QUark Language Reference Manual

Daria Jung (djj2115), Jamis Johnson (jmj2180), Jim Fan (lf2422), Parthiban Loganathan (pl2487)

Introduction

This is the reference manual for QUark, a high level language for quantum computing.

Lexical Conventions

Comments

Single line comments are denoted using a % while multi-line comments use %{   }%. Anything between the brackets will be commented out.

Identifiers

Identifiers are made up of alphabetical characters, numbers, underscores, and the first character cannot be a number. Identifiers are case sensitive.

Keywords

The following identifiers are reserved:

- qreg
- num
- complex
- frac
- bool
- str
- if
- elif
- else
- while
- return
- for
- in
- len
- bit
- and
- or
- null
- import
- mod

Constants

Number Constants

Numbers are represented as either a sequence of digits or an integer part, a decimal point, a fraction part, and an optionally-signed exponent part which consists of an 'e' and a sequence of integers. If the decimal point and the exponent part are included then the fraction part is necessary. All numbers are considered as floats and will be compiled down to c++’s 8-byte, double precision type.

String Constants

Strings can one or more string constants enclosed in double or single quotes. Individual string constants can be alphabetical characters - both lower and upper case - and special reserved escape sequences which are composed of a backslash \ followed by an alphabetical character. The following escape sequences are defined:

- \
- \
- \
- \\

Syntax Notation

In this definition we will use **bold** to define literals and *italics* for categories. We use Backus-Naur Form to specify the grammar.

Types

type-specifier ::= primitive-type | array-type | function-type | null

Identifiers have an associated type and the null type has no value.

Primitive Types

primitive-type ::= number-type | fraction-type | complex-type | quantum-register-type | boolean-type | string-type

Number Type

Numbers are denoted using the following the literal **num**

All numbers will be compiled to c++ doubles.

Fraction Type

Fractions are given by the following literal *frac* and can be constructed using the syntax

fraction-type ::= number-type $ number-type

Complex Type

*complex* is the literal used to denote the complex type and is composed of numbers having the form:

complex-type ::= number +/- number i

The real and imaginary parts can be accessed using `re` and `im`.

Quantum Register Type

There are two quantum register types: sparse and dense. The bracket literals, `< and >` are used to denote a quantum register and an optional apostrophe suffix, `'` means the quantum register is treated as sparse.

quantum-register-type ::= \| |'

The first number is the size of the quantum register and the right number is the initial state.

Boolean Type

Booleans use the literal *bool* and can take the value of the literals *true* or *false*.

String Type
We use the `str` literal to indicate a string type, and strings are sequential alphabetic characters or escape sequences wrapped in single or double quotes.

List Type

\[
\text{list-type} ::= \{\text{primitive-type}\}
\]

Function Type

Functions accept zero or more variables and return a primitive type or list type.

Expressions

\[
\text{expression} ::= \text{base-expression} | \text{multiplicative-expression} | \text{additive-expression} | \text{relational-expression} | \text{equality-expression} | \text{logical-expression} | \text{assignment} | \text{function-call}
\]

Base Expression

\[
\text{base-expression} ::= \text{identifier} | \text{constant} | (\text{expression})
\]

Multiplicative Expression

\[
\text{multiplicative-expression} ::= \text{expression} \times \text{expression} | \text{expression} / \text{expression} | \text{expression} \text{mod} \text{expression}^*
\]

Additive Expression

\[
\text{additive-expression} ::= \text{expression} + \text{expression} | \text{expression} - \text{expression}
\]

Relational Expression

\[
\text{relational-expression} ::= \text{expression} > \text{expression} | \text{expression} < \text{expression} | \text{expression} <= \text{expression} | \text{expression} >= \text{expression}
\]

Equality Expression

\[
\text{equality-expression} ::= \text{expression} == \text{expression} | \text{expression} != \text{expression}
\]

Logical Expression

\[
\text{logical-expression} ::= \text{expression} \text{and} \text{expression} | \text{expression} \text{or} \text{expression}
\]

Assignment

\[
\text{assignment} ::= \text{identifier} \text{type} = \text{expression}
\]

Assignments are right associative and therefore can be chained together such as: `alice = bob = "missing"

Functions

\[
\text{function-call} ::= \text{identifier} (\text{argument-list}) \text{argument-list} ::= \text{argument-list}, \text{expression} | \text{expression}
\]

Expressions are evaluated before passed into the function and all parameters are pass by-value.

Declarations
declaration ::= primitive-declaration | array-declaration | function-declaration

**Primitive Type Declarations**

primitive-declaration ::= identifier primitive-type-specifier | identifier primitive-type-specifier = expression

**Array Type Declarations**

array-declaration ::= identifier [primitive-type-specifier] | identifier [primitive-type-specifier] = [index-list] index-list ::= index-list, expression | expression

**Function Type Declarations**

function-call ::= def identifier return-type ( parameter-list ) statement-block parameter-list ::= param, parameter-list | param | ϵ

**Statements**

statement ::= expression | declaration | statement-block | selection-statement | iteration-statement | return-statement

**Blocks**

statement-block ::= { statement-list } statement-list ::= statement, statement-list | ϵ

**Selection Statements**

selection-statement ::= if ( expression ) statement else statement | if ( expression ) statement

You can nest if statements by writing else if ( expression ) statement.

**Selection Statements**

return-statement ::= return statement

**Iteration Statements**

iteration-statement ::= while ( expression ) statement | for ( iterator ) statement iterator ::= identifier in array-expression | identifier in range range ::= expression : expression : expression | expression : expression

**Import Statements**

import-statement ::= import string-literal

**Grammar**

top-level ::=

top-level-statement top-level

top-level-statement

top-level-statement ::=
datatype identifier (param-list) {statement-block}

datatype identifier (param-list)

declaration

import-statement

statement-block ::= 

statement statement-block

ε

import-statement ::= import string-literal

datatype ::= number / frac / complex / qreg / bool / string / null

expression ::= 

expression + expression

expression - expression

expression* expression

expression / expression

expression mod expression

expression < expression

expression <= expression

expression > expression

expression >= expression

expression == expression

expression != expression

expression or expression

expression and expression

(expression)

constant

{expression-list}

identifier ()

identifier (expression-list)
expression-list ::= 
expression , expression-list 
expression 
declaration ::= 
identifier = expression 
datatype identifier 
datatype [identifier] 
statement ::= 
if (expression) statement else statement 
if (expression) statement 
while ( expression ) statement 
for ( iterator ) statement 
{ statement-block } 
expression 
declaration 
return expression 
return 
iterator ::= 
identifier in range 
identifier in expression 
range ::= 
expression : expression : expression 
expression : expression 
param ::= 
datatype identifier 
datatype [identifier] 
param-list ::= 
param, param-list
param

\epsilon

\textit{constant} ::= \textit{number} \mid \textit{frac} \mid \textit{complex} \mid \textit{qreg} \mid \textit{bool} \mid \textit{string} \mid \textit{null}