A Programming Language For Quantum Oracle Construction

Ayush Tambde

August 2021

1 Abstract

Quantum programs consist of instructions for classical and quantum devices to perform computational tasks. Many of these tasks will require operations such as addition, subtraction and certain logical operations. However, such operations can be tedious to code in today's gate-based environment. To solve this problem the author designed a programming language named *Q* for use on quantum computers where the main objective was to ease the construction of a circuit to perform these operations. The compiler translates high-level code written in *Q* and converts it into *OpenQASM*[1], a gate-based assembly language that runs on IBM Quantum Systems and compatible simulators.

2 Introduction

Quantum Computers were first conceptualized by the late physicist Richard Feynman[2] in a landmark paper in which he observed that classical techniques are inefficient when simulating quantum mechanics. Since the memory required to store a quantum state increases exponentially with the size of the system, Feynman proposed a new form of computer which would be capable of processing quantum information, hence the term *Quantum Computer*, in which the fundamental unit of information is a two-level system known as a qubit. Quantum gates are applied to qubits to in order to change their state[3].

Unfortunately current quantum frameworks make it tedious to implement classical functions in the form of quantum circuits. This was the primary motivation behind Q where tasks such as addition, subtraction and multiplication on quantum states as well as relational operations can be performed in a user-friendly manner. This is aided by the Q compiler's resource estimator which heuristically computes the total number of qubits required to run a program before run-time and therefore allows dynamic resizing of registers at compile-time. Q has a particular focus on simplicity and expressiveness − the compiler generates OpenQASM, thus the output can be used in IBM's QISKit framework [1].

3 The Q Programming Language

A standard Q program consists of two main parts organized within subroutines:

- Declaration of data
- Program statements to manipulate data

Statements are terminated with a semicolon.
There are two types of data in Q:

- *Integer (int)*
- *Quantum-Integer (super)*

These two types distinguish between data allocated on two different devices, namely classical and quantum data.

Data is assigned through a *variable declaration.* Variables are represented by identifiers. The values of both data-types are denoted by integers. However, for quantum data, the integer n represents the upper-bound of a uniform superposition, which ranges from $\{0 : n - 1\}$. This provides a useful method for generating superposition states.

Expressions in Q can be of a quantum nature or classical, and are constructed using a selection of operators (infix notation) similar to C. The operators come under two specific groups:

- Arithmetic operators are addition, subtraction and multiplication
- Relational operators are for testing equality (==, !=) and order (>, <, >=, <=)

3.1 Functions and Oracles

Subroutines are *functions* that have the option to return a value or *oracles* which, unlike functions, are first-class objects − they return a memory address pointing to the location of the oracle in classical memory. Since all quantum operations are unitary and therefore reversible, the bodies of all subroutines in Q are expanded inline.

Functions are declared using the *function* keyword. If a function returns a value, it is specified by preceding the keyword *function* with the type. This is followed by a function identifier followed by a list of parameters enclosed in parentheses. The standard form for declaring parameters is $(< type, identifier >, ...)$. Finally, the function's body is enclosed in opening and closing braces$\{<$ *body* $>\}$. All parameters are passed-by-reference due to the no-cloning theorem which states that a quantum state cannot be cloned in its entirety [7].

The return statement saves whichever variable is returned from being uncomputed and the corresponding qubit register is returned and can be assigned a new identifier. The syntax for the return statement is the *return* keyword, followed by the variable one wishes to return.

Although they have no return value, oracles are declared in a similar manner using the keyword *oracle* instead of the keyword *function.* The primary reason for oracles is the need to pass functions as parameters to other functions, this is because low-level memory manipulation is not supported and function-pointers cannot be used. Instead, oracles were introduced into the language for this task. When they are passed as input, a structure is passed which contains their address in memory amongst other pieces of information. The reason they are given the term "oracle" is due to their presence in quantum search. Since there are a certain number of iterations that need to be performed, the intrinsic quantum search function takes as input a single oracle and iterates using the provided oracle and the diffusion operator.

Figure 1: Functions and Oracles in Q

```
# This is a comment

super function some_function() {
   # do stuff
    # return stuff
}

oracle some_oracle() {
   # do stuff
}

function main() {
   # do stuff
}
```

3.2 Type System

Variable declarations in Q must begin with a type specifier, succeeded by a variable identifier. The type system is designed to indicate where a variable should be allocated. A variable of type "int" declares an integer on a classical computer, whereas a declaration introduced with the keyword "super" allocates qubits on a quantum computer.

The choice of using the "super" keyword for quantum variables is purely to provide a shorthand way to declare uniform superpositions. Values are assigned to either type of variable using the operator "=" as in C.

There is a limitation on the values that can be assigned to a quantum variable $-$ it must be a power of 2, ie any quantum variable initialized with a value x must satisfy $log_2(x) \in N$. This limitation allows us to create a uniform superposition: $\sum_{i=0}^{x-1} \frac{1}{\sqrt{x}} |i_{10}\rangle$. If one measures this state it will collapse into a basis state $|i\rangle$ with probability $(\frac{1}{\sqrt{x}})^2 = \frac{1}{x}$. Reassignment of quantum variables is not allowed.

Figure 2: Type System in Q

```
function main() {
   super a = 4;
    int b = 2;
}
```

3.3 Operations

Operations can be performed on both quantum and classical data. The compiler maintains a resource estimator at compile-time during which it tracks each operation and dynamically resizes quantum registers as required. This is particularly useful when performing arithmetic operations since they can alter the size of a register from n qubits to m qubits, and saves the programmer from having to perform such tasks manually.

3.4 Conditional Expressions

Q supports the if-else model that permeates throughout most modern programming languages.

Conditional expressions can be based on classical or quantum test conditions, but not both. When they are quantum, they introduce entanglement.

To declare an if-statement, one must use the if keyword, followed by a test condition enclosed in parentheses, succeeded by the conditional-body enclosed within braces.

The syntax is similar for an elsif statement, which must succeed an if-statement or an elsif-statement. The only difference is instead of using the if keyword, one must use the $elsif$ keyword.

An else-statement signals the default case. It must succeed an if-statement or an elsif-statement. To declare an else-statement, one must use the *else* keyword, followed by the body of the else-statement enclosed within braces.

Figure 3: Conditionals in Q

```
function main() {
   if(some_condition) {
      # do stuff
    }

    elsif(some_other_condition) {
      # do other stuff
    }

    else {
      # do something else
    }
}
```

3.5 Loops

Loops are treated as classical constructs within Q by expanding them inline during compile time. There are two forms of loops in the language:

- For loop
- While loop

3.5.1 For Loop

The syntax for declaring a for loop is to use the keyword “for” followed by a series of three expressions enclosed in parentheses and separated by commas.

The first expression should declare or initialize any classical data to be used in the loop. The second should specify the halt condition, that is, the circumstances required for the loop to terminate. As in C, the third condition specifies the modifications to be made to data on each iteration. After these three conditions the loop body is specified within braces.

3.5.2 While Loop

The syntax for declaring a while loop is to use the keyword “while” followed by a single condition enclosed in parentheses. The condition specifies the circumstances required for the loop to run. After this statement the loop body is specified within braces.

Figure 4: Loops in Q

```
function main() {
      # FOR
   for(int i = 0; i < 5; i+=1) {
      # Do stuff
    }

    # WHILE
    while(true) {
      # Do stuff
    }
}
```

3.6 Assembly Instructions

Q supports basic quantum assembly instructions.

Figure 5: Q: Quantum Gates

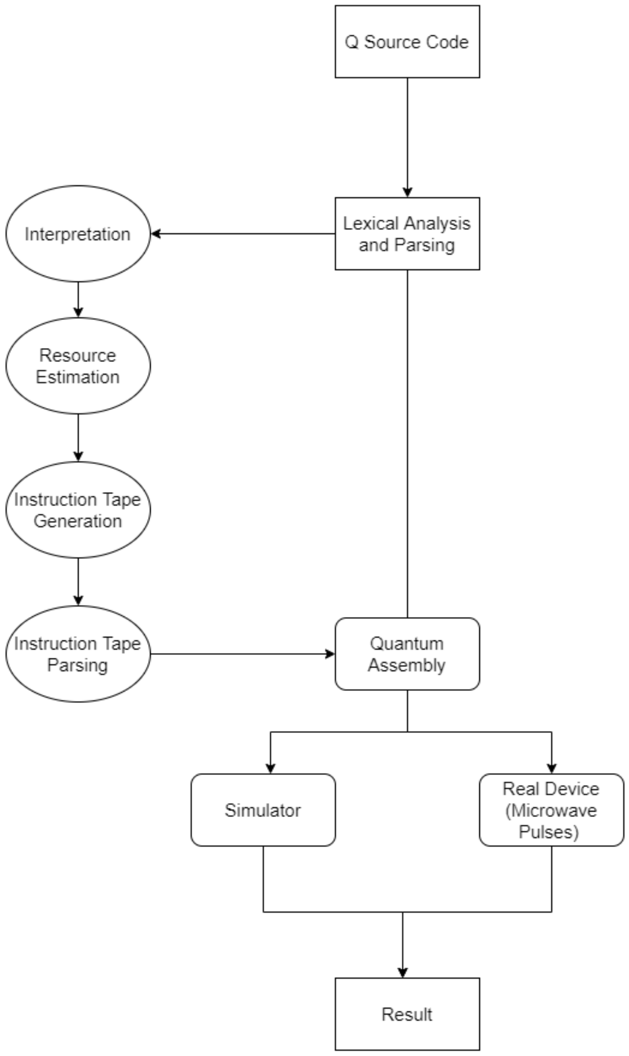
```
function main() {
   # Hadamard gate
   H(foo);
    # X/NOT gate
    X(foo);
    # Y gate
    Y(foo);
    # Z gate
    Z(foo);
    # Rotate X gate
    RX(foo, angle);
    # Rotate Z gate
    RZ(foo, angle);
    # Rotate Y gate
    RY(foo, angle);
    # Apply phase
    P(foo, angle);
    # S gate
    S(foo);
    # T gate
    T(foo);
    # Controlled-Not gate
    CX(foo, bar);
    # Controlled-Z gate
    CZ(foo, bar);
    # Controlled phase
    CP(foo, bar, angle);

}
```

3.7 Compiler Details

The Q compiler makes use of mechanisms such as register-size tracking in order to perform mathematical operations. Furthermore, ancillary registers used in such operations are dealt with internally and are uncomputed when not required and reset to zero automatically by maintaining an internal instruction tape intermediate program representation. They are referenced by the compiler as ancillaX where X is the number of ancillary registers decremented by one. Likewise, cmpX registers are used for storing results of relational operations. Purely classical expressions are evaluated at compile-time, leaving only quantum code to be compiled for later execution.

Figure 6: Q Compiler Structure Flowchart



3.7.1 Addition and Subtraction

The addition operator in Q is based on the Quantum Fourier Transform (QFT) [3] and is implemented as an optimized version of the method proposed by Draper[4]. The algorithm to perform $r = x + y$, is executed as followed:

1. Initialize two registers, x and y
2. Initialize a register r to $|0\rangle^{\otimes n}$, where $n = floor(log_2(x+y)) + 1$
3. Apply $H^{\otimes n}$ to r to obtain the state:
$$(\frac{|0\rangle + |1\rangle}{\sqrt{2}})^{\otimes n} = \frac{1}{\sqrt{2^n}}(|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) \otimes ...(|0\rangle + |1\rangle)$$
4. Apply a series of controlled phase operations to store the Fourier Transform of x in r:
$$\frac{1}{\sqrt{2^n}}(|0\rangle + e^{2\pi i 0.x_n} |1\rangle) \otimes (|0\rangle + e^{2\pi i 0.x_{n-1}...x_n} |1\rangle) \otimes ...(|0\rangle + e^{2\pi i 0.x_1 x_2 ... x_n} |1\rangle)$$
5. Apply a series of controlled phase operations to add the Fourier Transform of y into r. r now holds the sum of x and y in the Fourier Basis.
$$\frac{1}{\sqrt{2^n}}(|0\rangle + e^{2\pi i (0.x_n + 0.y_n)} |1\rangle) \otimes (|0\rangle + e^{2\pi i (0.x_{n-1}...x_n + 0.y_{n-1}...y_n)} |1\rangle) \otimes ...(|0\rangle + e^{2\pi i (0.x_1 x_2 ... x_n + 0.y_1 y_2 ... y_n)} |1\rangle)$$
$= \text{QFT}|\text{x + y}\rangle$
6. Apply $QFT^{\dagger}$ to $result$ to retrieve $|x+y\rangle$ in the computational basis:
$$QFT^{\dagger} QFT\, |x+y\rangle = |x+y\rangle$$

Note: To perform subtraction, the controlled operations in step 5 are inverted

3.7.2 Multiplication

The multiplication operator in Q is similarly based on the QFT.

Given two integers, k, l, the algorithm computes their product j:

$k \cdot l \rightarrow j$ The algorithm is as follows:

(a) Initialize registers k as multiplicand, l as multiplier and j as $|0\rangle^n$ where n is the size of the output in bits and defaults to $n = sizeof(k) + sizeof(l)$

(b) Apply $H^{\otimes n}$ to j, resulting in the state:

$$(\frac{|0\rangle + |1\rangle}{\sqrt{2}})^{\otimes n}) = \frac{1}{\sqrt{2^n}}(|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) \otimes ...(|0\rangle + |1\rangle)$$

The goal is to transform the state described above to the Fourier Transform of $k \cdot l = j$

(c) QFT$|j\rangle = \frac{1}{\sqrt{2^n}}(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle) \otimes (|0\rangle + e^{2\pi i 0.j_2 \ldots j_n}|1\rangle) \otimes ...(|0\rangle + e^{2\pi i 0.j_n}|1\rangle)$, therefore it is desirable to obtain the relative phase factor $e^{2\pi i 0.j_1 j_2 \ldots j_n}$.

This can be achieved through multiplying the binary fractional forms of the multiplier and multiplicand: $(0.k)(0.l) = 0.kl = (\frac{k_1}{2^1} + \frac{k_2}{2^2} ... \frac{k_n}{2^n}) \cdot (\frac{l_1}{2^1} + \frac{l_2}{2^2} ... \frac{l_n}{2^n}) =$

$$\frac{k_1 l_1}{2^2} + \frac{k_1 l_2}{2^3} \cdots \frac{k_1 l_n}{2^{n+1}} +$$
$$\frac{k_2 l_1}{2^3} + \frac{k_2 l_2}{2^4} \cdots \frac{k_2 l_n}{2^{n+2}} +$$
$$\frac{k_n l_1}{2^{n+1}} + \frac{k_n l_2}{2^{n+2}} \cdots \frac{k_n l_n}{2^{2n}}$$
$$= C$$

This is implemented as multi-controlled phase rotations (P) applied to a single qubit of j, to produce the phase $e^{2\pi i 0.j_1 j_2 \ldots j_n} = e^{2\pi i C}$, where P corresponds to the following matrix:

$\begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i \theta} \end{bmatrix}$ and θ is equal to the $\frac{1}{2^x}$ phase angles.

(d) Apply relative phases $e^{2\pi i 2^m C}$ to each qubit j_i, where m is the size of the output

(e) Apply $QFT^\dagger$ (Inverse QFT) on j to retrieve the product in the computational basis

4 Examples

4.1 Quantum Search

The quantum search algorithm, first published in 1996 by Lov K. Grover [5], offers a polynomial-time advantage over the best known classical algorithm for searching for an item in a set of unordered data.

The algorithm is as follows:

(a) Initialize a superposition of the search space

(b) Apply an oracle, O, on the superposition. The oracle is designed in such a way that it performs an operation that applies a phase of π if a term in the superposition fits a specified search constraint.

(c) Apply diffusion operator, i.e., $|s\rangle\langle s| - I$ where $|s\rangle$ is the initial search space.

(d) Repeat steps 1-3 $\frac{\pi}{4}\sqrt{N}$ times where N is the size of the search space.

(e) Measure result.

The following is an application of the quantum search algorithm to search for all x where $4x < 4$ and $0 \leq x \leq 7$. The only solution is the state 0, and the algorithm should discover it with high probability.

The first figure depicts the algorithm written in the author's Q Programming Language, and the second figure performs the same task yet written in IBM's QISKit.

Figure 7: Example Q program for Quantum Search Algorithm

```
# oracle takes a superposition as an input
oracle some_oracle(super var) {
   # check if the input variable multiplied by four is less than four
   if(var * 4 < 4) {
      # apply a phase of pi to var if var * 4 < 4
      mark(var,pi);
   }
}


function main() {
   #declare a uniform superposition of 3 qubits
   super variable = 8;
   # "filter" function is diffusion operator
   filter(some_oracle(variable));
   measure variable;
}
```

Figure 8: Corresponding QISKit Code

```
# import libraries
from qiskit.circuit.library.arithmetic import IntegerComparator
from qiskit.circuit.library import QFT, GroverOperator
from qiskit.visualization import plot_histogram
from qiskit import *
from math import pi
# create and initialize registers
input_reg = QuantumRegister(3,name="input")
output_reg = QuantumRegister(5, name="output")
qc1 = QuantumCircuit(input_reg, output_reg)
qc1.h(input_reg)
qc = QuantumCircuit(input_reg, output_reg
# set up multiplication circuit based on QFT
phase = pi*1/2**2*4
phase_copy = phase*2
for i in range(5):
    qc.h(output_reg[i])
    for j in range(3):
        if i >= 3 and j == 0 or (j==1 and i==4):
            phase /= (2**1)
            continue
        qc.cp(phase,input_reg[j], output_reg[i])
        phase /= (2 ** 1)

    phase = phase_copy
    phase_copy *= 2

 # apply inverse QFT
qft_circ = QFT(num_qubits=5).inverse()
qc.compose(qft_circ, qubits=[3,4,5,6,7], inplace=True)
qc1.compose(qc, qubits=range(8), inplace=True)
# compare with 4 (less than)
comparison_circ = IntegerComparator(5, 4, geq=False,name="comparison_circ")
circ_final = QuantumCircuit(15,3)
circ_final.compose(qc1, inplace=True)
circ_final.compose(comparison_circ, qubits=range(3,13), inplace=True)
# apply phase
circ_final.z(8)
# uncomputation
circ_final.compose(comparison_circ.inverse().to_gate(), qubits=range(3,13),
inplace=True)
circ_final.compose(qc.inverse().to_gate(), qubits=range(8), inplace=True)
# diffusion operator applied once
circ_final.h(range(3))
circ_final.x(range(3))
circ_final.mct([0,1,2], 13, 14, mode="basic")
circ_final.x(range(3))
circ_final.h(range(3))
# measurement
circ_final.measure(range(3), range(3))
```

4.2 Deutsch-Josza Algorithm

The algorithm was first proposed by Deutsch [6] in 1985 and later elaborated on in 1992 by Deutsch and Josza.

Given a function f, this algorithm checks if it is constant or balanced (guaranteed to be either).
If f is constant, f returns either 0 or 1 for all inputs.
If f is balanced,f returns 0 for half of all inputs and 1 for the other half.

The algorithm proceeds as follows:

(a) Initialize state $|s\rangle$ as a superposition: $\frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |i\rangle$.

(b) Apply f to $|s\rangle$ and XOR the result with a qubit in the $|-\rangle$ state.

(c) Apply Hadamard Gate (H), on $|s\rangle$.

(d) Measure $|s\rangle$.

(e) If $|s\rangle$ is measured to be zero, then f is constant. Else f is balanced.

The following is an application of the Deutsch-Josza algorithm for a function f whereby $f(x) = 1$ if $x + 7 > 14$ and $f(x) = 0$ if $x + 7 \leq 14$, and $0 \leq x \leq 15$. The function is balanced and therefore the algorithm should return a non-zero integer. In fact, it should return the bit-string 1000_2.

Figure 9: Example Q program for the Deutsch-Josza Algorithm

```
# Q program illustrating Deutsch-Josza Algorithm


function deutsch_josza(super inputs) {
   # check if state + 7 > 14 and store result in
   # 1-qubit register (called cmp0 internally)
   if(inputs + 7 > 14) {
      # XOR the contents of cmp0 with a qubit in the state |->
      mark(inputs,pi);
   }
}


function main() {
   # initialize superposition to 4 qubits
   super test = 16;
   deutsch_josza(test);
   # interference with Hadamard Gate.
    H(test);
   # store result in a classical register
    # called creg_test and can now be referenced as such
   measure test;
}
```

Figure 10: Corresponding QISKit Code

```
# import libraries for integer comparisons and addition
from qiskit.circuit.library.arithmetic import WeightedAdder,
IntegerComparator
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

# create circuits for addition and integer comparison
addition_circuit = WeightedAdder(7, [8,4,2,1, 4,2,1], name="adder_circ")
comparison_circuit = IntegerComparator(5, 15, name="comparison_circ")

# create registers and circuits for input and the integer "seven"
input_register = QuantumRegister(4, name="input_register")
seven = QuantumRegister(3, name="seven")
input_reg_to_circ = QuantumCircuit(input_register)
integer_seven_circ = QuantumCircuit(seven)

# apply HADAMARD on input to generate uniform superposition
input_reg_to_circ.h(input_register)
# flip all 3 bits in register to represent "7" in binary
integer_seven_circ.x(seven)
# add summands as input to addition circuit
addition_circuit.compose(qc.to_gate(), qubits=[0,1,2,3], front=True,
inplace=True)
addition_circuit.compose(qc1.to_gate(), qubits=[4,5,6], front=True,
inplace=True)
# create final circuit to hold all circuits
circuit_final = QuantumCircuit(22,4)
# append addition circuit in front of empty circuit
circuit_final.compose(addition_circuit.to_gate(), qubits=range(17),
front=True, inplace=True)
# add comparison circuit after addition circuit with the input
# being the sum from the previous circuit
circuit_final.compose(comparison_circuit.to_gate(),
qubits=[7,8,9,10,11,17,18,19,20,21], inplace=True)
# apply phase
circuit_final.z(17)
# uncomputation of circuits
circuit_final.compose(comparison_circuit.inverse().to_gate(),
qubits=[7,8,9,10,11,17,18,19,20,21], inplace=True)
circuit_final.compose(addition_circuit.inverse().to_gate(),
qubits=range(17), inplace=True)
# measure our input register
circuit_final.measure(range(4), range(4))
```

5 Conclusion

The author designed a new programming language that allows a higher-level description of oracle functions than available in existing frameworks. Although the language compiler can be used as a standalone tool, the author expects it to be used alongside other OpenQASM-based frameworks such as QISKit.

6 Acknowledgements

The author would like to thank Prof. David Abrahamson for his guidance in the creation of this paper, and Prof. Brendan Tangney Computer Science Statistics and a Fellow of Trinity College Dublin, Dr Keith Quille, Lecturer of Computing in the TUDublin, Tallaght Campus and CSIncl, for their insights and their help in the narrative of this paper. I would also like to thank Dr. Lee O'Riordan from the Irish Center For High-End Computing and Dr. Peter Rohde from UTS Australia, for their unique insights and suggestions.

7 Appendices

7.1 Lexical Specification

digit = [0..9]

number = digit+

letter = [a..z] | [A..Z]

identifier = letter (letter | digit | “″)∗

type = “int” | “super”

semicolon = “;”

lparen = “(”

rparen = “)”

lbrace = “{”

rbrace = “}”

assign = “=”

operator = “+” |“-” | “*” | “+=” | “-=” | “*=” | “¡” | “¿” | “¡=” | “¿=” | “==” | “!=”’

keyword = “return” | “measure” | “function”

conditional = “if” | “elsif” | “else”

loop = “for” | “while”

8 References

[1] Andrew W. Cross, Lev. S Bishop, John A. Smolin, Jay M. Gambetta, Open Quantum Assembly Language, 2017. arXiv:1707.03429 [quant-ph]

[2] Richard P Feynman. Simulating physics with computers, 1981. International Journal of Theoretical Physics, 21(6/7)

[3] Nielsen and Chuang. Quantum Computation and Quantum Information, 2000. Cambridge Press.

[4] Thomas G. Draper. Addition on a Quantum Computer, 2000. arXiv:quant-ph/0008033

[5] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996. arXiv:quant-ph/9605043.

[6] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum compuation. Proceedings of Royal Society of London, A439:553–558 (1992).

[7] Wootters, W., Zurek, W. A single quantum cannot be cloned. Nature 299, 802–803 (1982). https://doi.org/10.1038/299802a0