - void

**Function types**
**Declarations**
  - Declaring a Variable
  - Declaring an Array
  - Declaring a Matrix
Operators
Arithmetic
Concatenation
Assignment
Logical
Bitwise Logical / Unary
Quantum
Ternary Operator
Operator Precedence and Associativity

Statements
Blocks
Return Statement
If else if else Statement
While Loop
For Statement
Array Iterator
Range Iterator

Break and Continue
Functions
  Function Declaration

Imports
Casting
Pass by value pass by reference
Overloading
Scoping
Built-in Functions
  General
  Fraction
  Complex
  Math
  Matrix
  Matrix Generation
  Quantum Registers
  Quantum Gates (Functions apply a specific gate to a quantum register)
    Single bit gates
    Multi bit gates

  Control Gates
  Other Gates

Grammar

Project Plan
  Project Timeline:
  Project Log:
  Roles and Responsibilities

Architecture
  Global Overview
  Compiler Architecture
  Quark++ Simulator
Test Plan

Rationale
Implementation
Representative Program
Tests Used

Lessons Learned

Parthiban Loganathan
Jim Fan
Daria Jung
Jamis Johnson

Appendix

A.1 scanner.mll
A.2 parser.mly
A.3 type.ml
A.4 ast.ml
A.5 semantic.ml
A.6 sast.ml
A.7 generator.ml
A.8 preprocessor.ml
A.9 builtin.ml
A.10 compiler.ml
A.11 testall.sh
A.12 addition.qk
A.13 array.qk
A.14 complex.qk
A.15 elif.qk
A.16 float.qk
A.17 fraction.qk
A.18 gcd.qk
A.19 hello_world.qk
A.20 import.qk
A.21 logic.qk
A.22 matrix.qk
A.23 multi-array.qk
A.24 range.qk
A.25 while.qk
A.26 grover.qk
A.27 shor.qk
A.28 qreg.qk
Introduction

History

In the early 1980’s, Richard Feynman observed that certain quantum mechanical effects could not be efficiently simulated using classical computation methods. This led to the proposal for the idea of a “quantum computer”, a computer that uses the effects of quantum mechanics, such as superposition and entanglement, to its advantage.

Classical computers require data to be encoded in binary bits, where each unit is always in a definite state of either 0 or 1. Quantum computation uses qubits, a special unit that can be 0 and 1 at the same time, i.e. a superposition of base states. Measuring a qubit will force it to collapse to either 0 or 1, with a probability distribution determined by its amplitude.

Qubits effectively operate on exponentially large number of entangled states simultaneously, though all of them will collapse as soon as we make a measurement. With carefully designed quantum algorithms, we are able to speed up certain classical problems dramatically by tapping into such massive computational resources. It is not unlike parallel computing, but powered by quantum mechanical laws.

Though quantum computing is still in its infancy, the last two decades have witnessed two ingenious algorithms that produced much inspiration and motivation for quantum computing research. One is Shor’s algorithm (1994) for integer factorization, which yields exponential speedup over the best classical alternative, and the other is Grover’s search algorithm (1996), which provides quadratic speedup for unsorted database search. Once realized, the former would have significant impact on cryptography, while the latter would have great implication on NP-hard problems.

Language

Quark is a domain-specific imperative programming language to allow for expression of quantum algorithms. The purpose of QUARK is to ease the burden of writing quantum computing algorithms and describing quantum circuits in a user-friendly way. In theory, our language can produce quantum circuit instructions that are able to run on actual quantum computers in the future.

Most quantum algorithms can be decomposed into a quantum circuit part and a classical pre/post-processing part. Recognizing this, QUARK is designed to integrate classical and quantum data types and controls in a seamless workflow. Built in types like complex numbers, fractions, matrices and quantum registers combined with a robust built-in gate library make QUARK a great starting point for quantum computing researchers and enthusiasts.

A relatively efficient quantum circuit simulator is included as part of the QUARK architecture. Source code written in QUARK is compiled to C++, which can then be passed onto our quantum simulator.
Tutorial

Environment Installation

Install Vagrant, a tool for provisioning virtual machines used to maintain a consistent environment. You will also need to install VirtualBox which Vagrant uses for virtualization. After unzipping Quark.tar.gz, navigate to the main Quark directory (the one which contains bootstrap.sh and Vagrantfile) and run vagrant up. This will provision and run a Ubuntu 14.04 LTS virtual instance as well as download and install dependencies such as OCaml and g++-4.8. Run vagrant ssh to ssh into the VM. Make sure you are in the /vagrant directory by running the command pwd and if you are not run cd /vagrant. You should now see all the files from the Quark folder shared with the VM + this VM has all the dependencies installed. Now we can compile and run Quark programs.

Compiling and Running Quark Programs

The following Hello World example Quark program is saved in /tests/hello_world.qk.

```quark
def int main:
{
    print("hello world");

    return 0;
}
```

Before we can compile Quark programs into C++ we must build the Quark compiler quarkc. Navigate to /vagrant/quark/ and run make. Ensure quarkc was properly built by checking for error messages then running ./quarkc -h to see compilation options. You should see the following:

```
usage: quarkc -s source.qk [-c output.cpp ] [-o executable] [ -static ] [-g++ /path/to/g++]
  -s : quark source file
  -c : generated C++ file. If unspecified, print generated code to stdout
  -o : compile to executable. Requires g++ (version >= 4.8)
  -sco : shorthand for -s <file>.qk -c <file>.cpp -o <file>
  -sc : shorthand for -s <file>.qk -c <file>.cpp
  -g++ : shorthand for -s <file>.qk -c <file>.cpp
  -static : compile with static lib (otherwise with dynamic lib). Does NOT work on OSX
  -help : Display this list of options
```

As stated above, to compile tests/hello_world.qk into C++ and an executable run ./quark/quarkc -s tests/hello_world.qk -c hello_world.cpp -o hello_world. You can run the hello_world executable ./hello_world to get the output, and cat hello_world.cpp shows the generated C++ as follows:

```
vagrant@vagrant-ubuntu-trusty-64:vagrant$ ./hello_world
hello world

vagrant@vagrant-ubuntu-trusty-64:vagrant$ cat hello_world.cpp
```
#include "qureg.h"
#include "qumat.h"
#include "qugate.h"
#include "quarklang.h"

using namespace Qumat;
using namespace Qugate;

int main()
{
    std::cout << std::boolalpha << std::setprecision(6) << std::string("hello world") << std::endl;
    return 0;
} // end main()

The C++ includes are referencing our quantum simulator and these files can be found in the lib directory.

To run some quantum computing programs, compile shor.qk and grover.qk in the quark folder. They are examples of non-trivial programs performing Shor's algorithm and Grover's search. Their implementation can be found in the testing section and the appendix.

Given an actual quantum computer, we would be able to run these algorithms in the stated time. For now, we run them on our simulator in exponential time for small N examples.

Shor's algorithm can factorize large integers in polynomial time. Run ./quark/quarkc -s quark/shor.qk -c shor.cpp -o shor and then run the executable ./shor

Grover's search can search an unsorted database in O(N^{1/2}) time. Run ./quark/quarkc -s quark/grover.qk -c grove.cpp -o grover and then run the executable ./grover

Essential Syntax

Quark syntax resembles something along the lines of Python with static typing. It is influenced by a number of languages including Python, MATLAB and C. If you already know a popular imperative language, you should be able to easily glean the majority of the syntax by simply reading through these examples. Our language manual provides a more explicit outline of the language spec.

```python
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;
}
```

The keyword def declares a function, followed by the return type of the function, the function name, :, and then comma-
separated, typed parameters. The main function is the primary entry point to Quark programs. print is a builtin function and the full list of builtins and a description of their function can be found in the Language Manual section. Blocks are denoted using brackets but are not required when the block is only composed of a single line of code. Statements are terminated with ; and single-line comments written after %.

Control Flow

Conditional statements are supported via the if, elif and else keywords. There's also support for the dangling else case. For example,

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

Here is a simple while loop,

```quark
while x > 42:
    print(x);
    x = x - 1;
```

for val in arr: will iterate over an array of any type. Or you can iterate over an array created using the range syntax: for val in [0:5] which iterates over [0,1,2,3,4] and for val in [0:10:3] which iterates over [0,3,6,9] where 3 is the step size.

Declaration

How you declare each type in Quark,

```quark
int i = 5;
float f = 3.0;
bool b = true;
string s = "So Long, and Thanks for All the Fish";
fraction f = 10/3;
complex c = i(1.0, 2.0);
string [] arr = ["Ford", "Prefect"];
int [] [] arr2 = [[1,2,3],[4,5,6]];
```

Matrices

Quark matrices have a different syntax than 2-dimensional arrays because they are compiled to Eigen classes as opposed to the built-in C++ vector. This also means n-dimensional arrays can have variable length rows whereas matrices must have equal length rows.
A matrix is declared as follows,

```plaintext
float[,] mat = [[1.0, 2.1; 3.2, 42.1]];
```

Where semicolons indicate the end of rows. Matrices can be composed of `int`, `float`, or `complex`. You can transpose a matrix by appending an apostrophe,

```plaintext
mat = mat';
```

And matrix elements can be accessed using the bracket notation and zero-indexed row and column integers,

```plaintext
float f = mat[2, 1];
```

**Quantum Registers**

And last but definitely not least, the quantum register. Quantum registers are a key component for constructing quantum circuits. When declaring quantum registers, the left value denotes the initial size of a quantum register, and the right value denotes the initial bit.

```plaintext
qreg q = <10, 0 >;
```

And `qreg` is the only type that is pass by reference.

You measure the state of a quantum register by using the destructive `?` operator and the non-destructive, but physically unrealistic, `?' operator.

```plaintext
qreg q = <10, 0 >;
int meas = q ? [2:10]; % measures qubits 2 through 10
```

Note, you can only measure LValue `qreg` variables. There are many builtin functions that utilize quantum registers and aid in building quantum circuits and you can find a complete listing in Language Reference.
A program in QUARK includes at least one function definition, though something trivial like a variable declaration or a string should compile. Programs are written using a basic source character set accepted by the C++ compiler in use. Refer to what source-code file encoding your compiler accepts. The QUARK compiler will only output ASCII.

Comments

MATLAB style commenting is supported. A MATLAB style comment begins with % and ends with %. Multi-line MATLAB comments start with %{ and end with }%. Any sequence of characters can appear inside of a comment except the string }%. These comments do not nest.

Whitespace

Whitespace is defined as the ASCII space, horizontal tab and form feed characters, as well as line terminators and comments.

Tokens

Tokens in QUARK consist of identifiers, keywords, constants, and separators. Whitespace is ignored and not taken into consideration.

Identifiers

An identifier is composed of a sequence of letters and digits, the first of which must be a letter. There is no limit on the length of an identifier. The underscore character _ is included in the regular expression pattern for letters.

Two identifiers are the same if they have the same ASCII character for every letter and digit.

\[
\begin{align*}
\text{digit} & \rightarrow [0-9] \\
\text{letter} & \rightarrow [a-z \ 'A'-'Z' \ '_'] \\
\text{Identifier} & \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*
\end{align*}
\]

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:
Reserved Prefix

There is only one reserved prefix in QUARK:

```_QUARK_```

Pseudo-reserved

The dollar sign separates the numerator value from the denominator value in a fraction data type.

Mathematical Constants

QUARK has two mathematical constants, \( \pi \) and \( e \). \( \pi \) is for mathematical constant \( \pi = 3.141592... \)

\( e \) for the mathematical constant \( e = 2.718281... \)

Punctuation

**Parenthesis** – Expressions can include expressions inside parenthesis. Parenthesis can also indicate a function call.

**Braces** – Braces indicate a block of statements.

**Semicolon** – Semicolons are used at the end of every statement as a terminator. Semicolons are also used to separate rows in the matrix data type.

**Colon** – Colons are used to denote slicing in arrays and within a function declaration. In a function declaration, formal arguments appear between the colon and a left curly brace.

**Dollar Sign** – The dollar sign separates the numerator value from the denominator value in a fraction data type.
**Comma** – Commas have several use cases. Commas are used to separate formal arguments in a function declaration, elements in arrays and matrices, and the size and initial state of a qreg.

**Escape Sequences**

Certain characters within strings need to be preceded by a backslash. These characters and the sequences to produce them in a string are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>“</td>
</tr>
<tr>
<td>\n</td>
<td>linefeed</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>horizontal tabulation</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
</tbody>
</table>

**Data Types**

The data types available in QUARK are:

- `int`
- `float`
- `fraction`
- `bool`
- `complex`
- `string`
- `qreg`
- `matrix`
- `void`

Additionally, the aggregate data type of array is available to the user.

- `int`

  An `int` is a 64-bit signed integer.

- `float`

  A `float` is a 64-bit signed floating-point number. Comparing two floats is done to a tolerance of $1e^{-6}$.

- `fraction`
A fraction is denoted by two int types separated by $$. The int value to the left of $$ represents the numerator, and the int value to the right of $$ represents the denominator. QUARK provides an inverse operator ~. 

\[
\text{frac foo = 2$3; \% represents 2/3}
\]
\[
\sim\text{foo; \% 3$2}
\]

Note on int, float, and fraction

Fraction types may be compare to int and float types using the following comparators: <, >, <=, >=, !, and ==. 

```plaintext
int i = 3;
float ft = 2.0;
frac f = 2$3;

i > f;
ft <= f;
```

bool

A bool value is denoted using the literals true or false.

complex

A complex type is generated from two int or float values; if given a mix of int and float types, QUARK will implicitly type cast. A complex type can also be generated with one numerical value, which will be assigned to the real part of a complex number; imaginary will default to 0. The real and imaginary parts of a complex number can be accessed by real and imag accessors.

```plaintext
complex cnum = i(3.0, 1);
real(cnum); \% 3.0
imag(cnum); \% 1
complex cnum2 = i(9) \% this gives us i(9, 0)
```

Comparing two complex numbers is done with a tolerance of 1e-6.

string

A string is a sequence of characters. String literals are placed between double quotations. QUARK supports string lexicographic comparison using <, >, <=, >=, !=, and ==.

creg

A creg type represents a quantum register. A creg accepts two int types. The left value denotes the initial size of a quantum register, and the right value denotes the initial bit.

```plaintext
creg q = <| 1, 1 |>;
```
Qreg must be passed as an LValue to any function:

```quark
% disallowed
hadamard(<|10, 3|>);
```

```quark
% allowed
qreg q = <|10, 3|>;
hadamard(q);
```

Additionally, qreg values may be measured using the destructive ? operator and the non-destructive (but unrealistic) ?' operator; the ? and ?' operators may only operate on an LValue of type qreg.

```quark
q ? [2:10]; % measures qubit 2 to 10
```

Matrix

QUARK allows you to create matrices; a matrix uses a special bracket notation to distinguish from arrays, and rows are separated by semicolons. Matrices may be composed of only int, float, or complex. Matrix elements may be accessed with a square bracket notation by separating the column and row index numbers by commas.

A new matrix with all zeros (real or complex) can be constructed by

type[ | row_dim, column_dim | ] and used in the middle of any expression.

QUARK provides the special prime operator ' for matrix transposition, and power operator ** overloaded for matrix kronecker product.

Matrices are zero indexed.

```quark
float[|] mat = [| 1.2, 3.4; 5.6, 7.8 |];
mat[2, 1];
complex[|] mat2 = complex[| 5, 9 |];
% constructs a 5-by-9 complex zero matrix
mat = mat'; % transpose matrix
mat ** mat2; % kronecker product
```

Array

QUARK allows arrays of any of the above data types. Arrays can have variable length and can be arbitrarily dimensional.

Arrays can be initialized using a comma-separated list delimited by square brackets[]. Additionally, arrays can be declared with a size to create an array of default-initialized elements.

Arrays are zero indexed.

Arrays may be concatenated with the & operator as long as there is a dimension and type match.

```quark
int[5]; % gives us [0,0,0,0,0]
int[] a = [1, 2, 3]; % array initialization
int[][] b = [[1,2,3], [4,5,6]]; % 2-d array
print(complex[16]);
% constructs an array of 16 i(0, 0)
```
[11, 22, 33] & int[3];
% gives us [11, 22, 33, 0, 0, 0]

Array indices can be accessed using the square bracket notation with an integer such as:

```python
int[] arr = [0, 1, 2];
arr[0];
```

or

```python
int[] arr = [0, 1, 2];
int i = 0;
arr[i];
```

Indices of multidimensional arrays may be accessed by separating the dimensional index numbers by commas:

```python
int[][] arr = [[0,1,2],[3,4,5]]
arr[1, 1]; % accesses 4
```

The built-in `len` function returns an `int` representing the length of the array.

Membership may be tested using the keyword `in`.

```python
int x = 5;
if x in [1:10]:
    % statement here is executed
```

`void` void

Void is a type for a function that returns normally, but does not provide a result value to the caller.

**Function types**

Functions take in zero or more variables of primitive or array types and optionally return a variable of primitive or array type. A function declaration always begins with `def`, the return type of the function, a colon `:` , and a list of formal parameters which may be empty.

```python
def void main: int x
{
    % statement
}
```

**Declarations**
Declaring a Variable

Variables can be defined within individual functions or in the global scope.Variables may be declared and then defined, or declared and defined simultaneously. An expression to which a value may be assigned is called an LValue.

```c
int x; % definition
x = 5; % declaration
int y = 6; % definition and declaration
```

x and y are LValues. LValues are named as such because they can appear on the left side of an assignment (though they may also appear on the right side).

Declaring an Array

As previously shown, arrays can be multidimensional, and may be of variable length. Arrays may be declared on their own with a size to get an uninitialized array of the given size. They can also be initialized with values upon declaration.

```c
int[5]; % gives us [0,0,0,0,0]
int[] a = [1, 2, 3]; % array initialization
int[][] b = [[1,2,3], [4,5,6]]; % 2-d array
```

Declaring a Matrix

A matrix declaration uses the special notation of piped square brackets. Matrix rows are distinguished using the ; separator between elements of rows. Initializing an empty complex matrix initializes an all-zero 3-by-4 complex matrix.

```c
float[][] floatmat = [[ 1.2, 3.4; 5.6, 7.8 ]];
complex[][] mat; % this gives us complex[[ 3, 4 ]] 
```

Operators

Arithmetic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>++</td>
<td>unary increment by one</td>
</tr>
<tr>
<td>--</td>
<td>unary decrement by one</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
</tbody>
</table>
**Operator**

* multiplication

mod modulo

** power

Concatenation

& String and array concatenation

**Assignment**

**Operator**

= assigns value or right hand side to left hand side

+= addition assignment

-= subtraction assignment

*= multiplication assignment

/= division assignment

&= bitand assignment

Assignment has right to left precedence.

**Logical**

**Operator**

!= not equal to

== equal to

> greater than

>= greater than or equal to

< less than

<= less than or equal to
and  unary and
or  unary or
not unary not

Bitwise Logical / Unary

**Operator**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise not and fraction inversion</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise xor</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Bitwise left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Bitwise right shift</td>
</tr>
</tbody>
</table>

Quantum

**Operator**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>quantum measurement query (destructive)</td>
</tr>
<tr>
<td>?'</td>
<td>quantum measurement query (non-destructive)</td>
</tr>
</tbody>
</table>

The `?` and `?` operators may only be invoked on an LValue.

```plaintext
q ? [2:10];  % measures qubit 2 to 10
```

Ternary Operator

QUARK supports Python style ternary conditional operators:

```plaintext
3 if true else 5;
4 if 3 > 2 if not (1==1) else 3 < 2 else -2 if true else 3
```

Ternary operators are right associative.
## Operator Precedence and Associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>* / mod</td>
<td>left</td>
</tr>
<tr>
<td>+ -</td>
<td>left</td>
</tr>
<tr>
<td>&gt;&gt; &lt;&lt;</td>
<td>left</td>
</tr>
<tr>
<td>&gt; &gt;= &lt; &lt;=</td>
<td>left</td>
</tr>
<tr>
<td>== !=</td>
<td>left</td>
</tr>
<tr>
<td>&amp;</td>
<td>left</td>
</tr>
<tr>
<td>^</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>left</td>
</tr>
<tr>
<td>or</td>
<td>left</td>
</tr>
<tr>
<td>?</td>
<td>right</td>
</tr>
<tr>
<td>in</td>
<td>left</td>
</tr>
<tr>
<td>= += -= *= /= &amp;=</td>
<td>right</td>
</tr>
</tbody>
</table>

Operators within the same row share the same precedence. Higher rows indicate higher precedence.

## Statements

Statements are the smallest components of a program used to express that an action is to be carried out. Statements are used for variable declarations and assignment, control flow, loops, function calls, and expressions. All statements end with a semicolon ;. Statements are used within blocks. The following are examples of statements and are by no means exhaustive:

```plaintext
string hello = "hello world";
int x = 10;
if x > 5
  foo(4);
while x != true
  for x in [1:10]
    4 + 6
qreg q0 = <| nbit * 2, 0 |>
```
Blocks

A block is defined to be inside curly braces \{ \}, which may include empty statements and variable declarations.

A block looks like:

\{
  % statements here
\}

Return Statement

The return keyword accepts an expression, and exits out of the nearest calling block or smallest containing function.

If elif else Statement

If statements take expressions that reduce to a boolean, and followed by a colon : and a statement block. If the following statement is only one line, curly braces are unnecessary.

QUARK allows elif statements, similarly to Python. The else and elif statements are optional.

```quark
if p == 1:
  return a;
if (3 > 1):
  { % multiple statements
  }
if (x == 3):
  % do something
elif (x == 4):
  % do something
else:
  % do something
```

While Loop

A while loop is of the form:

```quark
while(condition):
  { % statement
  }
```
As with `if elif else` statements, if the following statement is only one line, curly braces `{}` are unnecessary.

```python
while exp_mod(b, i, M) != 1:
    i ++;
```

The condition of the while loop may not be empty.

**For Statement**

QUARK supports two types of iterators, array and range, for its for statements.

**Array Iterator**

An array iterator allows you to sweep a variable across an array, evaluating the inner statement with identifiers assigned to a new value before each iteration. The identifier after `for` is assigned to the value of each element of the array sequentially.

The identifier may be declared ahead of time or within the for statement itself.

```python
int[] arr = [1,2,3];
for int i in arr:
    print i;
    % 1
    % 2
    % 3
```

**Range Iterator**

A range iterator allows you to sweep a variable across an array, evaluating the inner statement with identifiers assigned to a new value before each iteration. The identifier after `for` is assigned to each integer in the range.

The identifier may be declared ahead of time or within the for statement itself.

```python
int i;
for i in [1:10]
for int i in [1:10:2]
```

A range consists of three integers separated by colons `[start : stop : step]`. Start denotes the start of the range, stop denotes the exclusive end of a range, and step denotes the step size of the range. If the step and the last colon is excluded, the step is defaulted to 1. If the start value is excluded, it is defaulted to 0.

The following are various ways of declaring ranges:

```plaintext
0:5:2 % this gives us 0, 2, 4
:5 % 0, 1, 2, 3, 4
1:3 % 1, 2
```
Break and Continue

The **break** statement causes a while loop or for loop to terminate.

The **continue** statement provides a way to jump back to the top of a loop earlier than normal; it may be used to bypass the remainder of a loop for an iteration.

Functions

QUARK allows users to define functions.

Function Declaration

Functions are composed of the form:

```python
def return_type func_name: type arg1, type arg2
{
  % statements in function body
  return return_type;
}
```

Functions are defined only by identifying the block of code and the keyword `def`, giving the function a name, supplying it with zero or more formal arguments, and defining a function body. Function return types are of any data type previously described, or `void` for no value.

Some examples of function declarations are:

```python
def void hello:
{
  print("hello world");
}

def int addition: int x, int y
{
  return x + y;
}
```

Imports

QUARK allows users to import qk files containing QUARK statements and definitions. QUARK supports relative file paths. All files are presumed to have the `.qk` extension.

```python
import ../lib/mylib1;
import ../lib/herlib2;
```
import imported_file; % contains foo2

def int foo:
    { return foo2(5); }

Casting

QUARK does not allow explicit type casting.

Pass by value pass by reference

QUARK passes arguments by value. The only exception to this is the qreg type which is passed by reference. This is because qregs are too expensive (and meaningless) to copy in C++, our intermediary representation. If you need to copy a qreg, please use the explicit builtin function qclone().

Overloading

QUARK keeps separate symbol tables for functions and types, and as such the same identifier may be used as a variable and a function at the same time.

Built-in functions are overridable because they are not stored in the function table, with the exception of the following built-in functions:

- print (with \n)
- print_noline
- apply_oracle

Otherwise, function overloading by itself is not supported.

Scoping

In QUARK, there are both global and local scopes. QUARK uses block scoping. A block is a section of code contained by a function, a conditional (if/while), or looping construct (for). Variables defined in the global scope can be used in any function, as well as in any block within that function. Each nested block creates a new scope, and variables declared in the new scope supersede variables declared in higher scopes.

Below is an example of the subtleties of QUARK's scoping rules:

```
int i = 1000;
for i in [0:7:2]:
    { % statement }
```
% i changed.

```c
int i = 1000;
for int i in [0:7]:
{
    % statement
}
i; % still 1000
```

# Built-in Functions

Below is a list of built-in functions that QUARK provides:

**General**

- `print(any_type)`: takes any type and returns void
- `print_noline(any_type)`: print but does not add a newline
- `len([any_type])`: takes any array and returns the length

**Fraction**

- `num(fraction)`: takes a fraction type and returns the numerator as an int
- `denom(fraction)`: takes a fraction type and returns the denominator as an int.

**Complex**

- `real(complex)`: takes a complex type and returns the real portion of a complex number as a float.
- `imag(complex)`: takes a complex type and returns the imaginary portion of a complex number as a float.

**Math**

- `sqrt(float)`: takes the square root of a number and returns a float.
- `rand_int(int, int)`: takes two ints as boundaries and returns an int between them.
- `rand_float(float, float)`: takes two floats as boundaries and returns a float between them.

**Matrix**

- `coldim(any_matrix)`: returns the column dimension of a matrix as an int.
- `rowdim(any_matrix)`: returns the row dimension of a matrix as an

**Matrix Generation**
hadamard_mat(int) : takes int and returns a complex matrix.

cnot_mat() : takes nothing and returns a complex matrix.

toffoli_mat(int) : takes int and returns complex matrix.

generic_control_mat(int, complex_matrix) : takes int and complex matrix and returns complex matrix.

pauli_X_mat() : takes nothing and returns a complex matrix.

pauli_Y_mat() : takes nothing and returns a complex matrix.

pauli_Z_mat() : takes nothing and returns a complex matrix.

rot_X_mat(float) : takes a float and returns a complex matrix.

rot_Y_mat(float) : takes a float and returns a complex matrix.

rot_Z_mat(float) : takes a float and returns a complex matrix.

phase_scale_mat(float) : takes a float and returns a complex matrix.

phase_shift_mat(float) : takes a float and returns a complex matrix.

ccontrol_phase_shift_mat(float) : takes a float and returns a complex matrix.

swap_mat() : takes nothing and returns a complex matrix.

cswap_mat() : takes nothing and returns a complex matrix.

qft_mat(int) : takes int and returns a complex matrix.

grover_diffuse_mat(int) : takes int and returns a complex matrix.

Quantum Registers

qsize(qreg) : takes a qreg and returns an int.

qclone(qreg) : takes a qreg and returns a qreg.

prefix_prob(qreg, int, int) : takes a qreg, and int, and an int, and returns a float.

apply_oracle(qreg, function, int) : takes a qreg, a defined function, and an int, and returns void.

Quantum Gates (Functions apply a specific gate to a quantum register)

Single bit gates

hadamard(qreg) : takes a qreg and returns void.

hadamard_top(qreg, int) : takes qreg and int and returns void.

pauli_X(qreg, int) : takes qreg and int and returns void.

pauli_Y(qreg, int) : takes qreg and int and returns void.

pauli_Z(qreg, int) : takes qreg and int and returns void.
rot_X(qreg, float, int): takes qreg, float, and int and returns void.

rot_Y(qreg, float, int): takes qreg, float, and int and returns void.

rot_Z(qreg, float, int): takes qreg, float, and int and returns void.

phase_scale(qreg, float, int): takes qreg, float, and int and returns void.

phase_shift(qreg, float, int): takes qreg, float, and int and returns void.

Multi bit gates

generic_1gate(qreg, complex_matrix, int): takes qreg, complex matrix, and int and returns void.

generic_2gate(qreg, complex_matrix, int, int): takes qreg, complex matrix, int, and int and returns void.

generic_ngate(qreg, complex_matrix, [int]): takes qreg, complex matrix, and an array of ints and returns void.

Control Gates

cnot(qreg, int, int): takes qreg, int, and int, and returns void.

toffoli(qreg, int, int, int): takes qreg, int, int, and int and returns void.

control_phase_shift(qreg, float, int, int): takes a qreg, float, int, and int, and returns void.

ncnot(qreg, [int], int): takes a qreg, and array of ints, and int, and returns void.

generic_control(qreg, complex_matrix, int, int): takes qreg, complex matrix, int, and int, and returns void.

generic_toffoli(qreg, complex_matrix, int, int, int): takes qreg, complex matrix, int, int, and int, and returns void.

generic_ncontrol(qreg, complex_matrix, [int], int): takes qreg, complex matrix, an array of ints, and int, and returns void.

Other Gates

swap(qreg, int, int): takes qreg, int, and int, and returns void.

cswap(qreg, int, int, int): takes qreg, int, int, and int, and returns void.

qft(qreg, int, int): takes qreg, int, and int, and returns void.

grover_diffuse(qreg): takes qreg and returns void.

Grammar

Below is the grammar for QUARK. Words in capital letters are tokens passed in from the lexer.

ident:
  ID

vartype:
  INT
  | FLOAT
datatype:
  | vartype
  | datatype []
  | datatype [[]]

/* Variables that can be assigned a value */
lvalue:
  | ident
  | ident [expr_list]

expr:
/* Logical */
  | expr < expr
  | expr <= expr
  | expr > expr
  | expr >= expr
  | expr == expr
  | expr != expr
  | expr and expr
  | expr or expr

/* Unary */
  | ~expr
  | -expr
  | not expr
  | expr`

/* Arithmetic */
  | expr + expr
  | expr - expr
  | expr * expr
  | expr / expr
  | expr mod expr
  | expr ** expr

/* Bitwise */
  | expr & expr
  | expr ^ expr
  | expr | expr
  | expr << expr
  | expr >> expr

/* Query */
  | expr ? expr
  | expr ?` expr
  | expr ? [ : expr ]
  | expr ?` [ : expr ]
  | expr ? [expr : expr ]
  | expr ?` [expr : expr]

/* Parenthesis */
  | (expr)

/* Assignment */
lvalue = expr

/* Special assignment */
| lvalue += expr
| lvalue -= expr
| lvalue *= expr
| lvalue /= expr
| lvalue &= expr

/* Post operation */
| lvalue ++
| lvalue --

/* Membership testing with keyword 'in' */
| expr in expr

/* Python-style tertiary */
| expr if expr else expr

/* literals */
| INT_LITERAL
| FLOAT_LITERAL
| BOOLEAN_LITERAL
| expr $ expr
| STRING_LITERAL
| [ expr_list ]
| datatype [ expr ]
| [ ] matrix_row_list []
| datatype [ ] expr , expr []
| i( expr , expr )
| i( expr )
| <| expr , expr |>

/* function call */
| ident ()
| ident (expr_list)

expr_list:
| expr , expr_list
| expr

matrix_row_list:
| [ r00 , r01 ; r10 , r11 ; r20 , r21 ] /*
expr_list ; matrix_row_list
expr_list

decl:
| datatype ident = expr ;
| datatype ident ;

statement:
| if expr : statement else statement
| if expr : statement

| while expr : statement
| for iterator : statement
| {statement_seq}
expr;
;
decl
return expr;
return;

/* Control flow */
break
continue

iterator:
    | ident in [range]
    | datatype ident in [range]
    | datatype ident in expr

range:
    | expr : expr : expr
    | expr : expr
    | : expr : expr
    | : expr

top_level_statement:
    | def datatype ident : param_list {statement_seq}
    | datatype ident : param_list ;
    | decl

param:
    | datatype ident

non_empty_param_list:
    | param, non_empty_param_list
    | param

param_list:
    | non_empty_param_list
    | []

top_level:
    | top_level_statement top_level
    | top_level_statement

statement_seq:
    | statement statement_seq
    | []
Project Plan

Tools used:
- Trello for task assignment
- Git for version control
- GitHub for code management
- Vagrant with Ubuntu 14.04 LTS 64-bit for consistent development environments

We also used an external simulator that Jim created over the summer. Our compiler’s output is C++ specifically designed to work with the simulator.

Project Timeline:

These are goals we set for our project.

<table>
<thead>
<tr>
<th>Date</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/1/14</td>
<td>Complete scanner and parser</td>
</tr>
<tr>
<td>11/14/14</td>
<td>Complete semantic checking</td>
</tr>
<tr>
<td>11/25/14</td>
<td>Complete code generation</td>
</tr>
<tr>
<td>12/16/14</td>
<td>Complete test suite</td>
</tr>
<tr>
<td>12/1/14</td>
<td>Complete end-to-end</td>
</tr>
<tr>
<td>12/5/14</td>
<td>Finish testing and code freeze</td>
</tr>
<tr>
<td>12/8/14</td>
<td>Complete project report</td>
</tr>
</tbody>
</table>

Project Log:

Actual progress of project.

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestones</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/8/14</td>
<td>Team formed</td>
</tr>
<tr>
<td>9/9/14</td>
<td>Set up dev environment and GitHub repository</td>
</tr>
</tbody>
</table>
### Roles and Responsibilities

Here are our official roles for the project.

<table>
<thead>
<tr>
<th>Role</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Manager</td>
<td>Parthiban Loganathan</td>
</tr>
<tr>
<td>Language Guru</td>
<td>Jim Fan</td>
</tr>
<tr>
<td>System Architect</td>
<td>Jamis Johnson</td>
</tr>
<tr>
<td>Verification &amp; Validation</td>
<td>Daria Jung</td>
</tr>
</tbody>
</table>

In practice, we didn’t follow these roles very strictly. All of us worked on multiple parts of the code and took responsibility for whatever we touched. The parts of the compiler and project that we primarily worked on can roughly be split up into the following:
<table>
<thead>
<tr>
<th>Category</th>
<th>Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Management</td>
<td>Parthiban Loganathan</td>
</tr>
<tr>
<td>Language Reference Manual</td>
<td>Daria Jung</td>
</tr>
<tr>
<td>Scanner, Parser</td>
<td>Parthiban Loganathan, Daria Jung, Jim Fan</td>
</tr>
<tr>
<td>Semantic Checking</td>
<td>Jim Fan, Jamis Johnson</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Jim Fan</td>
</tr>
<tr>
<td>Testing</td>
<td>All</td>
</tr>
<tr>
<td>Simulator</td>
<td>Jim Fan</td>
</tr>
<tr>
<td>Project Report</td>
<td>All</td>
</tr>
<tr>
<td>Presentation</td>
<td>Parthiban Loganathan</td>
</tr>
</tbody>
</table>

Though some names are not listed under certain sections, it doesn't mean they didn't contribute towards it. For example, Daria helped with project management in the middle of the semester when things were hectic and no one was co-operating. All of us worked on different chunks of semantic checking before we decided on a major rewrite since it was hard to separate it from code generation. Due to our decision to follow the “democracy” approach as opposed to the “dictatorship” approach we faced issues with accountability, but each one of us also got to see more of the compiler in the process.
Architecture

Global Overview

The Quark architecture primarily consists of two major components, a compiler frontend and a simulator backend. The compiler translates Quark source code into C++ code, which is then compiled with Quark++ (simulator) headers by GNU g++.

When the program runs, it links with precompiled a Quark++ dynamic library and executes the quantum circuit simulation instructions. Optionally, the user can compile the generated C++ code with a static library to produce one portable executable, without any external dependencies. This option can be enabled with `quarkc -static`. It only works on Windows and Linux.

The Quark compiler is OS aware and will extract the correct library automatically. It supports all major OSes (tested on Windows 7 & 8, Mac OS X and Ubuntu 14.04).

Compiler Architecture

The compiler is written entirely in OCaml. This section outlines the compilation pipeline we design.

1. **Preprocessing**

   The preprocessor mainly resolves all import statements. The file path following each import is checked and added to a hashtable, which ensures that no circular or repetitive imports is allowed.

   The imported file can contain import themselves, so the preprocessor recursively expands all imported sources until no more import statements left.

2. **Scanning**

   The scanner tokenizes the source code into OCaml symbols. Details of the scanner rules can be found in the LRM.

3. **Parsing**

   The parser defines the syntactical rules of the Quark language. It takes the stream of tokens produced by the scanner as input, and produces an abstract syntax tree (AST), a recursive data structure.

   More details of the grammar can be found in the LRM.

4. **Semantic checking**

   The semantic checker ensures that no type conflicts, variable/function declaration/definition errors exist in a syntactically correct AST. It takes an AST as input and produces a similar recursive structure - the Semantic Abstract Syntax Tree (SAST).

   The struct `var_info` keeps information about a variable's type and depth in scope. A semantic exception is thrown if a variable with the same name is redeclared within the same scope.

   ```ocaml
   type var_info = {
      v_type: A.datatype;
      v_depth: int;
   }
   ```
The struct `func_info` keeps information about a function interface.

```haskell
type func_info = {
  f_args: A.datatype list;
  f_return: A.datatype;
  f_defined: bool; (* for forward declaration *)
}
```

The environment struct is carried along every recursive call to the semantic checker. It contains a variable table, a function table, the current scope depth, `is_returned` to check whether a function is properly returned, and `in_loop` to ensure that `continue` and `break` statements appear only in loops.

```haskell
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 *)
}
```

Our SAST is carefully designed to minimize code generation efforts. A major discovery is that the SAST does not need to carry the type information. Instead, a special `op_tag` is added, which contains all the information the code generator requires to produce C++.

For example, the binary ampersand `&` in Quark is used for both integer bitwise and and array/string concatenation. The SAST does not need to carry along the operands' type information to tell the code generator which meaning of `&` to translate. It only needs to tag the binary operator expression with either `OpVerbatim` or `OpConcat`.

A separate source file `builtin.ml` includes all the builtin function interfaces supported by the Quark++ simulator. The user, however, is free to The `print` and `print_noline` are special because they take an arbitrary number of arguments of arbitrary type.

The semantic checker also features very informative error messages to help the user debug better. The following is a few examples:

- "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"

5. **Code Generation**

The code generator takes an SAST as input and produces a string of translated C++ code, excluding the headers.

The following is a list of type mappings:

- `int` → C++ `int64_t`
- `float` → C++ `primitive float`
The generator relies on the op_tag given by the semantic checker to generate the right C++ function. For example, OpCastComplex1 instructs the generator to cast the first operand of a binary operator to std::complex<float>.

The generator uses a very special way to handle for-loops. Our for-range loop syntax is

```
for int i in [start : end : step]
```

When step is negative, the for loop must go in the reverse direction. The sign of step, however, is not available at compilation time. So the code generator uses system reserved temporary variables to handle this situation. The temporary identifier has the format _QUARK_[10_random_ascii_chars].

As an example, the following quark code

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

is compiled to

```
int64_t _QUARK_5H0aq5mw6x = 0;
int64_t _QUARK_v3YH0O1B0h = step();
int64_t _QUARK_l03AMaXh6u = _QUARK_v3YH0O1B0h > 0 ? 1 : -1;
for (int64_t i = 10; _QUARK_l03AMaXh6u * i < _QUARK_l03AMaXh6u * 0; i += _QUARK_v3YH0O1B0h) {
    std::cout << std::boolalpha << std::setprecision(6) << i << std::endl;
}
```

The code generator must conform to the Quark++ simulator library interface. In practice, the simulator has to be updated with minor changes to accommodate the compiled code as well as its interaction with the Eigen matrix library.

6. User Interface

Quarkc implements a number of command line arguments to improve user experience. Shorthand args are also provided for convenience.

The project is self-contained. It requires little to no user-managed dependencies.
The simulator is written by Jim Fan before the beginning of this term. It contains around 6,000 lines of C++ 11 code, compiles and runs successfully on Windows, Mac and Ubuntu.

It features a complete and optimized quantum circuit simulation engine that is able to run the most celebrated quantum algorithms ever conceived, including but not limited to Shor's factorization, Grover's search, Simon's period finding algorithm, etc. It can be included in other quantum computing research projects as a standalone library.
Test Plan

The test suite for QUARK consisted of simple regression tests for language features, as well as longer tests to demonstrate target programs in the language.

We created a set of test scripts in quark (with extension .qk) and expected output text files (with extension .output). The test suite script testall.sh compiles all test scripts and runs the output C++ using the simulator. If the output matches the corresponding expected output (.coutput) from the .output file, the test succeeds, else fails.

We tested each significant individual component of the language from the LRM with a separate test.

Rationale

We chose not to include unit testing and relied on OCaml's type system to detect major bugs or give us warnings about things such as missing cases in a pattern match.

Tests were run frequently to detect changes or unimplemented features. The regression test system allowed us to utilize test-driven development. The test suite was run with tests that used unimplemented language features, simply failing until those features are implemented.

Implementation

We created a tests folder in the Quark source repository and a shell script testall.sh to implement the build procedure.

To run tests, in the top level directory of the Quark repository, run ./testall.sh to:

1. Using the QUARK compiler quarkc, compile all programs with the extension .qk in tests to C++ files.
2. Using g++ 4.8, compile the generated C++ files to executables linked with the QUARK simulator libraries.
3. Run the executables and compare the outputs (.coutput) to the expected output files (.output). Any files with differing output are considered to be failing tests.

Add a .qk file to the tests folder with a corresponding .output file to add a new test.

Representative Program

Below is our implementation of Grover's search, a quantum algorithm for searching an unsorted database in $O(N^{1/2})$ time and a non-trivial example of a Quark program.

```quark
int nbit = 7;
int key = 73; % secret key

def int grover_oracle : int x
{
```
if $x$ == key:
    return 1;
else
    return 0;
}

def int main:
    {
        qreg q = <| nbit+1, 1 |>
        
        int N = 1 << nbit;
        int sqrtN = sqrt(N);

        hadamard(q);

        int ans = 0;
        float[] probAtKey;

        for int iter in [: sqrtN + 1] :
            {
                apply_oracle(q, "grover_oracle", nbit);

                hadamard_top(q, nbit);
                grover_diffuse(q);
                hadamard_top(q, nbit);

                if iter == sqrtN:
                    while grover_oracle(ans) == 0:
                        ans = q ? [nbit];

                        probAtKey &= [prefix_prob(q, nbit, key)];
            }

        print("Found key: ", ans);
        print("Probability = ", probAtKey);
        return 0;
    }

Another non-trivial program we wrote is Shor’s algorithm that factorizes large integers in polynomial time (if we had a real quantum computer).

int M = 221;

def int gcd: int a, int b
    {
        int c;
        while a != 0:
            {
                c = a;
                a = b mod a;
                b = c;
            }
        return b;
    }

def int exp_mod: int b, int e, int m
    {
        int remainder;
        int x = 1;
        ...
while e != 0:
    { remainder = e mod 2;
      e = e >> 1;

      if remainder == 1:
          x = (x * b) mod m;
          b = (b * b) mod m;
    }
    return x;
}

def int smallest_period: int b, int M

    { int i = 1;
      while exp_mod(b, i, M) != 1:
          i ++;
      return i;
    }

def int long_pow: int a, int p

    { if p == 1:
        return a;
    int partial = long_pow(a, p / 2);
    if p mod 2 == 1:
        return partial * partial * a;
    else
        return partial * partial;
    }

def int log2_int: int x

    { int ans = 0;
      while x > 0:
          { x = x >> 1;
            ans ++;
          }
      return ans;
    }

%[
    If size == 0, continue until 0
]

def int[] to_continued_fraction: fraction frac, int size

    { int[] cfrac;
      int i = 0;
      while size < 1 or i < size:
          { % array concatenation
              cfrac &= [num(frac) / denom(frac)];
              frac -= cfrac[i];
              if num(frac) == 0 : break;

              % denom/num built-in
              frac = ~frac;
              i ++;
          }
def fraction_to_fraction: int[] cfrac, int size
{
    if size < 1:
        size = len(cfrac);
    fraction ans = 1$cfrac[size - 1];
    for int i in [size-2 :0 :-1] :
    {
        ans += cfrac[i];
        ans = -ans;
    }
    return ans + cfrac[0];
}

int nbit = log2_int(M) + 1;

% This is the user defined function that should be passed as a string argument
def int shor_oracle: int x
{
    return exp_mod(nbit, x, M);
}

def int main:
{
    qreg q0 = <| nbit * 2, 0 |>
;
    qft(q0, 0, nbit);

    int b; int i;
    while true:
    {
        b = rand_int(2, M);
        if gcd(b, M) != 1: continue;
        qreg q = qclone(q0);
        apply_oracle(q, "shor_oracle", nbit);
        qft(q, 0, nbit);
        int mTrial = 0;
        int measured;

        while mTrial < 10:
        {
            mTrial ++;
            measured = q ?' [nbit];
            if measured != 0:
            {
                int[] cfrac = to_continued_fraction((1 << nbit)$measured, 0);
                for int size in [len(cfrac):0:-1] :
                {
                    int p = num(to_fraction(cfrac, size));
                    int P = p;

                    while P < 128 and P < M :
                    {
                        if P mod 2 == 0
                        }
and exp_mod(b, P, M) == 1 :
{
    int check = exp_mod(b, P / 2, M);
    if check != 1 and check != M - 1 :
    {
        int b_P_1 = long_pow(b, P / 2) - 1;
        int prime = gcd(M, b_P_1);

        if prime != 1 and prime != -1 :
        {
            int prime2 = gcd(M, b_P_1 + 2);
            print("Found period r = ", P);
            print("b ^ r = ", b, " ^ ", P, " = 1 mod ", M);
            print("b ^ (r/2) = ", b, " ^ ", P / 2, " = ", check, " mod ", M);
            int prime2 = gcd(M, b_P_1 + 2);
            print("gcd("M", "", ", b_P_1, ") = ", prime);
            print("gcd("M", "", ", b_P_1 + 2, ") = ", prime2);
            print("\nFactorize ", M, " = ", prime, " * ", M/prime if prime2==1 or prime2==-1
            else prime2);
            return 0;
        }
    }
    P += p;
}
}
print("FAIL");
return 0;
}

Tests Used

<table>
<thead>
<tr>
<th>Test name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd.qk</td>
<td>ensures QUARK passes the GCD test</td>
</tr>
<tr>
<td>addition.qk</td>
<td>ensures integer arithmetic works</td>
</tr>
<tr>
<td>array.qk</td>
<td>ensures that arrays can both be written to and read from</td>
</tr>
<tr>
<td>complex.qk</td>
<td>ensures QUARK's support for complex numbers works correctly</td>
</tr>
<tr>
<td>hello_world.qk</td>
<td>ensures basic print functionality works</td>
</tr>
<tr>
<td>import.qk</td>
<td>ensures QUARK's import system can correctly access code in another .qk file</td>
</tr>
<tr>
<td>logic.qk</td>
<td>ensures boolean logic works correctly</td>
</tr>
<tr>
<td>matrix.qk</td>
<td>ensures matrices can both be written to and read from</td>
</tr>
<tr>
<td>File</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>range.qk</td>
<td>ensures range iteration works correctly</td>
</tr>
<tr>
<td>while.qk</td>
<td>ensures while loops correctly execute based on condition</td>
</tr>
<tr>
<td>float.qk</td>
<td>ensures float arithmetic works</td>
</tr>
<tr>
<td>elif.qk</td>
<td>ensures else-if works correctly</td>
</tr>
<tr>
<td>fraction.qk</td>
<td>ensures fraction arithmetic works</td>
</tr>
<tr>
<td>multi-array.qk</td>
<td>ensures we are able to access and write to multi-dimensional arrays</td>
</tr>
<tr>
<td>qreg.qk</td>
<td>tests qreg initialization and query</td>
</tr>
<tr>
<td>shor.qk</td>
<td>non-trivial program in QUARK</td>
</tr>
<tr>
<td>grover.qk</td>
<td>non-trivial program in QUARK</td>
</tr>
</tbody>
</table>

Testing was primarily done by Daria Jung with help from Parthiban Loganathan and Jamis Johnson.
Lessons Learned

Parthiban Loganathan

1. We should have heeded the numerous warnings at the beginning of the class that we should start early. While we made good initial progress with team formation, setting up our environment and drafting a proposal, we failed to actually start working on the compiler till around the midterm - partly due to insufficient knowledge of how a compiler works. Slow and steady progress would have been much less stressful than working a large number of hours in the past few weeks.
2. Project management is hard. It was difficult to get everyone to meet periodically to discuss progress and language design. Unlike a company, where your primary responsibility is to be developing software, as students with other classes and responsibilities, the project was not a priority till the end of the semester.
3. Allocating work into sizable chunks was a challenge due to the interrelatedness of the different components. For example, even after defining interfaces, we often found minor specification differences between the parser and code generation led to issues.
4. Focus on the primary purpose of the language. We initially toyed with the idea of dynamic typing and other advanced features that did not come to fruition.

Jim Fan

1. Communication is key to successful teamwork, but it is also the hardest part of teamwork. When one group member disagrees with another on the high-level approach to a problem (e.g. semantic checking), the lack of effective communication will delay the whole group and give rise to misunderstanding. Since code speaks more than words, a working prototype of an idea is found to be more convincing than hours of persuasion.
2. We started semantic checking by looking at open source codes from PLT groups in the past. It turned out to be an unwise idea. We would have made progress much more quickly if we had not wasted so much time trying to decode someone else's uncommented source. Their language was simply too different from ours to be a worthwhile reference. We ended up writing every line of semantic checking and code generation from scratch.
3. OCaml is not as easy as I think. In addition to the slides in class, I read the relevant chapters in the O'Reilly book Real World OCaml after I get stuck in too many places.
4. Cross-platform compilation is hard. I was using Windows and the rest of the team were using Mac. I actually bought a new Mac for this purpose. It turned out that Mac did not support static library compilation, and did not conform to the "rpath" standard on Linux. I did miss Java at those moments of pain.

Daria Jung

Group projects are pretty frustrating when the group is comprised of several overworked university students. Real life can get in the way (one group member experienced a death in the family), things always take longer than expected (programmers are the worst at estimating how long something will take), and writing a compiler can get pretty complicated. Communication, or lack thereof, was an impediment to our progress when we started to get going, so it is imperative to be transparent and crystal clear to other teammates about what is happening. I wish that we had had Bob Martin's talk earlier in the semester so we had a better sense of the sorts of things to watch out for. The pace of the project was much different than working on something at a company.

Definitely start the project as early as you can, which I'm sure most people have said, or agree with. Things inevitably start to pile up
(interviews, school, midterms), and if you have buffer time, then things won't be as hectic in the last few weeks of the semester.

It's pretty difficult to delegate/divide up work, so I would have liked to pair program more. Inevitably, some of the work fell on certain people throughout the project due to the nature of everyone's different schedules.

Jamis Johnson

Communication is vital. Pick a time to meet every week for an hour or two and don't leave without knowing specifically who is doing what before next week's meeting. Decide immediately how you will all communicate (email, sms, facebook messenger) and constantly inform your teammates of your progress!

Jump in and start coding stat! The language spec changes rapidly so don't dwell on the minutiae. What matters most is the core objective of your language. Complete the first edition of the LRM by your team's first or second meeting and move on to writing code.

Team members will inevitably become overwhelmed with life, other school work, and with each other, and it's easy to get frustrated. Help each other out and go easy on one another (and see above: communicate!). Also, early progress will generally ameliorate stress, as will clearly defined roles.
Appendix

A.1 scanner.mll

Primary authors: Parthiban Loganathan, Daria Jung

```ml
{ open Parser }

let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z' '_']
let sign = ['+' '-' ]
let floating =
  digit+ ',' digit* | '.' digit+ 
  | digit+ ('.' digit)*? 'e' '-'? digit+ 
  | '.' digit+ 'e' '-'? digit+ 

rule token = parse
  (* whitespace *) 
  | [' ' '	' '' '\n'] { token lexbuf }

  (* meaningful character sequences *) 
  | '::' { SEMICOLON } 
  | ':.' { COLON } 
  | ',' { COMMA } 
  | '$' { DOLLAR } 
  | '(' { LPAREN } | ')' { RPAREN } 
  | '{' { LCURLY } | '}' { RCURLY } 
  | '[' { LSQUARE } | ']' { RSQUARE } 
  | '=' { ASSIGN } 
  | '' { PRIME } 
  | '?' { QUERY } 

  (* unrealistic query that doesn't disrupt quantum state *) 
  | '?' { QUERY_UNREAL } 
  | '<(' { COMPLEX_SYM } 
  | '<|' { LQREG } | '|>' { RQREG } 
  | '][' { LMATRIX } | '|' { RMATRIX } 
  | 'def' { DEF }

  (* arithmetic *) 
  | '+' { PLUS } 
  | '-' { MINUS } 
  | '*' { TIMES } 
  | '/' { DIVIDE } 
  | '' { MODULO }

  (* logical *)
  | '<' { LT } 
  | '<=' { LTE } 
  | '>' { GT } 
  | '>=' { GTE } 
  | '' { EQUALS }
```
and comments = parse
   | "\n" { token lexbuf }
   | _ { comments lexbuf }

and inline_comments = parse
   | "\n" { token lexbuf }
   | _ { inline_comments lexbuf }

A.2 parser.mly

Primary authors: Parthiban Loganathan, Daria Jung
Secondary author: Jim Fan

{% open Ast %}
{% open Type %}

%token LPAREN RPAREN LCURLY RCURLY LSQUARE RSQUARE
%token LQREG RQREG LMATRIX RMATRIX
%token COMMA SEMICOLON COLON
%token ASSIGN
%token QUERY QUERY_UNREAL
%token PLUS_EQUALS MINUS_EQUALS TIMES_EQUALS DIVIDE_EQUALS BITAND_EQUALS
%token LSHIFT_EQUALS RSHIFT_EQUALS BITOR_EQUALS BITXOR_EQUALS /* unused */
%token LSHIFT RSHIFT BITAND BITOR BITXOR AND OR
%token LT LTE GT GTE EQUALS NOT_EQUALS
%token PLUS MINUS TIMES DIVIDE MODULO
%token NOT UMINUS BITNOT DECREMENT INCREMENT
%token DOLLAR PRIME QUERY POWER COMPLEX_SYM
%token IF ELSE WHILE FOR IN
%token COMPLEX_LITERAL FRACTION_LITERAL
%token DEF
%token RETURN BREAK CONTINUE
%token EOF
%token BOOLEAN STRING INT FLOAT QREG FRACTION_LITERAL COMPLEX_LITERAL VOID
%token <string> ID TYPE STRING_LITERAL INT_LITERAL FLOAT_LITERAL BOOLEAN_LITERAL

%right ASSIGN PLUS_EQUALS MINUS_EQUALS TIMES_EQUALS DIVIDE_EQUALS BITAND_EQUALS

%nonassoc IFX
%nonassoc ELSE

%left IN
%right QUERY QUERY_UNREAL

%right IF
%left COMPLEX_SYM
%left OR
%left AND
%left BITOR
%left BITXOR
%left BITAND
%left EQUALS NOT_EQUALS
%left LT LTE GT GTE
%left LSHIFT RSHIFT
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%left DOLLAR
%left DEF

%right NOT BITNOT POWER UMINUS
%left PRIME /* matrix transpose */

%start top_level
%type <Ast.statement list> top_level

%

ident:
   ID { Ident($1) }

vartype:
   INT { Int }
   | FLOAT { Float }
   | BOOLEAN { Bool }
   | STRING { String }
   | QREG { Qreg }
   | FRACTION { Fraction }
   | COMPLEX { Complex }
   | VOID { Void }

datatype:
   | vartype { DataType($1) }
   | datatype LSQUARE RSQUARE { ArrayType($1) }
   | datatype LMATRIX RSQUARE { MatrixType($1) } /* int[][] */

/* Variables that can be assigned a value */
lvalue:
   | ident { Variable($1) }
   | ident LSQUARE expr_list RSQUARE { ArrayElem($1, $3) }

expr:
/* Logical */
   | expr LT expr { Binop($1, Less, $3) }
   | expr LTE expr { Binop($1, LessEq, $3) }
   | expr GT expr { Binop($1, Greater, $3) }
   | expr GTE expr { Binop($1, GreaterEq, $3) }
   | expr EQUALS expr { Binop($1, Eq, $3) }
   | expr NOT_EQUALS expr { Binop($1, NotEq, $3) }
   | expr AND expr { Binop($1, And, $3) }
   | expr OR expr { Binop($1, Or, $3) }

/* Unary */
   | BITNOT expr { Unop(BitNot, $2) }
   | MINUS expr %prec UMINUS { Unop(Neg, $2) }
   | NOT expr { Unop(Not, $2) }
   | expr PRIME { Unop(Transpose, $1) }

/* Arithmetic */
   | expr PLUS expr { Binop($1, Add, $3) }
   | expr MINUS expr { Binop($1, Sub, $3) }
   | expr TIMES expr { Binop($1, Mul, $3) }
   | expr DIVIDE expr { Binop($1, Div, $3) }
   | expr MODULO expr { Binop($1, Mod, $3) }
   | expr POWER expr { Binop($1, Pow, $3) }

/* Bitwise */
   | expr BITAND expr { Binop($1, BitAnd, $3) }

expr BITXOR expr { Binop($1, BitXor, $3) }
expr BITOR expr { Binop($1, BitOr, $3) }
expr LSHIFT expr { Binop($1, Lshift, $3) }
expr RSHIFT expr { Binop($1, Rshift, $3) }

/* Query */
expr QUERY expr { Queryop($1, Query, $3, IntLit("QuerySingleBit")) }
expr QUERY_UNREAL expr { Queryop($1, QueryUnreal, $3, IntLit("QuerySingleBit")) }
expr QUERY LSQUARE COLON expr RSQUARE { Queryop($1, Query, IntLit("0"), $5) }
expr QUERY_UNREAL LSQUARE COLON expr RSQUARE { Queryop($1, QueryUnreal, IntLit("0"), $5) }
expr QUERY LSQUARE expr COLON expr RSQUARE { Queryop($1, Query, $4, $6) }
expr QUERY_UNREAL LSQUARE expr COLON expr RSQUARE { Queryop($1, QueryUnreal, $4, $6) }

/* Parenthesis */
LPAREN expr RPAREN { $2 }

/* Assignment */
lvalue ASSIGN expr { Assign($1, $3) }
lvalue { Lval($1) }

/* Special assignment */
lvalue PLUS_EQUALS expr { AssignOp($1, AddEq, $3) }
lvalue MINUS_EQUALS expr { AssignOp($1, SubEq, $3) }
lvalue TIMES_EQUALS expr { AssignOp($1, MulEq, $3) }
lvalue DIVIDE_EQUALS expr { AssignOp($1, DivEq, $3) }
lvalue BITAND_EQUALS expr { AssignOp($1, BitAndEq, $3) }

/* Post operation */
lvalue INCREMENT { PostOp($1, Inc) }
lvalue DECREMENT { PostOp($1, Dec) }

/* Membership testing with keyword 'in' */
expr IN expr { Membership($1, $3) }

/* Python-style tertiary */
expr IF expr ELSE expr { Tertiary($1, $3, $5) }

/* literals */
INT_LITERAL { IntLit($1) }
FLOAT_LITERAL { FloatLit($1) }
BOOLEAN_LITERAL { BoolLit($1) }
expr DOLLAR expr { FractionLit($1, $3) }
STRING_LITERAL { StringLit($1) }
LSQUARE expr_list RSQUARE { ArrayLit($2) }
datatype LSQUARE expr RSQUARE { ArrayCtor($1, $3) }
LMATRIX matrix_row_list RMATRIX { MatrixLit($2) }
datatype LMATRIX expr COMMA expr RMATRIX { MatrixCtor($1, $3, $5) }
COMPLEX_SYM expr COMMA expr RPAREN { ComplexLit($2, $4) }
COMPLEX_SYM expr RPAREN { ComplexLit($2, FloatLit("0.0")) }
LOREG expr COMMA expr RQREG { QRegLit($2, $4) }

/* function call */
ident LPAREN RPAREN { FunctionCall($1, []) }
ident LPAREN expr_list RPAREN { FunctionCall($1, $3) }

expr_list:
expr COMMA expr_list { $1 :: $3 }
expr { [$1] }

/* [ [ r00, r01; r10, r11; r20, r21 ] */
matrix_row_list:
expr_list SEMICOLON matrix_row_list { $1 :: $3 }
expr_list { [$1] }

deq:
| datatype ident ASSIGN expr SEMICOLON { AssigningDecl($1, $2, $4) }
| datatype ident SEMICOLON { PrimitiveDecl($1, $2) }

statement:
| IF expr COLON statement ELSE statement { IfStatement($2, $4, $6) }
| IF expr COLON statement %prec IFX { IfStatement($2, $4, EmptyStatement) }
| WHILE expr COLON statement { WhileStatement($2, $4) }
| FOR iterator COLON statement { ForStatement($2, $4) }
| LCURLY statement_seq RCURLY { CompoundStatement($2) }
| expr SEMICOLON { Expression($1) }
| SEMICOLON { EmptyStatement }
| decl { Declaration($1) }
| RETURN expr SEMICOLON { ReturnStatement($2) }
| RETURN SEMICOLON { VoidReturnStatement }

/* Control flow */
| BREAK { BreakStatement }
| CONTINUE { ContinueStatement }

/* iterator_list:
| iterator COMMA iterator_list { $1 :: $3 }
| iterator { [$1] } */

iterator:
| ident IN LSQUARE range RSQUARE { RangeIterator(NoneType, $1, $4) }
| datatype ident IN LSQUARE range RSQUARE { RangeIterator($1, $2, $5) }
| datatype ident IN expr { ArrayIterator($1, $2, $5) }

range:
| expr COLON expr COLON expr { Range($1, $3, $5) }
| expr COLON expr { Range($1, $3, IntLit("1")) }
| COLON expr COLON expr { Range(IntLit("0"), $2, $4) }
| COLON expr { Range(IntLit("0"), $2, IntLit("1")) }

top_level_statement:
| DEF datatype ident COLON param_list LCURLY statement_seq RCURLY { FunctionDecl($2, $3, $5, $7) }
| datatype ident COLON param_list SEMICOLON { ForwardDecl($1, $2, $4) }
| decl { Declaration($1) }

param:
| datatype ident { PrimitiveDecl($1, $2) }

non_empty_param_list:
| param COMMA non_empty_param_list { $1 :: $3 }
| param { [$1] }

param_list:
| non_empty_param_list { $1 }
A.3 type.ml

Primary authors: Parthiban Loganathan, Daria Jung

type vartype =
  | Int
  | Float
  | Bool
  | Fraction
  | Complex
  | Qreg
  | String
  | Void

let str_of_type = function
  | Int -> "int"
  | Float -> "float"
  | Bool -> "bool"
  | Fraction -> "fraction"
  | Complex -> "complex"
  | Qreg -> "qreg"
  | String -> "string"
  | Void -> "void"

A.4 ast.ml

Primary authors: Parthiban Loganathan, Daria Jung
Secondary author: Jim Fan

module T = Type

type binop =
  | Add
  | Sub
  | Mul
  | Div
  | Mod
  | Pow
  | Lshift
type queryop =
    Query
    QueryUnreal

type unop =
    Neg
    Not
    BitNot
    Transpose

type postop =
    Dec
    Inc

type datatype =
    DataType of T.vartype
    ArrayType of datatype
    MatrixType of datatype
    NoneType (* if a symbol doesn't exist *)

type ident = Ident of string

type lvalue =
    Variable of ident
    ArrayElem of ident * expr list

and expr =
    Binop of expr * binop * expr
    AssignOp of lvalue * binop * expr
    Queryop of expr * queryop * expr * expr
    Unop of unop * expr
    PostOp of lvalue * postop
    Assign of lvalue * expr
    IntLit of string
    BoolLit of string
    FractionLit of expr * expr
    QRegLit of expr * expr
    FloatLit of string
    StringLit of string
    ArrayLit of expr list
    ArrayCtor of datatype * expr
    MatrixLit of expr list list
    MatrixCtor of datatype * expr * expr
| ComplexLit of expr * expr
| Lval of lvalue
| Membership of expr * expr
| FunctionCall of ident * expr list
| Tertiary of expr * expr * expr

type decl =
| PrimitiveDecl of datatype * ident
| AssigningDecl of datatype * ident * expr

type range = Range of expr * expr * expr

type iterator =
| RangeIterator of datatype * ident * range
| ArrayIterator of datatype * ident * expr

type statement =
| CompoundStatement of statement list
| Declaration of decl
| Expression of expr
| EmptyStatement
| IfStatement of expr * statement * statement
| WhileStatement of expr * statement
| ForStatement of iterator * statement
| FunctionDecl of datatype * ident * decl list * statement list
| ForwardDecl of datatype * ident * decl list
| ReturnStatement of expr
| VoidReturnStatement
| BreakStatement
| ContinueStatement

let rec str_of_datatype = function
  | DataType(t) ->
  T.str_of_type t
  | ArrayType(t) ->
    str_of_datatype t ^ "[]"
  | MatrixType(t) -> (match t with
    | DataType(elem_type) -> (match elem_type with
      (* only support 3 numerical types *)
      | T.Int | T.Float | T.Complex ->
        "[|" ^ T.str_of_type elem_type ^ "|]"
      | _ -> failwith "INTERNAL non-numerical matrix type to str"
    )
    | NoneType -> "[[AnyType]]"
    (* we shouldn't support float[][][] *]
    | _ ->
      failwith "INTERNAL bad matrix type to str"
  )
  | NoneType -> "AnyType"

let str_of_binop = function
  | Add -> "+
  | Sub -> "-
  | Mul -> "*
  | Div -> "/
  | Mod -> "mod"
  | Pow -> "**"
  | Lshift -> "<<"
let str_of_unop = function
| Neg -> "-"
| Not -> "not"
| BitNot -> "~"
| Transpose -> "transpose"
| _ -> failwith "INTERNAL unhandled unop"

let str_of_postop = function
| Inc -> "++"
| Dec -> "--"
| _ -> failwith "INTERNAL unhandled postop"

A.5 semantic.ml

Primary author: Jim Fan
Secondary authors: Jamis Johnson, Parthiban Loganathan, Daria Jung

module A = Ast
module S = Sast
module T = Type

module StrMap = Map.Make(String)

(* debug printout flag *)
let _DEBUG_ENABLE = false

(* utilities *)
let fst_2 = function x, _ -> x;;
let snd_2 = function _, x -> x;;
let fst_3 = function x, _, _ -> x;;
let snd_3 = function _, x, _ -> x;;
let trd_3 = function _, _, x -> x;;

let get_id (A.Ident name) =
  (* reserved prefix *)
  let forbid = Builtin.forbidden_prefix in
  let forbid_len = String.length forbid in
if String.length name < forbid_len then name
else
  if String.sub name @ forbid_len = forbid
  then failwith @@ "Identifier name cannot start with "
    ^ "the reserved prefix " ^ forbid ^ ": " ^ name
  else name

(****** Environment definition ******)

type func_info = {
  f_args: A.datatype list;
  f_return: A.datatype;
  f_defined: bool; (* for forward declaration *)
}

type var_info = {
  v_type: A.datatype;
  v_depth: int; (* how deep in scope *)
}

(* map string ident name to datatype or function info *)

type environment = {
  var_table: var_info StrMap.t;
  func_table: func_info StrMap.t;
  (* current function name waiting for 'return' *)
  func_current: string;
  depth: int;
  is_returned: bool;
  in_loop: bool; (* check break/continue validity *)
}

(************** DEBUG ONLY **************)

(* print out the func decl param list *)

let debug_s_decl_list f_args =
  let paramStr =
    List.fold_left
      (fun s typ -> s ^ (A.str_of_datatype typ) ^", ")
    ""
    f_args
  in
  if paramStr = "" then ""
  else
    String.sub paramStr 0 ((String.length paramStr) - 2)

(* print out the func decl param list *)

let debug_env env msg =
  if _DEBUG_ENABLE then
    begin
      print_endline @@ "ENV " ^ msg ^ "{";
      let paramStr =
        List.fold_left
          (fun s typ -> s ^ (A.str_of_datatype typ) ^", ")
        ""
        f_args
      in
      if paramStr = "" then ""
      else
        String.sub paramStr 0 ((String.length paramStr) - 2)
      end
      print_string "Var= ";
      StrMap.iter
        (fun key vinfo -> print_string @"
          key ^ ": " ^ A.str_of_datatype vinfo.v_type ^ "(" ^ string_of_int vinfo.v_depth ^ "); ")
        env.var_table;
      print_string "\nFunc= ";
      StrMap.iter
        (fun key finfo -> print_endline @"
          key ^ "(" ^ string_of_bool finfo.f_defined ^ "); " ^
            debug_s_decl_list finfo.f_args ^ " => " ^ A.str_of_datatype finfo.f_return ^ ");")
        env.func_table;
      print_endline @@ "Current= " ^ env.func_current;
      print_endline @@ "is_returned= " ^ string_of_bool env.is_returned;
      print_endline "}";
let debug_print msg = if _DEBUG_ENABLE then print_endline @@ "DEBUG " ^ msg

(****** Environment var_table ******)
(* return updated var_table field (keep env.depth) *)
let update_var_table env var_typ var_id =
  StrMap.add (get_id var_id)
  { v_type = var_typ; v_depth = env.depth }
  env.var_table

(* if doesn't exist, return NoneType *)
let get_env_var env var_id =
  let id = get_id var_id in
  try
    StrMap.find id env.var_table
  with Not_found ->
    (* if the identifier appears as a func_id, then error! *)
    if StrMap.mem id env.func_table then
      failwith @@ "A function is confused with a variable: " ^ id
    else
      { v_type = A.NoneType; v_depth = -1 }

let update_env_var env var_typ var_id =
  let vinfo = get_env_var env var_id in
  match vinfo.v_type with
  | A.NoneType
  | _ when vinfo.v_depth < env.depth -> (* we can safely add the var if it's in the inner scope *)
    { env with var_table = update_var_table env var_typ var_id }
  | _ -> failwith (@"Variable redeclaration: " ^ A.str_of_datatype var_typ ^ " " ^ get_id var_id

(* go one scope deeper *)
let incr_env_depth env =
  { env with depth = env.depth + 1 }
(* go one scope shallower *)
let decr_env_depth env =
  { env with depth = env.depth - 1 }

let set_env_returned env =
  { env with is_returned = true }

(****** Environment func_table ******)
let get_env_func env func_id =
  let fid = get_id func_id in
  try
    StrMap.find fid env.func_table
  with Not_found ->
    (* look at the built-in functions *)
    let arg_types, return_type = Builtin.find_builtin fid in
    { f_args = arg_types;
f_return = return_type;
f_defined = return_type <> A.NoneType;
}

(* Used in A.FunctionDecl *)
(* add all formal params to updated var_table *)
let update_env_func env return_type func_id s_param_list is_defined =
  let finfo = get_env_func env func_id in
  let errmsg_str = ": " ^ get_id func_id ^ "()" in
  let s_arg_types =
    List.map (function
      | A.PrimitiveDecl(typ, id) -> typ
      | _ -> failwith @@ "Function parameter list declaration error" ^ errmsg_str
    ) s_param_list
  in
  (* add formal params to var scope. This is a lambda function *)
  let add_formal_var_lambda =
    List.fold_left (
      fun env -> function
        | A.PrimitiveDecl(typ, id) -> update_env_var env typ id
        | _ -> failwith @@ "Function parameter list declaration error" ^ errmsg_str
      )
  in
  (* utility function *)
  let add_new_func_table_to_env func_table' =
  {
    var_table = env.var_table;
    func_table = StrMap.add (get_id func_id) func_table' env.func_table;
    func_current =
      (* if forward_decl, we don't have a func_current *)
      if is_defined then get_id func_id else "";
    depth = env.depth;
    is_returned = not is_defined; (* if not forward decl, we need to return *)
    in_loop = false;
  } in

  match finfo.f_return with
  | A.NoneType -> begin
    let func_table' = {
      (* only keep the formal param types *)
      f_args = s_arg_types;
      f_return = return_type;
      f_defined = is_defined
    } in
    let env' = add_new_func_table_to_env func_table' in
    if is_defined then
      (* add the formal param idents to scope *)
      add_formal_var_lambda env' s_param_list
    else
      (* simply forward decl, don't add stuff to scope *)
      env'
  end
  | _ when not finfo.f_defined ->
    if is_defined then
      (* check param list and return type, should be the same *)
      if finfo.f_return = return_type && finfo.f_args = s_arg_types then
        let func_table' = {
          f_args = finfo.f_args;
          f_return = finfo.f_return;
          f_defined = true
        } in
        let env' = add_new_func_table_to_env func_table' in
        add_formal_var_lambda env' s_param_list
else
  failwith @@ "Incompatible forward declaration" ^ errmsg_str
else
  failwith @@ "Function forward redeclaration" ^ errmsg_str
| _ -> failwith @@ "Function redefinition" ^ errmsg_str

(******* Helpers for gen_s_expr ******)
(* Fraction, Qureg, Complex *)
let compound_type_err_msg name type1 type2 =
  "Invalid " ^ name ^ " literal operands: " ^
  A.str_of_datatype type1 ^","^ A.str_of_datatype type2

let is_matrix = function
| A.MatrixType(_) -> true
| _ -> false

let is_lvalue = function
| A.Lval(_) -> true
| _ -> false

(* flatten a matrix (list list) into row major 1D list *)
let flatten_matrix = List.fold_left
  (fun acc row -> acc @ row ) []

(********* Main expr semantic checker entry *********)
(* return env', S.expr, type *)
let rec gen_s_expr env =
function
(* simple literals *)
| A.IntLit(i) -> env, S.IntLit(i), A.DataType(T.Int)
| A.BoolLit(b) -> env, S.BoolLit(b), A.DataType(T.Bool)
| A.FloatLit(f) -> env, S.FloatLit(f), A.DataType(T.Float)
| A.StringLit(s) -> env, S.StringLit(s), A.DataType(T.String)

(* compound literals *)
| A.FractionLit(num_ex, denom_ex) ->
  let
    env, s_num_ex, num_type = gen_s_expr env num_ex
  in
  let
    env, s_denom_ex, denom_type = gen_s_expr env denom_ex
  in
  match num_type, denom_type
  with
  | A.DataType(T.Int), A.DataType(T.Int) ->
    env, S.FractionLit(s_num_ex, s_denom_ex), A.DataType(T.Fraction)
  | _ -> failwith @@ compound_type_err_msg "fraction" num_type denom_type

| A.QRegLit(qex1, qex2) ->
  let
    env, s_qex1, q1_type = gen_s_expr env qex1
  in
  let
    env, s_qex2, q2_type = gen_s_expr env qex2
  in
  match q1_type, q2_type
  with
  | A.DataType(T.Int), A.DataType(T.Int) ->
    env, S.QRegLit(s_qex1, s_qex2), A.DataType(T.Qreg)
  | _ -> failwith @@ compound_type_err_msg "qreg" q1_type q2_type

| 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 
)

| A.ArrayLit(exprs) -> 
  let env, s_exprs, elem_type = gen_s_array env exprs in 
  let arr_type = A.ArrayType(elem_type) in 
  let _ = debug_print @@ "ARRAY " ^ (A.str_of_datatype arr_type) in 
  env, S.ArrayLit(arr_type, s_exprs), arr_type

(* constructs an array with size *)
| A.ArrayCtor(elem_type, size_expr) -> 
  let env, s_size_expr, size_type = gen_s_expr env size_expr in 
  if size_type = A.DataType(T.Int) then 
    let arr_type = A.ArrayType(elem_type) in 
    env, S.ArrayCtor(arr_type, s_size_expr), arr_type 
  else 
    failwith @@ "Array constructor size must be int, but " ^ A.str_of_datatype size_type ^ " provided"

| A.MatrixLit(exprs_list_list) -> 
  let env, s_matrix, elem_type, coldim = gen_s_matrix env exprs_list_list in 
  let elem_type = A.DataType(elem_type) in 
  let _ = debug_print @@ "MATRIX " ^ (A.str_of_datatype (A.MatrixType(elem_type))) ^ " cols= " ^ string_of_int coldim ^ " rows= " ^ string_of_int (List.length exprs_list_list) in 
  env, S.MatrixLit(elem_type, s_matrix, coldim), A.MatrixType(elem_type)

(* constructs a matrix with row, col dim *)
| A.MatrixCtor(elem_type, rowdim_ex, coldim_ex) -> ( 
  match elem_type with 
  | A.DataType(t) -> 
    if t = T.Int || t = T.Float || t = T.Complex then 
      let env, s_rowdim_ex, rowdim_type = gen_s_expr env rowdim_ex in 
      let env, s_coldim_ex, coldim_type = gen_s_expr env coldim_ex in 
      if rowdim_type = A.DataType(T.Int) && coldim_type = A.DataType(T.Int) then 
        let mat_type = A.MatrixType(elem_type) in 
        env, S.MatrixCtor(mat_type, s_rowdim_ex, s_coldim_ex), mat_type 
      else 
        failwith @@ "Matrix constructor row/column dimensions must be int/int, but " ^ A.str_of_datatype rowdim_type ^ "/" ^ A.str_of_datatype coldim_type ^ " provided" 
      else 
        failwith @@ "Non-numerical matrix constructor type: " ^ A.str_of_datatype elem_type 
      | _ -> failwith @@ "Invalid matrix constructor type: " ^ A.str_of_datatype elem_type 
  )

(* Binary ops *)
(* '+' used for matrix addition, '&' for array concatenation *)
| A.Binop(expr1, op, expr2) -> 
  (* helper functions for Binop case *)
  let err_msg_helper func_str_of op type1 type2 = 
    "Incompatible operands for binary op " ^ func_str_of type1 ^ " -.- " ^ func_str_of type2 in 
  let err_msg = err_msg_helper T.str_of_type in 
  (* basic types *)
  let err_msg_arrmat = err_msg_helper A.str_of_datatype in 
  (* array/matrix types *)
  let err_msg_arrmat = err_msg_helper A.str_of_datatype in 
  (* check left and right children *)
let env, s_expr1, ex_type1 = gen_s_expr env expr1 in
let env, s_expr2, ex_type2 = gen_s_expr env expr2 in
begin
match ex_type1, ex_type2 with
(* cases with raw types *)
| A.DataType(type1), A.DataType(type2) ->
  begin
    let logic_relational op type1 type2 =
      match typel, type2 with
      | T.Int, T.Int
      | T.Float, T.Float
      | T.Int, T.Float
      | T.Float, T.Int
      | T.Fraction, T.Fraction
      | T.Int, T.Fraction
      | T.Float, T.Fraction
      | T.Fraction, T.Int
      | T.Fraction, T.Float -> T.Bool, S.CastFraction2
      | t1, t2 -> failwith @@ err_msg op t1 t2
    in
    let binop_math op type1 type2 =
      let notmod = op <> A.Mod in
      let notmodpow = notmod && op <> A.Pow in
      match typel, 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, T.Fraction
      | T.Fraction, T.Int
      | T.Fraction, T.Fraction
      when notmodpow ->
        T.Fraction, S.CastFraction2
      | t1, t2 -> failwith @@ err_msg op t1 t2
    in
    let logic_basic op type1 type2 =
      match typel, type2 with
      | t1, t2 when t1 = t2 -> T.Bool, S.OpVerbatim
      | t1, t2 -> failwith @@ err_msg op t1 t2
    in
    let logic_equal op type1 type2 =
      match typel, type2 with
      | T.Float, T.Int
      | T.Int, T.Float
      | T.Float, T.Float
      | T.Complex, T.Complex -> T.Bool, S.OpFloatComparison
      | t1, t2 when t1 = t2 -> T.Bool, S.OpVerbatim
      | t1, t2 -> failwith @@ err_msg op t1 t2
    in
  in
end
let binop_bitwise op type1 type2 =  
  match type1, type2 with  
  | T.String, T.String when op = A.BitAnd ->  
    T.String, S.OpStringConcat  
  | t1, t2 -> failwith @@ err_msg op t1 t2  
in  
let result_type, optag =  
  match op with  
  | A.Add | A.Sub | A.Mul | A.Div | A.Pow | A.Mod | A.Eq | A.NotEq | A.Last | A.LessEq | A.Greater | A.GreaterEq | A.And | A.Or | A.BitAnd | A.BitOr | A.BitXor | A.Lshift | A.Rshift | _ -> failwith "INTERNAL unmatched binop"  
  in  
env, S.Binop(s_expr1, op, s_expr2, optag), A.DataType(result_type)  
end  

(* At least one of the binop operand is an array/matrix *)
| type1, type2 ->  
let result_type, optag =  
  match op with  
  | A.Eq | A.NotEq when type1 = type2 ->  
    A.DataType(T.Bool),  
    if is_matrix type1 then (* matrix floats should be compared with tol *)  
      S.OpFloatComparison else S.OpVerbatim  
  | A.Add | A.Sub | A.Mul | A.Pow | A.Mod | A.Eq | A.NotEq | A.Last | A.LessEq | A.Greater | A.GreaterEq | A.And | A.Or | A.BitAnd | A.BitOr | A.BitXor | A.Lshift | A.Rshift | _ -> failwith "INTERNAL unmatched binop"  
  in  
env, S.Binop(s_expr1, op, s_expr2, optag), result_type  
end  

(* Query ops *)
| A.Queryop(qreg_ex, op, start_ex, end_ex) ->  
let env, s_qreg_ex, qreg_type = gen_s_expr env qreg_ex  
begin  
match qreg_type with  
  | A.DataType(T.Qreg) ->  
    (* we disallow measurement on an rvalue, e.g. a qureg literal *)  
    let _ = if not (is_lvalue qreg_ex) then  
      failwith "Measurement query on a Qreg must be made on lvalue type"  
    in  
  | _ -> failwith "INTERNAL unmatched binop"  
  in  
env, s_qreg_ex, qreg_type = gen_s_expr env qreg_ex in  
begin  
match qreg_type with  
  | A.DataType(T.Qreg) ->  
    (* we disallow measurement on an rvalue, e.g. a qureg literal *)  
    let _ = if not (is_lvalue qreg_ex) then  
      failwith "Measurement query on a Qreg must be made on lvalue type"  
    in  
  | _ -> failwith "INTERNAL unmatched binop"  
  in  
env, s_qreg_ex, qreg_type = gen_s_expr env qreg_ex in
let env, s_end_ex, end_type = gen_s_expr env end_ex in
let optag = match s_end_ex with
| S.IntLit("QuerySingleBit") -> S.OpQuerySingleBit
| _ -> S.OpVerbatim in (match start_type, end_type with
| A.DataType(T.Int), A.DataType(T.Int) ->
(* query check success *)
env, S.Queryop(s_qreg_ex, op, s_start_ex, s_end_ex, optag), A.DataType(T.Int)
| _ -> failwith @@ "Incompatible query args: " ^ A.str_of_datatype start_type
^ (if optag = S.OpVerbatim then ", " ^ A.str_of_datatype end_type else "")
)
| _ -> failwith @@ "Measurement must be queried on a qureg, not " ^ A.str_of_datatype qreg_type
end

(* Unary ops *)
| A.Unop(op, ex) ->
let env, s_ex, typ = gen_s_expr env ex in
let err_msg op t = "Incompatible operand for unary op 
" ^ A.str_of_unop op ^ ": 
" ^ A.str_of_datatype t in
let return_type, optag = if is_matrix typ then
(* matrix support negation and transposition *)
let optag = match op with
| A.Neg -> S.OpVerbatim
| A.Transpose -> S.OpMatrixTranspose
| _ -> failwith @@ err_msg op typ
in
typ, optag
else (match op with
| A.Neg -> (match raw_type with
| A.DataType(t) -> t
| _ -> failwith @@ err_msg op typ)
in
match op with
| A.Neg -> (match raw_type with
| T.Int | T.Float | T.Fraction | T.Complex -> raw_type
| _ -> failwith @@ err_msg op typ)
| A.Not -> (match raw_type with
| T.Bool -> T.Bool
| _ -> failwith @@ err_msg op typ)
| A.BitNot -> (match raw_type with
(* ~fraction inverts the fraction *)
| T.Int | T.Fraction -> raw_type
| _ -> failwith @@ err_msg op typ)
in
S.OpVerbatim
)
let return_type, optag = if is_matrix typ then
(* matrix support negation and transposition *)
let optag = match op with
| A.Neg -> S.OpVerbatim
| A.Transpose -> S.OpMatrixTranspose
| _ -> failwith @@ err_msg op typ
in
match op with
| A.Neg -> (match raw_type with
| A.DataType(t) -> t
| _ -> failwith @@ err_msg op typ)
in
match op with
| A.Neg -> (match raw_type with
| T.Int | T.Float | T.Fraction | T.Complex -> raw_type
| _ -> failwith @@ err_msg op typ)
| A.Not -> (match raw_type with
| T.Bool -> T.Bool
| _ -> failwith @@ err_msg op typ)
| A.BitNot -> (match raw_type with
(* ~fraction inverts the fraction *)
| T.Int | T.Fraction -> raw_type
| _ -> failwith @@ err_msg op typ)
in
S.OpVerbatim
)
let return_type, optag = if is_matrix typ then
(* matrix support negation and transposition *)
let optag = match op with
| A.Neg -> S.OpVerbatim
| A.Transpose -> S.OpMatrixTranspose
| _ -> failwith @@ err_msg op typ
in
match op with
| A.Neg -> (match raw_type with
| A.DataType(t) -> t
| _ -> failwith @@ err_msg op typ)
in
match op with
| A.Neg -> (match raw_type with
| T.Int | T.Float | T.Fraction | T.Complex -> raw_type
| _ -> failwith @@ err_msg op typ)
| A.Not -> (match raw_type with
| T.Bool -> T.Bool
| _ -> failwith @@ err_msg op typ)
| A.BitNot -> (match raw_type with
(* ~fraction inverts the fraction *)
| T.Int | T.Fraction -> raw_type
| _ -> failwith @@ err_msg op typ)
in
S.OpVerbatim
)
env, S.Unop(op, s_ex, optag), return_type

| A.Lval(lval) ->
let s_lval, ltype = match lval with
| A.Variable(id) ->
let vtype = (get_env_var env id).v_type in
let idstr = get_id id in
if vtype = A.NoneType then
failwith @@ "Variable " ^ idstr ^ " is undefined"
else
S.Variable(idstr), vtype
Array/matrix lvalue e.g. arr[2,3,4] *)
| A.ArrayElem(id, ex_list) ->
  let vtype = (get_env_var env id).v_type in
  let idstr = get_id id in
  if vtype = A.NoneType then
    failwith @@ "Array/Matrix " ^ idstr ^ " is undefined"
  else
    let sub_dim = List.length ex_list in (* subscript [2,3,4] dimension *)
    let s_ex_list = (* check subscript types, must all be ints *)
      List.map (fun ex ->
        let _, s_ex, typ = gen_s_expr env ex in
        if typ = A.DataType(T.Int) then s_ex
        else failwith @@ "Subscript contains non-int: " ^ idstr ^
          "[" ^ A.str_of_datatype typ ^ "]") ex_list
    in
    match vtype with
    | A.ArrayType(elem_type) ->
      (* dim(original array) = dim(result lval) + dim(subscript) *)
      (* think of this as de-[] operation *)
      let rec get_array_lval_type sub_dim elem =
        if sub_dim = 0 then elem
        else match elem with
          | A.DataType(_) ->
            failwith @@ "Invalid subscript dimension for array: " ^ idstr
          | A.ArrayType(elem') ->
            get_array_lval_type (sub_dim - 1) elem'
            (* assume decl has already checked that matrix type is valid *)
          | A.MatrixType(A.DataType(raw_elem)) ->
            if sub_dim = 2 then A.DataType(raw_elem)
            else failwith @@
              "Invalid subscript dimension for array that contains matrix: " ^ idstr
          | _ -> failwith @@ "INTERNAL bad array type: " ^ idstr
        in
      let lval_type = get_array_lval_type (sub_dim - 1) elem_type in
      S.ArrayElem(idstr, s_ex_list), lval_type
    | A.MatrixType(elem_type) ->
      if sub_dim = 2 then
        match elem_type with
          | A.DataType(_) ->
            S.MatrixElem(idstr, s_ex_list), elem_type
          | _ -> failwith @@ "INTERNAL bad matrix type should've been handled in S.decl: " ^ idstr
    else failwith @@ "Subscript of matrix " ^ idstr
      " must have 2 args, but " ^ string_of_int sub_dim ^ " provided"
  (* bad lvalue *)
  | _ -> failwith @@ idstr ^ " is not an array/matrix"
in
env, S.Lval(s_lval), ltype

(* Special assignment *)
| A.AssignOp(lval, op, ex) ->
  let binop = match op with
    | A.AddEq -> A.Add
    | A.SubEq -> A.Sub
| A.MulEq -> A.Mul |
| A.DivEq -> A.Div |
| A.BitAndEq -> A.BitAnd |
| _ -> failwith @@ "INTERNAL bad AssignOp: " ^ A.str_of_binop op |
in |
gen_s_expr env (A.Assign(lval, A.Binop(A.Lval(lval), binop, ex))) |

(* Post ++ and -- *)
| A.PostOp(lval, op) -> |
| let env, s_lval_ex, typ = gen_s_expr env (A.Lval(lval)) in |
| let s_lval = match s_lval_ex with |
| S.Lval(s_lval) -> s_lval |
| _ -> failwith "INTERNAL in postop: doesn't return S.Lval as expected" |
in |
| match typ with |
| A.DataType(T.Int) | A.DataType(T.Float) -> |
| env, S.PostOp(s_lval, op), typ |
| _ -> failwith @@ "Incompatible operand for post op " ^ A.str_of_postop op ^ ": " ^ A.str_of_datatype typ |

(* Assignment *)
| A.Assign(lval, rhs_ex) -> |
| let env, s_lval_ex, l_type = gen_s_expr env (A.Lval(lval)) in |
| let env, s_rhs_ex, r_type = gen_s_expr env rhs_ex in |
| let s_lval = match s_lval_ex with |
| S.Lval(s_lval) -> s_lval |
| _ -> failwith "INTERNAL in postop: doesn't return S.Lval as expected" |
in |
| let return_type = if l_type = r_type then l_type |
else |
match l_type, r_type with |
| A.DataType(T.Int), A.DataType(T.Float) -> A.DataType(T.Int) |
| A.DataType(T.Float), A.DataType(T.Int) -> A.DataType(T.Float) |
| _ -> failwith @@ "Assignment type mismatch: " ^ A.str_of_datatype l_type ^ ": " ^ A.str_of_datatype r_type |
in |
| _ = debug_print @@ "ASSIGN returns " ^ A.str_of_datatype return_type |
in |
env, S.Assign(s_lval, s_rhs_ex), return_type |

(* Function calls *)
| A.FunctionCall(func_id, ex_list) -> |
| let env, s_rsc_ex, func_id = gen_s_expr env func_id in |
| let fidstr = get_id func_id in |
if Builtin.is_print fidstr then |
(* 'print' built-in functions support any number of args *)
(* We keep a bool list of whether each arg is a matrix. For eigen prettyprint *)
| let s_rsc_list, is_matrix_list = |
| List.fold_right ( (* fold right so we don't have to List.rev *) |
| fun (ex, is_matrix_list) -> |
| let _, s_rsc, ex_type = gen_s_expr env ex in |
| if ex_type = A.DataType(T.Void) then |
| failwith "Cannot print void type" |
else |
| s_rsc :: s_rsc_list, (is_matrix ex_type) :: is_matrix_list |
| ) ex_list ([], []) |
in |
env, S.FunctionCall(fidstr, s_rsc_list, is_matrix_list), finfo.f_return |
else (* non-special cases *)
if finfo.f_defined then |
let f_args = finfo.f_args in
let farg_len = List.length f_args in
let actual_len = List.length ex_list in
if farg_len = actual_len then
  let s_ex_list = List.map2 (fun ex f_arg ->
    (* check ex type must agree with expected arg type *)
    let _, s_ex, ex_type = gen_s_expr env ex in
    match ex_type, f_arg with
    | A.DataType(T.Int), A.DataType(T.Float)
    | A.DataType(T.Float), A.DataType(T.Int)
      (* Array(None) means built-in function matches any array type *)
    | A.ArrayType(_), A.ArrayType(A.NoneType)
    | A.MatrixType(_), A.MatrixType(A.NoneType) -> s_ex
    | ex_type', f_arg' when ex_type' = f_arg' ->
      if ex_type' = A.DataType(T.Qreg) then
        (* disallow non-lvalue qreg to be used as function parameter *)
        if is_lvalue ex then
          s_ex
        else
          failwith @@
            "Qreg parameter to function " ^ fidstr ^ "() must be lvalue type"
      else
        s_ex
    | _ -> failwith @@
      "Incompatible args for function " ^ fidstr ^ ": "
      ^ A.str_of_datatype ex_type ^ " given but "
      ^ A.str_of_datatype f_arg ^ " expected"
  ) ex_list f_args
in
(* check apply_oracle: arg#2(string) must represent a function(int) returns int *)
let _ = if fidstr = "apply_oracle" then
  let oracle_ex = List.nth ex_list 1 in
  let oracle_id = match oracle_ex with
    | A.StringLit(id) -> id
    | _ -> failwith "Arg #2 of built-in apply_oracle() must be a string literal"
  in
  let oracle_finfo = get_env_func env (A.Ident(oracle_id)) in
  if oracle_finfo.f_args <> [A.DataType(T.Int)]
  || oracle_finfo.f_return <> A.DataType(T.Int) then
    failwith @@
      "Arg #2 of built-in apply_oracle(): user-defined function "
      ^ oracle_id ^ " must have signature 'int " ^ oracle_id ^ "(int)'"
else
  failwith @@
    "Function " ^ fidstr ^ " requires " ^ string_of_int farg_len
  ^ " arg but " ^ string_of_int actual_len ^ " provided"
else
  failwith @@ if finfo.f_return = A.NoneType then
    "Function " ^ fidstr ^ " is undefined" else
    "Only forward declaration, no actual definition is found for " ^ fidstr

(* Membership testing with keyword 'in' *)
| A.Membership(elem, array_ex) ->
  let env, s_array_ex, array_type = gen_s_expr env array_ex in
  let env, s_elem, elem_type = gen_s_expr env elem in
  let arr_elem_type = match array_type with
    | A.ArrayType(elem_type) -> elem_type
    | _ -> failwith @@
      "Membership testing must operate on array type, not " ^ A.str_of_datatype array_type
  in
  let _ = match elem_type, arr_elem_type with
    | A.DataType(T.Int), A.DataType(T.Float)
    | A.DataType(T.Float), A.DataType(T.Int) -> ()
    | elem_type', arr_elem_type' when elem_type' = arr_elem_type' -> ()
    | _ -> failwith @@ "Membership testing has incompatible types: "

A.str_of_datatype elem_type "-.-" A.str_of_datatype arr_elem_type

in
env, S.Membership(s_elem, s_array_ex), A.DataType(T.Bool)

(* Python style tertiary *)
| A.Tertiary(true_ex, pred, false_ex) ->
  let env, s_pred, pred_type = gen_s_expr env pred in
  if pred_type = A.DataType(T.Bool) then
    let env, s_true_ex, true_type = gen_s_expr env true_ex in
    let env, s_false_ex, false_type = gen_s_expr env false_ex in
    let result_type, optag = match true_type, false_type with
    | A.DataType(t), A.DataType(f) ->
      let ret, optag = match t, f with
      | T.Int, T.Float
      | T.Float, T.Int -> T.Float, S.OpVerbatim
      | T.Int, T.Fraction
      | T.Fraction, T.Int -> T.Fraction, S.CastFraction1
      | T.Fraction, T.Float -> T.Fraction, S.CastFraction2
      | T.Int, T.Complex
      | T.Complex, T.Float -> T.Complex, S.CastComplex1
      | T.Float, T.Complex -> T.Complex, S.CastComplex2
      | t', f' when t' = f' -> t', S.OpVerbatim
      | _ -> failwith @@ "Tertiary expression has incompatible types: "
          ^ A.str_of_datatype t "-.-" A.str_of_datatype f in A.DataType(ret), optag
    | true', false' when true' = false' -> true', S.OpVerbatim
    | _ -> failwith @@ "Tertiary expression has incompatible types: "
          ^ A.str_of_datatype true_type "-.-" A.str_of_datatype false_type
    in
    env, S.Tertiary(s_true_ex, s_pred, s_false_ex, optag), result_type
  else
    failwith @@
    "Tertiary predicate must be bool, but " ^ A.str_of_datatype pred_type " provided"
    | _ -> failwith "INTERNAL some expr not properly checked"

and gen_s_array env exprs =
  let env, s_exprs, array_type = List.fold_left
  (* evaluate each expression in the list *)
  (fun (env, checked_exprs_acc, prev_type) unchecked_expr ->
    let env, checked_expr, expr_type = gen_s_expr env unchecked_expr in
    match prev_type with
    | A.NoneType -> (* means we're seeing the 1st expr in the array and now know array type *)
      env, checked_expr :: checked_exprs_acc, expr_type
    | Array Type -> ( (* ensure all elems in array are the same type *)
      match array_type, expr_type with (* <> for != *)
      | A.DataType(T.Int), A.DataType(T.Float)
      | A.DataType(T.Float), A.DataType(T.Int) ->
        env, checked_expr :: checked_exprs_acc, A.DataType(T.Float)
      | _ when array_type = expr_type ->
        env, checked_expr :: checked_exprs_acc, array_type
      | _ -> failwith @@ "Array element type conflict: "
          ^ A.str_of_datatype array_type "-.-" A.str_of_datatype expr_type
    )
  )
  (env, [], A.NoneType) exprs in
(env, List.rev s_exprs, array_type)
and gen_s_matrix env exprs_list_list =
  let env, matrix, matrix_type, row_length = List.fold_left
  (fun (env, rows, curr_type, row_length) exprs ->
   (* evaluate each row where each row is an expr list *)
   let env, exprs, row_type = gen_s_array env exprs in
   let prev_type =
     match row_type with
     | A.DataType(prev_type) ->
     let prev_type =
       match prev_type with
       | T.Int | T.Float | T.Complex -> prev_type
       | _ -> failwith @@ "Matrix element type unsupported: " ^ T.str_of_type prev_type
     )
     | _ -> failwith @@ "Invalid matrix row type: " ^ A.str_of_datatype row_type
   in
   let exprs_length = List.length exprs in
   match curr_type with
   | T.Void ->
     (* means this is the 1st row which means we now know the matrix type *)
     env, (exprs :: rows), prev_type, exprs_length
   | _ ->
     let curr_type' =
     (* the same length*)
     if row_length <> exprs_length
     then failwith "All rows in a matrix must have the same length"
     else (
     (* ensure all rows have the same type and can only be complex, int or float *)
     match curr_type, prev_type with
     | T.Float, T.Int
     | T.Int, T.Float
     | T.Float, T.Float -> T.Float
     | T.Int, T.Int -> T.Int
     | T.Complex, T.Complex -> T.Complex
     | _ -> failwith @@
       "Array element type conflict: " ^ T.str_of_type curr_type ^ " -.- " ^ T.str_of_type prev_type
     )
     in
   (env, (exprs :: rows), curr_type', row_length ))
  (env, [], T.Void, 0) exprs_list_list
  in
  let matrix = flatten_matrix (List.rev matrix) in
  env, matrix , matrix_type, row_length

  let gen_s_param =
  | A.PrimitiveDecl(typ, id) ->
    S.PrimitiveDecl(typ, get_id id)
  | _ -> failwith "Function parameter list declaration error"

  (* Used in A.FunctionDecl *)
  let gen_s_param_list param_list =
  List.map
  (fun param -> gen_s_param param) param_list

  (* decl *)
  let rec check_matrix_decl idstr typ =
  match typ with
  | A.DataType(_) -> ()
  | A.ArrayType(t) -> check_matrix_decl idstr t

A.MatrixType(t) -> (match t with
| A.DataType(mat_type) -> (match mat_type with
  (* only support 3 numerical types *)
  | T.Int | T.Float | T.Complex -> ()
  | _ -> failwith @@ "Non-numerical matrix declaration: " ^ idstr ^ " with " ^ T.str_of_type mat_type
  (* we shouldn’t support float[][][] *)
  | _ -> failwith @@ "Invalid matrix declaration: " ^ idstr^ " with " ^ A.str_of_datatype t
| A.NoneType -> failwith "INTERNAL NoneType encountered in check_matrix"

(* update_env_var checks redeclaration error *)
let gen_s_decl env = function
| A.AssigningDecl(typ, id, ex) ->
  let idstr = get_id id in
  let _ = check_matrix_decl idstr typ in (* disallow certain bad matrices *)
  let env, s_ex, ex_type = gen_s_expr env ex in
  let _ = match typ, ex_type with
  | A.DataType(T.Int), A.DataType(T.Float) -> ()
  | A.DataType(T.Float), A.DataType(T.Int) -> ()
  | _ -> failwith @@ "Incompatible assignment: " ^ A.str_of_datatype typ ^ " " ^idstr^ " = " ^ A.str_of_datatype ex_type
  in
  let env' = update_env_var env typ id in
  env', S.AssigningDecl(typ, idstr, s_ex)
| A.PrimitiveDecl(typ, id) ->
  let idstr = get_id id in
  let _ = check_matrix_decl idstr typ in (* disallow certain bad matrices *)
  let env' = update_env_var env typ id in
  let env' = S.PrimitiveDecl(typ, idstr)

(* for-loop iterator syntax *)
let gen_s_range env id = function
| A.Range(start_ex, end_ex, step_ex) ->
  let vtype = (get_env_var env id).v_type in
  let idstr = get_id id in
  match vtype with
  | A.NoneType -> failwith @@ "For-iterator " ^idstr^ " undefined"
  | A.DataType(T.Int) | A.DataType(T.Float) -> begin
    let env, s_start_ex, start_type = gen_s_expr env start_ex in
    let env, s_end_ex, end_type = gen_s_expr env end_ex in
    let env, s_step_ex, step_type = gen_s_expr env step_ex in
    match start_type, end_type, step_type with
    | A.DataType(start_raw_typ), A.DataType(end_raw_typ), A.DataType(step_raw_typ) ->
      if not (start_raw_typ = T.Float || start_raw_typ = T.Int) ||
      not (end_raw_typ = T.Float || end_raw_typ = T.Int) ||
      not (step_raw_typ = T.Float || step_raw_typ = T.Int) then
      else
        S.Range(vtype, s_start_ex, s_end_ex, s_step_ex)
    | _ -> failwith @@ "Unsupported range type: " ^ A.str_of_datatype start_type ^ "", " ^ A.str_of_datatype end_type ^ "", " ^ A.str_of_datatype step_type
  end
let gen_s_iter env = function
| A.RangeIterator(typ, id, range) -> ( (* if typ = NoneType, there's no new iterator variable defined in the loop *)
  match typ with
  | A.NoneType ->
    env, S.RangeIterator(typ, get_id id, gen_s_range env id range)
  | _ ->
    (* add the declared var to scope *)
    let env', _ = gen_s_decl env (A.PrimitiveDecl(typ, id)) in
    env', S.RangeIterator(typ, get_id id, gen_s_range env id range)
)

| A.ArrayIterator(typ, id, array_ex) ->
  let idstr = get_id id in
  let env', s_array_ex, array_type = gen_s_expr env array_ex in
  let env', _ = gen_s_decl env (A.PrimitiveDecl(typ, id)) in
  let elem_type = match array_type with
    | A.ArrayType(elem_type) -> elem_type
    | _ -> failwith @@ "Array-style for-loop must operate on array type, not " ^ A.str_of_datatype array_type
  in
  (* check iterator variable and list consistency *)
  let _ = match typ, elem_type with
    | A.DataType(T.Int), A.DataType(T.Float) -> ()
    | typ', elem_type' when typ' = elem_type' -> ()
    | _ -> failwith @@ "For-loop has incompatible types: " ^ A.str_of_datatype typ
      ^" "^idstr^ " but " ^ A.str_of_datatype elem_type ^ " expected"
  in
  env', S.ArrayIterator(typ, idstr, s_array_ex)

(* When if/while/for are followed by a non-compound single-line stmt, *)
(* we need to go one scope deeper *)
let handle_compound_env env = function
  | A.CompoundStatement(_) -> env
  | _ -> incr_env_depth env

(********** Main entry point: AST -> SAST **********)
(* return env, [stmt] *)
let rec gen_sast env = function
  | [] -> (env, [])
  | stmt :: rest ->
    let env_new, s_stmt =
      match stmt with
      (* top level statements *)
      | A.FunctionDecl(return_type, func_id, param_list, stmt_list) ->
        let _ = debug_env env "before FunctionDecl" in
        let s_param_list = gen_s_param_list param_list in
        let fidstr = get_id func_id in
        (* check: mustn't override certain built-in functions *)
        let _ = Builtin.overridable fidstr in
        (* check: main function must have void main() signature *)
        let _ = if fidstr = "main" then
          if List.length param_list > 0
            || return_type <> A.DataType(T.Int) then
            failwith "Main entry function must have signature 'int main()'" in
        in
let env' = incr_env_depth env in
let env' = update_env_func env' return_type func_id param_list true in
let _ = debug_env env' "after FunctionDecl" in
(* get the function declaration, then close 'func_current' *)
let env_after_decl, s_stmt_list = gen_sast env stmt_list in
(* check if properly returned *)
let is_returned = env_after_decl.is_returned in
let _ = if is_returned then ()
else if return_type = A.DataType(T.Void) then ()
else failwith @@ "Function " ^ fidstr
^ " should have at least one return: " ^ A.str_of_datatype return_type in
let function_decl = S.FunctionDecl(return_type, get_id func_id, s_param_list, s_stmt_list) in
let env' = {
  var_table = env.var_table;
  func_table = env'.func_table;
  func_current = "";
  depth = env.depth;
  is_returned = true;
  in_loop = false;
} in
let _ = debug_env env' "closed after FuncDecl" in
env', function_decl

| A.ForwardDecl(return_type, func_id, param_list) ->
  let fidstr = get_id func_id in
  (* check: mustn't override certain built-in functions *)
  let _ = Builtin.overridable fidstr in
  (* check: cannot forward decl main function *)
  let _ = if fidstr = "main" then
    failwith "Cannot forward declare main()"
  in
  let s_param_list = gen_s_param_list param_list in
  let env' = update_env_func env return_type func_id param_list false in
  env', S.ForwardDecl(return_type, fidstr, s_param_list)

(* statements *)
| A.IfStatement(pred_ex, stmt_if, stmt_else) ->
  let env', s_pred_ex, pred_type = gen_s_expr env pred_ex in
  if pred_type = A.DataType(T.Bool) then
    let env_if = handle_compound_env env' stmt_if in
    let env_if, s_stmt_if = gen_sast env_if [stmt_if] in
    let env_else = handle_compound_env env' stmt_else in
    let env_else, s_stmt_else = gen_sast env_else [stmt_else] in
    let env = if env_if.is_returned || env_else.is_returned then set_env_returned env else env in
    env, S.IfStatement(s_pred_ex, List.hd s_stmt_if, List.hd s_stmt_else)
  else
    failwith @@ "If predicate must be bool, but " ^ A.str_of_datatype pred_type ^ " provided"

| A.WhileStatement(pred_ex, stmt) ->
  let env', s_pred_ex, pred_type = gen_s_expr env pred_ex in
  let env' = handle_compound_env env' stmt in
  if pred_type = A.DataType(T.Bool) then
    let env' = { env' with in_loop = true } in
    let env', s_stmt = gen_sast env' [stmt] in
    let env = if env'.is_returned then set_env_returned env else env in
env, S.WhileStatement(s_pred_ex, List.hd s_stmt)
else
  failwith @@ "While predicate must be bool, but " ^ A.str_of_datatype pred_type ^ " provided"

| A.ForStatement(iter, stmt) ->
(* hack: first go one scope deeper, then go back to ensure that *)
(* the iterator variable is in the right scope *)
let env' = incr_env_depth env in
let env', s_iter = gen_s_iter env' iter in
let env' = decr_env_depth env' in
let env' = handle_compound_env env' stmt in
let env', s_stmt = gen_sast env' [stmt] in
let env =
  if env'.is_returned then set_env_returned env else env in
env, S.ForStatement(s_iter, List.hd s_stmt)

| A.CompoundStatement(stmt_list) ->
let env' = incr_env_depth env in
let env', s_stmt_list = gen_sast env' stmt_list in
let env =
  if env'.is_returned then set_env_returned env else env in
env, S.CompoundStatement(s_stmt_list)

| A.Declaration(dec) ->
let env', s_dec = gen_s_decl env dec in
let _ = debug_env env' "after decl" in
env', S.Declaration(s_dec)

| A.Expression(ex) ->
let env', s_ex, _ = gen_s_expr env ex in
env', S.Expression(s_ex)

| A.ReturnStatement(ex) ->
if env.func_current = "" then
  failwith @@ "Invalid return statement outside function definition"
else
  let f_return =
    (get_env_func env (A.Ident(env.func_current))).f_return in
  let _, s_ex, return_type = gen_s_expr env ex in
  let s_ex = match f_return, return_type with
  | A.DataType(T.Int), A.DataType(T.Float) -> s_ex
  | f_return', return_type' when f_return' = return_type' -> s_ex
  | _ -> failwith @@ "Function " ^env.func_current
    ^ " should return " ^ A.str_of_datatype f_return
    ^ ", not " ^ A.str_of_datatype return_type
  in
  let env' = set_env_returned env in
  env', S.ReturnStatement(s_ex)

| A.VoidReturnStatement ->
if env.func_current = "" then
  failwith @@ "Invalid return statement outside function definition"
else
  let f_return =
    (get_env_func env (A.Ident(env.func_current))).f_return in
  if f_return = A.DataType(T.Void) then
    let env' = set_env_returned env in
    env', S.VoidReturnStatement
else
    failwith @@ "Function " ^env.func_current
    ^ " should return " ^ A.str_of_datatype f_return ^ " , not void"

| A.BreakStatement ->
  if env.in_loop then
    env, S.BreakStatement
  else failwith "Invalid break statement outside a loop"

| A.ContinueStatement ->
  if env.in_loop then
    env, S.ContinueStatement
  else failwith "Invalid continue statement outside a loop"

| A.EmptyStatement ->
  env, S.EmptyStatement

| _ -> failwith "INTERNAL unhandled statement"
in
let env_new, s_rest = gen_sast env_new rest in
(env_new, (s_stmt :: s_rest))
module A = Ast
module S = Sast
module T = Type

let header_code =
"#include "qureg.h"
#include "qumat.h"
#include "qugate.h"
#include "quarklang.h"

using namespace Qumat;
using namespace Qugate;

A.7 generator.ml

Primary author: Jim Fan
(* surround with parenthesis *)
let surr str = "(" ^ str ^ ")"

(* subtle differences from Ast print *)
let gen_binop = function
  | A.Mod -> "\%
  | A.Pow -> "pow"
  | A.And -> "\&\&"
  | A.Or -> "\|
  | other -> A.str_of_binop other

let gen_unop = function
  | A.Not -> "!"
  | A.Transpose -> ".transpose()"
  | other -> A.str_of_unop other

let gen_postop = A.str_of_postop

let gen_basictype = function
  | T.Int -> "int64_t"
  | T.Float -> "float"
  | T.Bool -> "bool"
  | T.Fraction -> "Frac"
  | T.Complex -> "std::complex<float>"
  | T.Qreg -> "Qureg"
  | T.String -> "std::string"
  | T.Void -> "void"

let rec gen_datatype = function
  | A.DataType(t) -> gen_basictype t
  | A.ArrayType(t) -> "vector<" ^ gen_datatype t ^ ">"
  | A.MatrixType(t) ->
    match t with
    | A.DataType(matType) ->
      match matType with
      | T.Int | T.Float | T.Complex -> "Matrix<" ^ gen_basictype matType ^ ", Dynamic, Dynamic>"
      | _ -> failwith
        "INTERNAL codegen non-numerical matrix in gen_datatype"
    | _ -> failwith "INTERNAL bad matrix type to str"
  | A.NoneType -> failwith "INTERNAL codegen NoneType in gen_datatype"

(* system temp var (20 char long)*)
(* must take a unit arg to make this a function *)
let gen_temp_var _ =
  let rec seq = function
    | 0 -> []
    | x -> 0 :: seq (x - 1)
  in
  List.fold_left (fun acc _ ->
    (* randomly generate from 3 classes: number, lower/upper case letters *)
    let rand_char =
      match Random.int 3 with
      | 0 ->...
let trim_last str =
  if String.length str > 1 then
    String.sub str 0 ((String.length str) - 2)
  else str

let two_arg func code1 code2 =
  func ^"(" ^ code1 ^", " ^ code2 ^")"

let more_arg_helper left_delimiter right_delimiter func code_list =
  let codes =
    List.fold_left (fun acc code -> acc ^ code ^", ")
    "" code_list in
  let codes = trim_last codes in
  func ^left_delimiter^ codes ^right_delimiter

let array_arg = more_arg_helper
  "{" ^ "}" in

let cast_complex ex_code =
  two_arg "complex<float>" ex_code "0.0"

let cast_fraction ex_code =
  two_arg "Frac" ex_code "1"

let fail_unhandle msg =
  failwith @@ "INTERNAL codegen unhandled " ^ msg

let rec gen_expr = function
  (* simple literals *)
  | S.IntLit(i) -> i
  | S.BoolLit(b) -> b
  | S.FloatLit(f) -> f
  | S.StringLit(s) ->
    gen_basictype T.String ^ "(" ^ s ^ "")"

  (* compound literals *)
  | S.FractionLit(num_ex, denom_ex) ->
    two_arg "Frac" (gen_expr num_ex) (gen_expr denom_ex)

  | S.QRegLit(qex1, qex2) ->
    two_arg "Qureg::create<true>" (gen_expr qex1) (gen_expr qex2)

  | S.ComplexLit(real_ex, im_ex) ->
two_arg (gen_basictype T.Complex) (gen_expr real_ex) (gen_expr im_ex)

| S.ArrayLit(arr_type, ex_list) ->
| array_arg (gen_datatype arr_type) (ex_to_code_list ex_list)

| S.ArrayCtor(arr_type, dim) ->
| gen_datatype arr_type ^"("gen_expr dim ^")"

| S.MatrixLit(elem_type, ex_list, coldim) ->
| (* we flatten the matrix to be a vector *)
| let flattened = gen_expr
| (S.ArrayLit( A.ArrayType(elem_type), ex_list))
| in
| (* a utility function from quarklang.h *)
| two_arg "matrix_literal" (string_of_int coldim) flattened

| S.MatrixCtor(mat_type, rowdim, coldim) ->
| two_arg (gen_datatype mat_type ^ "::Zero")
| (gen_expr rowdim) (gen_expr coldim)

| S.Binop(expr1, op, expr2, optag) ->
| let expr1_code = gen_expr expr1 in
| let expr2_code = gen_expr expr2 in
| (* cast helpers *)
| let parenthize code1 op code2 =
| surr @@ code1 ^ gen_binop op ^ code2
| in
| begin
| match optag with
| | S.OpVerbatim ->
| | if op = A.Pow then (* special: not infix! *)
| | two_arg "pow" expr1_code expr2_code
| | else
| | parenthize expr1_code op expr2_code
| | | S.CastComplex1 ->
| | | parenthize (cast_complex expr1_code) op expr2_code
| | | S.CastComplex2 ->
| | | parenthize expr1_code op (cast_complex expr2_code)
| | | S.CastFraction1 ->
| | | parenthize (cast_fraction expr1_code) op expr2_code
| | | S.CastFraction2 ->
| | | parenthize expr1_code op (cast_fraction expr2_code)
| | | S.OpArrayConcat ->
| | | two_arg "concat_vector" expr1_code expr2_code
| | | S.OpStringConcat ->
| | | parenthize 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"
| | in
| | two_arg equal_func expr1_code expr2_code
| | _ -> fail_unhandle "optag in binop"
| end

| S.Queryop(qreg_ex, op, start_ex, end_ex, optag) ->
| let qreg_code = gen_expr qreg_ex in
| let start_code = gen_expr start_ex in
let end_code = gen_expr end_ex in
let real_flag = if op = A.Query then "true" else "false" in
begin
match optag with
| S.OpQuerySingleBit ->
  more_arg "measure" [qreg_code; start_code; real_flag]
| S.OpVerbatim ->
  more_arg "measure_range" [qreg_code; start_code; end_code; real_flag]
| _ -> fail_unhandle "optag in queryop"
end

/* Unary ops */) | S.Unop(op, ex, optag) ->
  let ex_code = gen_expr ex in
begin
match optag with
| S.OpVerbatim -> surr @@ (gen_unop op) ^ ex_code
  (* blablamat.adjoint() *)
| S.OpMatrixTranspose -> surr @@ (surr ex_code) ^ (gen_unop op)
| _ -> fail_unhandle "optag in unop"
end

| S.Lval(lval) -> begin
match lval with
| S.Variable(id) -> id
| S.ArrayElem(id, ex_list) ->
  let subscripts =
    List.fold_left (fun acc ex ->
      acc ^"["^ gen_expr ex ^"]
    ) "" ex_list
  in
  id ^ subscripts
| S.MatrixElem(id, ex_list) ->
  (* hackish: Eigen lib access matrix elem just like funcall *)
  more_arg id (ex_to_code_list ex_list)
end

/* Post ++ and -- */) | S.PostOp(lval, op) ->
surr @@ gen_expr (S.Lval(lval)) ^^ " gen_postop op

/* Assignment */) | S.Assign(lval, rhs_ex) ->
surr @@ gen_expr (S.Lval(lval)) ^^ " gen_expr rhs_ex

/* Function calls */) | S.FunctionCall(func_id, ex_list, is_matrix_list) ->
  (* handle print specially *)
  if Builtins.is_print func_id then
    if List.length ex_list = 0 then ""
    else
      let delim = " << " in
      let cout_code = List.fold_left2 (fun acc ex is_matrix -> acc ^delim^ gen_expr ex
        ^ (if is_matrix then ".format(QuarkEigenIOFormat)" else "")
      ) "std::cout << std::boolalpha << std::setprecision(6)" ex_list is_matrix_list
      in
      cout_code ^ if func_id = "print" then " << std::endl" else ""
else

if func_id = "apply_oracle" then

let code_list = ex_to_code_list ex_list in

(* de-string arg #2, which is actually a function parameter *)
let arg2 = List.nth code_list 1 in
let str_type_len = String.length (gen_basic_type T.String) in
let arg2 = String.sub arg2
(str_type_len+2) ((String.length arg2) - (str_type_len+4)) in
let code_list = [List.nth code_list 0; arg2; List.nth code_list 2] in
more_arg func_id code_list

else (* non-special cases *)

more_arg func_id (ex_to_code_list ex_list)

(* Membership testing with keyword 'in' *)
| S.Membership(elem, array) ->

(* from quarklang.h *)
two_arg "membership_in" (gen_expr elem) (gen_expr array)

(* python-style tertiary operator *)
| S.Tertiary(true_ex, pred, false_ex, optag) ->

let true_code = gen_expr true_ex in
let true_code = match optag with
| S.CastFraction1 -> cast_fraction true_code
| S.CastComplex1 -> cast_complex true_code
| _ -> true_code
in
let false_code = gen_expr false_ex in
let false_code = match optag with
| S.CastFraction2 -> cast_fraction false_code
| S.CastComplex2 -> cast_complex false_code
| _ -> false_code
in
let pred_code = gen_expr pred in
surr @@ pred_code ^ " ? " ^ true_code ^ " : " ^ false_code
| _ -> fail_unhandle "expr"

(* helper: expr_list -> code(string)_list *)
and ex_to_code_list ex_list =
List.map (fun ex -> gen_expr ex) ex_list

(* Used in A.FunctionDecl *)
let gen_param_list param_list =
List.fold_left
((fun accstr param -> accstr ^
  ((function
    | S.PrimitiveDecl(typ, id) ->
      let type_code = if typ = A.DataType(T.Qreg)
        then "const Qureg&" else gen_datatype typ
      in
      type_code ^ " " ^ id
    | _ -> fail_unhandle "gen_param_list"
  ) param) ^ ", "
) "" param_list

(********** Main entry point: SAST -> string **********)
let rec gen_code = function
  | [] -> ""
  | stmt :: rest ->
let stmt_code =
match stmt with
(* top level statements *)
| S.FunctionDecl(return_type, func_id, param_list, stmt_list) ->
let param_list_code = gen_param_list param_list in
let stmt_list_code = gen_code stmt_list in
let return_type_code =
if func_id = "main" then "int" (* otherwise int64_t *)
else gen_datatype return_type in
return_type_code ^ " " ^ func_id ^ "(" ^ trim_last param_list_code ^ ")\n" ^ "{" ^ stmt_list_code ^ "} // end " ^ func_id ^ "();\n"

| S.ForwardDecl(return_type, func_id, param_list) ->
let param_list_code = gen_param_list param_list in
gen_datatype return_type ^ " " ^ func_id ^ "(" ^ trim_last param_list_code ^ ");\n"

(* statements *)
| S.IfStatement(pred_ex, stmt_if, stmt_else) ->
let code_if = handle_compound stmt_if in
let code_else = handle_compound stmt_else in
"if (" ^ gen_expr pred_ex ^ ")\nelse " ^ code_else ^ 
// end if"

| S.WhileStatement(pred_ex, stmt) ->
let code_while = handle_compound stmt in
"while (" ^ gen_expr pred_ex ^ ")\n" ^ code_while ^ 
// end while"

| S.ForStatement(iter, stmt) ->
let code_for = handle_compound stmt in
begin
match iter with
| S.ArrayIterator(typ, id, array_ex) ->
let array_code = gen_expr array_ex in
"for (" ^ gen_datatype typ ^ " id ^ " array_code ^ ")\n" ^ code_for ^ 
// end for-array"
| S.RangeIterator(typ, id, range) ->
match range with
| S.Range(iter_type, start_ex, end_ex, step_ex) ->
(* pre-store the results in system-reserved temp variables *)
let start_code = gen_expr start_ex in
let end_code = gen_expr end_ex in
let step_code = gen_expr step_ex in
let end_temp = gen_temp_var () in
let step_temp = gen_temp_var () in
(* determines the direction of iteration *)
let dir_temp = gen_temp_var () in
let init = if typ = A.NoneType then id
else gen_datatype typ ^ " id in
let temp_type = gen_datatype iter_type in
(* store step_expr to a temp var *)
temp_type ^ " end_temp ^ " end_code ^ 
(* store step_expr to a temp var *)
temp_type ^ " step_temp ^ " step_code ^ 
^ dir_temp^ " step_temp^ " > 0 ? 1 : -1;\n"
A.8 preprocessor.ml

Primary author: Jim Fan

{  (* default Quarklang source code extension *)
    let extension = "\.qk";;
}

let white = [   ' '    '	'    '
'    'r'   '\r'    '\n'  ]
let nonwhite = \[` ' `\t` `\r` `\n` `;`\]

(* code: string, imports: string[] of file paths *)
rule scan code imports = parse
  (* we shouldn't preprocess anything in a string literal *)
  | ("" ("" _ | ["" ])* "") as s { scan (code ^ s) imports lexbuf }
  (* gets the import file name *)
  | "import" white* ((nonwhite [^';']* nonwhite) as filename) white* ';'
    { let import_file = filename ^ extension in
      scan code (import_file :: imports) lexbuf }
  | "import" white* ';'
    { failwith "Empty import statement" }
  (* supports python-style elif by simple string replacement! *)
  | white+ "elif" white+ { scan (code ^ " else if ") imports lexbuf }
  (* copy anything else verbatim *)
  | _ as c { scan (code ^ String.make 1 c) imports lexbuf }
  (* returns both the processed code and list of imported files *)
  | eof  { (code, List.rev imports) }

(* trailer *)
{
  let read_file_lines filename =
    let lines = ref [] in
    let chan = open_in filename in
    try
      while true; do
        lines := input_line chan :: !lines
      done; []
    with End_of_file ->
      close_in chan;
      List.rev !lines

  let read_file_str filename =
    let lines = ref "" in
    let chan = open_in filename in
    try
      while true; do
        lines := !lines ^ "\n" ^ input_line chan
      done; ""
    with End_of_file ->
      close_in chan;
      !lines

  (* Handle nested imports and circular imports *)
  module StrSet = Set.Make(String)

  (* shaded scan *)
  let scan code =
    scan "" [] (Lexing.from_string code)

  (* preprocessor gets the import relative to the source file itself *)
  (* not where the user invokes the quark compiler *)
  let process filename =
    let filedir = Filename.dirname filename in
    (* return processed_code, new_seen_set *)
    let rec proc_rec filename seen_set =
      try
        let this_code, imports =
          scan (read_file_str filename) in
        let imported_code, seen_set' =

A.9 builtin.ml

Primary author: Jim Fan

(**** list of built-in functions and their interfaces *****)

(* system-reserved temporary variable prefix *)

let forbidden_prefix = "_QUARK_

let wrap basic_type = A.DataType(basic_type)

(* NoneType is a placeholder: len works with any array type *)

let any_array = A.ArrayType(A.NoneType)

(* NoneType is a placeholder, works with any matrix type *)

let any_matrix = A.MatrixType(A.NoneType)

let cx_mat = A.MatrixType(wrap T.Complex)

let i = wrap T.Int
let f = wrap T.Float
let b = wrap T.Bool
let frac = wrap T.Fraction
let cx = wrap T.Complex
let qreg = wrap T.Qreg
let s = wrap T.String
let void = wrap T.Void
let find_builtin = function
| "print" -> [], void
| "print_noline" -> [], void
| "len" -> [any_array], i
| "num" -> [frac], i
| "denom" -> [frac], i
| "real" -> [cx], f
| "imag" -> [cx], f
| "sqrt" -> [f], f
| "rand_int" -> [i; i], i
| "rand_float" -> [f; f], f
| "coldim" -> [any_matrix], i
| "rowdim" -> [any_matrix], i
| "hadamard_mat" -> [i], cx_mat
| "cnot_mat" -> [], cx_mat
| "toffoli_mat" -> [i], cx_mat
| "generic_control_mat" -> [i; cx_mat], cx_mat
| "pauli_X_mat" -> [], cx_mat
| "pauli_Y_mat" -> [], cx_mat
| "pauli_Z_mat" -> [], cx_mat
| "rot_X_mat" -> [f], cx_mat
| "rot_Y_mat" -> [f], cx_mat
| "rot_Z_mat" -> [f], cx_mat
| "phase_scale_mat" -> [f], cx_mat
| "control_phase_shift_mat" -> [f], cx_mat
| "swap_mat" -> [], cx_mat
| "cswap_mat" -> [], cx_mat
| "qft_mat" -> [i], cx_mat
| "grover_diffuse_mat" -> [i], cx_mat
| "qsize" -> [qreg], i
| "qclone" -> [qreg], qreg
| "prefix_prob" -> [qreg; i; i], f
| "apply_oracle" -> [qreg; s; i], void

(* single-bit gates *)
| "hadamard" -> [qreg], void
| "hadamard_top" -> [qreg; i], void
| "pauli_X" -> [qreg; i], void
| "pauli_Y" -> [qreg; i], void
| "pauli_Z" -> [qreg; i], void
| "rot_X" -> [qreg; f; i], void
| "rot_Y" -> [qreg; f; i], void
| "rot_Z" -> [qreg; f; i], void
| "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; A.ArrayType(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; A.ArrayType(i); i], void
| "generic_control" -> [qreg; cx_mat; i; i], void
| "generic_toffoli" -> [qreg; cx_mat; i; i; i], void
| "generic_ncontrol" -> [qreg; cx_mat; A.ArrayType(i); i], void
(* other gates *)
| "swap" -> [qreg; i; i], void
| "cswap" -> [qreg; i; i; i], void
| "qft" -> [qreg; i; i], void
| "grover_diffuse" -> [qreg], void
| _ -> ([], A.NoneType)

(* print is special: it accepts any number of args *)
let is_print = function
  | "print" | "print_noline" -> true
  | _ -> false

(* User not allowed to override certain built-in funtions*)
(* because otherwise will cause trouble in codegen *)
let overridable func =
  match func with
  | "print" | "print_noline"
  | "apply_oracle" ->
    failwith @@ "Built-in function " ^func^ "() not overridable"
  | _ -> ()

A.10 compiler.ml

Primary author: Jim Fan
Secondary author: Jamis Johnson

open Semantic

(*********** Configs ***********)
(* g++ compilation flags *)
let gpp_flags = " -std=c++11 -O3 "

(* where to find the libraries, relative to the executable *)
let relative_lib_path = "../lib"

let ext = Preprocessor.extension (* enforce .qk *)

(* detect OS and select the appropriate quark static/dynamic library *)
let is_win = Sys.os_type <> "Unix"

let quark_shared_lib =
  if is_win then
    [ "libquark.dll" ]
  else
    (* can't really tell if the OS is Mac or Linux *)
    ["libquark.dylib"; "libquark.so"]

let quark_static_lib =
if is_win then
  "quark_win" (* libquark_win.a *)
else
  "quark_unix" (* libquark_unix.a *)

(*********** Main entry of Quark compiler ***********)
let _ =
(* from http://rosettacode.org/wiki/Command-line_arguments#OCaml *)
let srcfile = ref ""
and cppfile = ref ""
and exefile = ref ""
(* User-defined g++ path *)
and gppath = ref "g++"
(* Use static lib or dynamic lib? *)
and is_static = ref false in
let check_src_format src =
  if Sys.file_exists src then
  if Filename.check_suffix src ext then
    srcfile := src
  else
    failwith @@ "Quark source file must have extension " ^ ext
  else
    failwith @@ "Source file doesn't exist: " ^ src
in
let speclist = [
  ("-s", Arg.String(fun src -> check_src_format src),
   ": quark source file"),
  ("-c", Arg.String(fun cpp -> cppfile := cpp),
   ": generated C++ file. If unspecified, print generated code to stdout"),
  ("-o", Arg.String(fun exe -> exefile := exe),
   ": compile to executable. Requires g++ (version >= 4.8)"),
  ("-sco", Arg.String(fun src ->
    check_src_format src;
    let name = Filename.chop_suffix src ext in
    cppfile := name ^ ".cpp";
    exefile := name
  ), ": shorthand for -s <file>.qk -c <file>.cpp -o <file>"),
  ("-sc", Arg.String(fun src ->
    check_src_format src;
    let name = Filename.chop_suffix src ext in
    cppfile := name ^ ".cpp"
  ), ": shorthand for -s <file>.qk -c <file>.cpp"),
  ("-g++", Arg.String(fun gpp -> gppath := gpp),
   ": shorthand for -s <file>.qk -c <file>.cpp"),
  ("-static", Arg.Unit(fun () -> is_static := true),
   ": compile with static lib (otherwise with dynamic lib). Does NOT work on Mac"),
] in
let usage = "usage: quarkc -s source.qk [-c output.cpp ] [-o executable] [-static] [-g++ /path/to/g++]
               in
let _ = Arg.parse speclist
(* handle anonymous args *)
(fun arg -> failwith @@ "Unrecognized arg: " ^ arg)
usage
in
let _ = if !srcfile = "" then
    failwith "Please specify a source file with option -s"

in
(* Preprocessor: handles import and elif *)
let processed_code = Preprocessor.process !srcfile in
(* Scanner: converts processed code to stream of tokens *)
(* let lexbuf = Lexing.from_channel stdin in *)
let lexbuf = Lexing.from_string processed_code in
(* Parser: converts scanned tokens to AST *)
let ast = Parser.top_level Scanner.token lexbuf in
(* Semantic checker: verifies and converts AST to SAST *)
let env = {
    var_table = StrMap.empty;
    func_table = StrMap.empty;
    func_current = ""
    depth = 0;
    is_returned = true;
    in_loop = false;
} in
let _, sast = Semantic.gen_sast env ast in
(* Code generator: converts SAST to C++ code *)
let code = Generator.gen_code sast in
(* Output the generated code *)
let _ = if !cppfile = "" then (* print to stdout *)
    print_endline code
else
    let file_channel = open_out !cppfile in
    output_string file_channel code;
    close_out file_channel
in
(* Compile to binary executable with g++ *)
if !exefile <> "" then
    if !cppfile = "" then
        failwith "Please specify -c <output.cpp> before compiling to executable"
    else
        let lib_folder = Filename.concat
            (Filename.dirname Sys.argv.0) relative_lib_path in
        let lib_path name = Filename.concat lib_folder name in
        let lib_exists name = Sys.file_exists (lib_path name) in
        if Sys.file_exists lib_folder then
            begin
                if not (lib_exists "Eigen") then
                    (* extract from Eigen.tar library *)
                    if lib_exists "Eigen.tar" then
                        let cmd = "tar xzf "
                            ^ lib_path "Eigen.tar" ^ " -C " ^ lib_folder in
                        prerr_endline @@ "Extracting Eigen library from tar: \n"
                        ^ cmd ^ "\n";
                        ignore @@ Sys.command cmd
                    else
                        failwith "Neither lib/Eigen/ nor lib/Eigen.tar found"
                ;
                (* Invokes g++ *)
                (* static lib works on cygwin but not Mac *)
                let cmd = if !is_static then
                    !gppath ^ gpp_flags ^ " -I " ^ lib_folder
                    ^ " -static " ^ !cppfile ^ " -L " ^ lib_folder
                    ^ " -l " ^ quark_static_lib ^ " -o " ^ !exefile
                else
                    let _ = (* copy required shared libs to the exefile folder *)
                    List.map (}
fun libfile ->
  let cpcmd = "cp " ^ lib_path libfile ^ " " ^
  Filename.dirname !exefile in
  ignore @@ Sys.command cpcmd
  ) quark_shared_lib in
  !gppath ^ gpp_flags ^ "-I " ^ lib_folder
  ^ " " ^ !cppfile
  ^ " -L " ^ lib_folder
  ^ " -Wl,-rpath,'$ORIGIN'" (*^ Filename.concat (Filename.current_dir_name) lib_folder*)
  ^ " -l" ^ "quark" ^ " -o " ^ !exefile
  in
  prerr_endline @@ "Invoking g++ command: \n" ^ cmd;
  ignore @@ Sys.command cmd;
end
else
  failwith "Library folder ../lib doesn't exist. Cannot compile to executable. "

A.11 testall.sh

Primary authors: Daria Jung, Parthiban Loganathan, Jamis Johnson

#!/bin/bash
COMPILER="quark/quarkc"

cd quark
make clean
make

cd ../

for TESTFILE in tests/*.qk;
do
  echo " TESTING $TESTFILE"

  LEN=$(( $(#TESTFILE)-3))
  CPPOUTNAME="$TESTFILE:0:$LEN".cpp"
  EXECUTABLENAME="$TESTFILE:0:$LEN"
  OUTFILENAME="$TESTFILE:0:$LEN".coutput"
  TESTFILENAME="$TESTFILE:0:$LEN".output"

  ./$COMPILER -s "$TESTFILE" -c "$CPPOUTNAME" -o "$EXECUTABLENAME"
  ./$EXECUTABLENAME > "$OUTFILENAME"
  if (diff "$OUTFILENAME" "$TESTFILENAME")
    then
      echo " ------ SUCCESS ------"
  else
    echo " -------- FAIL --------"
  fi
  rm "$OUTFILENAME" "$CPPOUTNAME" "$EXECUTABLENAME"
done

A.12 addition.qk
```python
def int add: int x, int y
{
    return x + y;
}
def int main:
{
    int x = 7;
    int y = 20;
    print(add(x, y));
    return 0;
}
```

A.13 array.qk

```python
def int main:
{
    int[] arr1 = [0, 1, 2, 3, 4, 5];
    int x;
    for x in [len(arr1)]:
        print(x);
    return 0;
}
```

A.14 complex.qk

```python
def void print_complex: complex x
{
    print(real(x));
    print(imag(x));
}
def int main:
{
    complex x = i(1.3, 2.6);
    print_complex(x);
    return 0;
}
```

A.15 elif.qk
def void test: string x
{
    if x == "Parthi":
        print("Loganathan");
    elif x == "Daria":
        print("Jung");
    elif x == "Jamis":
        print("Johnson");
    elif x == "Jim":
        print("Fan");
}

def int main:
{
    test("Jim");
    test("Jamis");
    test("Daria");
    test("Parthi");

    return 0;
}

A.16 float.qk

def int main:
{
    float x = 3.0;
    float y = 5.6;

    print(x + y);
    print(x - y);
    print(x * y);
    print(x / y);

    return 0;
}

A.17 fraction.qk

def int main:
{
    int x = 3;
    while x > 0:
    {
        fraction frac = 10$3;
        print(frac);
        x--;
    }

    fraction z = 1$2;
A.18 \texttt{gcd.qk}

```python
def \texttt{int testgcd: int x, int y}
{
    \texttt{while y != 0:}
    \{\texttt{
        int r = x mod y;}
        x = y;
        y = r;
    \}
    \texttt{return x;}
}
def \texttt{int main:}
{
    \texttt{print(testgcd(10, 20));}
    \texttt{return 0;}
}
```

A.19 \texttt{hello\_world.qk}

```python
def \texttt{int main:}
{
    \texttt{print("hello world");}
    \texttt{return 0;}
}
```

A.20 \texttt{import.qk}

```python
\texttt{import /imports/test;}
def \texttt{int main:}
{
    \texttt{int y = test(5);}  
    \texttt{print(y);}  
    \texttt{return y;}
}
```
where /imports/tests.qk is

```python
def int test: int x
{
    return x + 1;
}
```

A.21 logic.qk

```python
def int main:
{
    if 3 == 3:
        print("3 == 3");
    if 3 < 5:
        print("3 < 5");
    if 3 != 5:
        print("3 != 5");
    if true == false:
        print("this should not print");
    if true != false:
        print("this should print");
    if 2 >= 2:
        print("2 >= 2");
    if 1 <= 4:
        print("1 <= 4");
    return 0;
}
```

A.22 matrix.qk

```python
def int main:
{
    float[] mat = [1.2, 3.4; 5.6, 7.8];
    print(mat[1, 1]);
    return 0;
}
```

A.23 multi-array.qk
```python
def int main:
{
    int[][] arr = [[1,2,3],[4,5,6]];

    for int x in [0:len(arr[0])]:
        {print(arr[0, x]);
         print(arr[1, x]);
        }

    return 0;
}
```

A.24 range.qk

```python
def int main:
{
    int x;
    for x in [0:5]:
        print(x);

    return 0;
}
```

A.25 while.qk

```python
def int main:
{
    int x = 10;

    while x > 0: {
        print(x);
        x = x - 1;
    }

    return 0;
}
```

A.26 grover.qk

```python
int nbit = 7;
int key = 73; % secret key

def int grover_oracle : int x
{
    if x == key:
```
```python
return 1;
else
    return 0;
}

def int main:
{
    qreg q = <| nbit+1, 1 |>

    int N = 1 << nbit;
    int sqrtN = sqrt(N);

    hadamard(q);

    int ans = 0;
    float[] probAtKey;

    for int iter in [: sqrtN + 1] :
    {
        apply_oracle(q, "grover_oracle", nbit);

        hadamard_top(q, nbit);
        grover_diffuse(q);
        hadamard_top(q, nbit);

        if iter == sqrtN:
            while grover_oracle(ans) == 0:
                ans = q ?' [:nbit];

            probAtKey &= [prefix_prob(q, nbit, key)];
    }

    print("Found key: ", ans);
    print("Probability = ", probAtKey);
    return 0;
}

A.27 shor.qk

int M = 221;

def int gcd: int a, int b
{
    int c;
    while a != 0:
    {
        c = a;
        a = b mod a;
        b = c;
    }
    return b;
}

def int exp_mod: int b, int e, int m
{
    int remainder;
```
```python
int x = 1;

while e != 0:
    {rem
        remainder = e mod 2;
        e = e >> 1;

        if remainder == 1:
            x = (x * b) mod m;
            b = (b * b) mod m;
    }
return x;
}

def smallest_period: int b, int M
{
    int i = 1;
    while exp_mod(b, i, M) != 1:
        i ++;
return i;
}

def long_pow: int a, int p
{
    if p == 1:
        return a;
    int partial = long_pow(a, p / 2);
    if p mod 2 == 1:
        return partial * partial * a;
    else
        return partial * partial;
}

def log2_int: int x
{
    int ans = 0;
    while x > 0:
    {x = x >> 1;
     ans ++;
    }
return ans;
}
%
  If size == 0, continue until 0
%
def to continuation froce: fraction frac, int size
{int[] cfrac;
int i = 0;
while size < 1 or i < size:
    {a
        % array concatenation
        cfrac &= [num(frac) / denom(frac)];
        frac -= cfrac[i];
        if num(frac) == 0 : break;

        % denom/num built-in
        frac = ~frac;
    i ++;
```
def fraction_to_fraction: int[] cfrac, int size
{
    if size < 1:
        size = len(cfrac);
        fraction ans = 1$cfrac[size - 1];
    for int i in [size-2 : 0 :-1] :
    {
        ans += cfrac[i];
        ans = -ans;
    }
    return ans + cfrac[0];
}

int nbit = log2_int(M) + 1;

% This is the user defined function that should be passed as a string argument
def shor_oracle: int x
{
    return exp_mod(nbit, x, M);
}

def main:
{
    qreg q0 = <| nbit * 2, 0 |>;
    qft(q0, 0, nbit);

    int b; int i;
    while true:
    {
        b = rand_int(2, M);
        if gcd(b, M) != 1: continue;
        qreg q = qclone(q0);
        apply_oracle(q, "shor_oracle", nbit);
        qft(q, 0, nbit);
        int mTrial = 0;
        int measured;
        while mTrial < 10:
        {
            mTrial ++;
            measured = q ?' [:nbit];
            if measured != 0:
            {
                int[] cfrac = to_continued_fraction((1 << nbit)$measured, 0);
                for int size in [len(cfrac):0:-1] :
                {
                    int p = num(to_fraction(cfrac, size));
                    int P = p;
                    while P < 128 and P < M :
                    {
                        }
if $P \mod 2 = 0$ and $\exp \mod (b, P, M) = 1$:
{
    int check = $\exp \mod (b, P / 2, M)$;
    if check != 1 and check != $M - 1$:
    {
        int b_P_1 = long_pow(b, P / 2) - 1;
        int prime = gcd(M, b_P_1);

        if prime != 1 and prime != -1:
        {
            print("Found period r = ", P);
            print("b ^ r = ", b, " ^ ", P, " = 1 mod ", M);
            print("b ^ (r/2) = ", b, " ^ ", P / 2, " = ", check, " mod ", M);
            int prime2 = gcd(M, b_P_1 + 2);
            print("gcd(M, b_P_1 + 2) = ", prime);
            print("gcd(M, b_P_1 + 2) = ", prime2);
            print("\nFactorize ", M, " = ", prime, " * ", M/prime if prime2==1 or prime2==1 else prime2);
            return 0;
        }
    }
    P += p;
}

A.28 qreg.qk

def int main:
{
    % nonsense, but just to test if query works
    qreg ans = <[4, 0]>;

    print(ans ? [:3]);
    print(ans ? '[:3]);

    return 0;
}