Qwerty: A Basis-Oriented Quantum Programming Language

QCE ’25

Austin J. Adams*, Sharjeel Khan*, Arjun S. Bhamra*, Ryan R. Abusaada*, Travis S. Humble†, Jeffrey S. Young*, Thomas M. Conte*

September 5th, 2025

*Georgia Tech and †ORNL

Quantum Programming is Tough for Newcomers

Two leaps:

Quantum Programming is Tough for Newcomers

Two leaps:

① Problem $\longrightarrow$ Algorithm

Quantum Programming is Tough for Newcomers

Two leaps:

① Problem $\rightarrow$ Algorithm

② Algorithm $\rightarrow$ Implementation

Quantum Programming is Tough for Newcomers

Two leaps:

① Problem $\longrightarrow$ Algorithm

② Algorithm $\longrightarrow$ Implementation

Two leaps:

① Problem $\longrightarrow$ Algorithm

② Algorithm $\longrightarrow$ Implementation

Quantum programming languages

Example operation:

$$|+\rangle|+\rangle|+\rangle|+\rangle \mapsto -|+\rangle|+\rangle|+\rangle|+\rangle$$

Example operation:

$$|+\rangle|+\rangle|+\rangle|+\rangle \;\mapsto\; -|+\rangle|+\rangle|+\rangle|+\rangle$$

How does this look in most quantum programming languages?

Step 1: Do the math

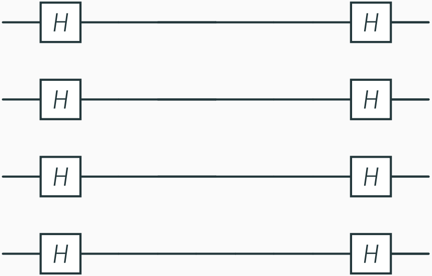
$$
\begin{aligned}
U = & -|{+}{+}{+}{+}{+}\rangle\langle{+}{+}{+}{+}{+}| \\
& + \left(I^{\otimes 4} - |{+}{+}{+}{+}{+}\rangle\langle{+}{+}{+}{+}{+}|\right)
\end{aligned}
$$

Step 2: Design a circuit

$$\begin{aligned} U = & -|{+}{+}{+}{+}\rangle\langle{+}{+}{+}{+}| \\ & + \left(I^{\otimes 4} - |{+}{+}{+}{+}\rangle\langle{+}{+}{+}{+}|\right) \end{aligned}$$

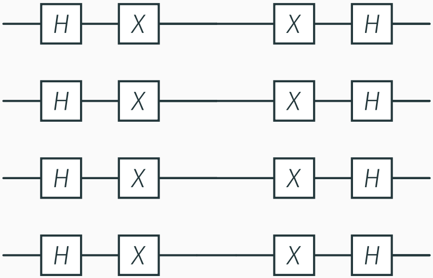
Step 2: Design a circuit

$$U = H^{\otimes 4}(-|0000\rangle\langle 0000| + (I^{\otimes 4} - |0000\rangle\langle 0000|))H^{\otimes 4}$$



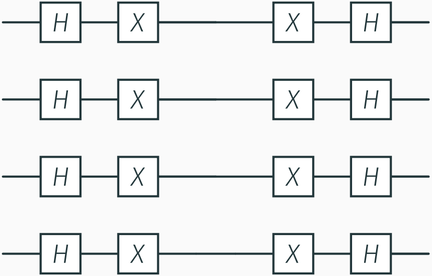
Step 2: Design a circuit

$$U = H^{\otimes 4} X^{\otimes 4} (-\,|1111\rangle\langle 1111| + (I^{\otimes 4} - |1111\rangle\langle 1111|)) X^{\otimes 4} H^{\otimes 4}$$



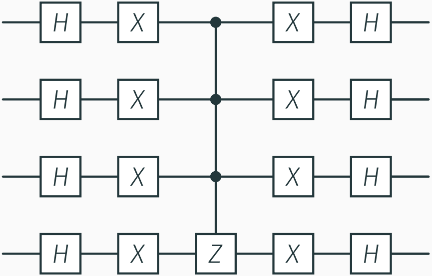
Step 2: Design a circuit

$$U = H^{\otimes 4} X^{\otimes 4} \big(-|1111\rangle\langle 1111| + (I^{\otimes 4} - |1111\rangle\langle 1111|) \big) X^{\otimes 4} H^{\otimes 4}$$



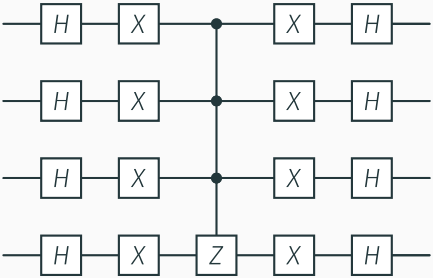
Step 2: Design a circuit

$$U = H^{\otimes 4} X^{\otimes 4} (CCCZ) X^{\otimes 4} H^{\otimes 4}$$



Step 2: Design a circuit

$$U = -|{+}{+}{+}{+}{+}\rangle\langle{+}{+}{+}{+}{+}| + (I^{\otimes 4} - |{+}{+}{+}{+}{+}\rangle\langle{+}{+}{+}{+}{+}|) \quad \checkmark$$



Step 3: Write synthesis code

QCL (2000)

```
operator diffuse(qureg q) {
  H(q);
  Not(q);
  CPhase(pi,q);
  !Not(q);
  !H(q);
}
```

Step 3: Write synthesis code

QCL (2000)

```
operator diffuse(qureg q) {
  H(q);
  Not(q);
  CPhase(pi,q);
  !Not(q);
  !H(q);
}
```

Q# (2025)

```
operation Diffuse(q : Qubit[])
                 : Unit {
  for qi in q {
    H(qi);
    X(qi);
  }
  Controlled Z(Most(q),
               Tail(q));
  for qi in q {
    X(qi);
    H(qi);
  }
}
```

Step 3: Write synthesis code

QCL (2000)

```
operator diffuse(qureg q) {
  H(q);
  Not(q);
  CPhase(pi,q);
  !Not(q);
  !H(q);
}
```

Q# (2025)

```
operation Diffuse(q : Qubit[])
                  : Unit {
  within {
    for qi in q {
      H(qi);
      X(qi);
    }
  } apply {
    Controlled Z(Most(q),
                 Tail(q));
  }
}
```

Step 3: Write synthesis code

QCL (2000)

```
operator diffuse(qureg q) {
  H(q);
  Not(q);
  CPhase(pi,q);
  !Not(q);
  !H(q);
}
```

Q# (2025)

```
operation Diffuse(q : Qubit[])
                 : Unit {
  within {
    ApplyToEachA(H, q);
    ApplyToEachA(X, q);
  } apply {
    Controlled Z(Most(q),
                 Tail(q));
  }
}
```

Step 3: Write synthesis code

QCL (2000)

```
operator diffuse(qureg q) {
  H(q);
  Not(q);
  CPhase(pi,q);
  !Not(q);
  !H(q);
}
```

Q# (2025)

```
operation Diffuse(q : Qubit[])
                 : Unit {
  within {
    ApplyToEachA(H, q);
    ApplyToEachA(X, q);
  } apply {
    Controlled Z(Most(q),
                 Tail(q));
  }
}
```

Most Quantum PLs today require circuit synthesis expertise

Goal:

$$|+\rangle|+\rangle|+\rangle|+\rangle \mapsto -|+\rangle|+\rangle|+\rangle|+\rangle$$

Goal:

$$|+\rangle|+\rangle|+\rangle|+\rangle \mapsto -|+\rangle|+\rangle|+\rangle|+\rangle$$

Qwerty syntax:

```
'pppp' >> -'pppp'
```

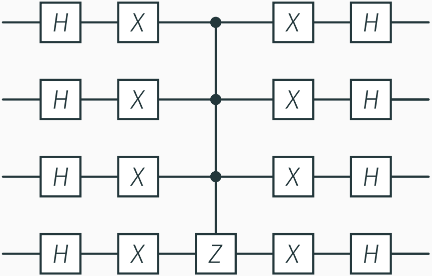
Goal:

$$|+\rangle|+\rangle|+\rangle|+\rangle \mapsto -|+\rangle|+\rangle|+\rangle|+\rangle$$

Qwerty syntax:

```
'pppp' >> -'pppp'
```

Circuit synthesized by ASDF:



Our Contributions

1. Qwerty programs expressed with **basis translations** and **qubit literals**
2. Programmers can **interpret** the behavior of Qwerty programs **without circuit synthesis experience**
3. Qwerty integrates with a popular classical PL (**Python**)

Hello World in Qwerty

```python
from qwerty import *
















print(grover())
```

Hello World in Qwerty

```
from qwerty import *

@classical
def oracle(x: bit[4]) -> bit:
  return x[0] & ~x[1] & x[2] & ~x[3]













print(grover())
```

Hello World in Qwerty

```
from qwerty import *

@classical
def oracle(x: bit[4]) -> bit:
  return x[0] & ~x[1] & x[2] & ~x[3]

@qpu
def grover_iter(q):
  return q | oracle.sign | 'pppp' >> -'pppp'









print(grover())
```

Hello World in Qwerty

```
from qwerty import *

@classical
def oracle(x: bit[4]) -> bit:
  return x[0] & ~x[1] & x[2] & ~x[3]

@qpu
def grover_iter(q):
  return q | oracle.sign | 'pppp' >> -'pppp'

@qpu
def grover():
  return ('pppp' | grover_iter
                 | grover_iter
                 | grover_iter
                 | measure**4)

print(grover())
```

Qubit Initialization

Qubit Literals: String Analogy

- Standard basis: `'0'` and `'1'`

Qubit Literals: String Analogy

- Standard basis: `'0'` and `'1'`
- In typical classical PLs,
  `"yee" + "haw" == "yeehaw"`

Qubit Literals: String Analogy

- Standard basis: `'0'` and `'1'`
- In typical classical PLs,
  `"yee" + "haw" == "yeehaw"`
- In Qwerty,
  `'0' * '1' == '01'`

Qubit Literals: Tilt

- Phase $e^{i\pi/4}|1\rangle$ represented as *tilt*: `'1' @ 45`
- Syntax evokes `'1'`⥀`45`

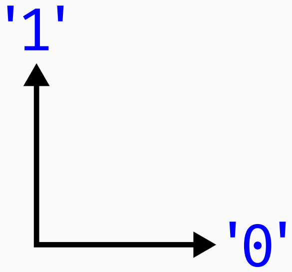
Qubit Literals: Tilt

- Phase $e^{i\pi/4}\,|1\rangle$ represented as *tilt*: `'1' @ 45`
- Syntax evokes `'1'`⟳`45`
- Fun to draw as 

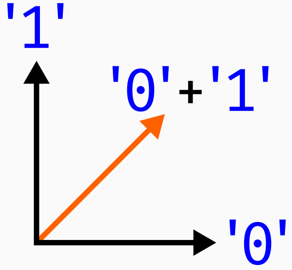
Qubit Literals: Tilt

- Phase $e^{i\pi/4}\,|1\rangle$ represented as *tilt*: `'1' @ 45`
- Syntax evokes `'1'`⟲`45`
- Fun to draw as 
- `-'1'` is syntactic sugar for `'1'@180`

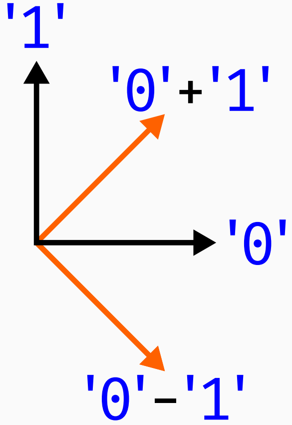
Qubit Literals: Superposition



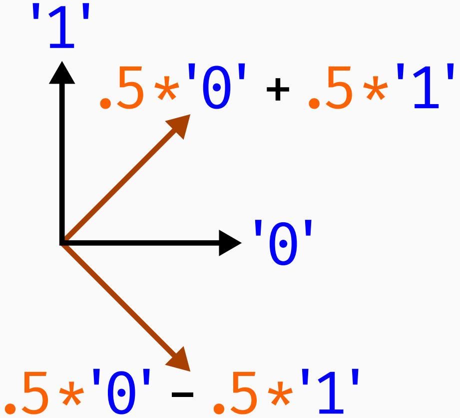
Qubit Literals: Superposition



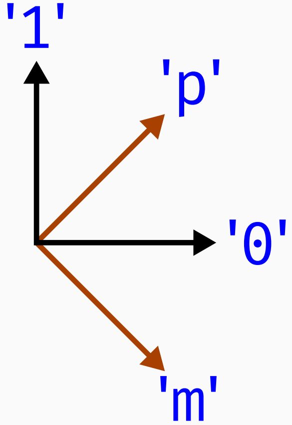
Qubit Literals: Superposition



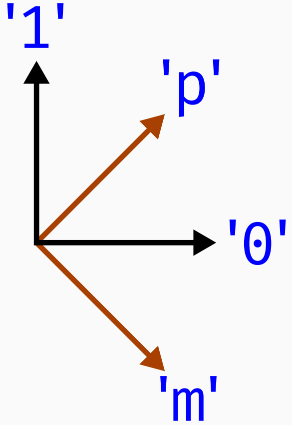
Qubit Literals: Superposition



Qubit Literals: Superposition



Qubit Literals: Superposition



Why `p` and `m` instead of `+` and `-`?

Because this looks confusing: `'+'+-'-'`

State Evolution

Basis Translations

```
{'0','1'} >> {'m','p'}
```

Basis Translations

```
{'0','1'} >> {'m','p'}
```

Replace with

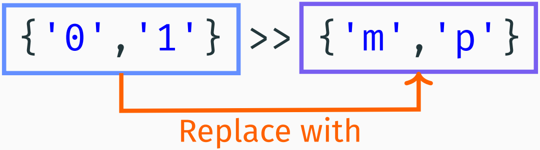
Basis Translations

```
{'0','1'} >> {'m','p'}
```

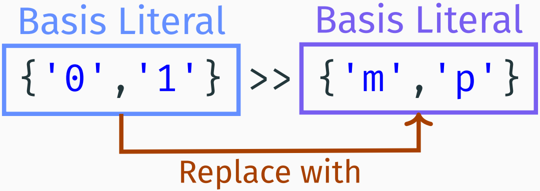
Basis Translations

```
{'0','1'} >> {'m','p'}
```

Basis Translations



Basis Translations



Basis Literals

- Ordered list of basis vectors
- Example: `{'00','01','10','11'}` is two-qubit standard basis

Basis Literals

- Ordered list of basis vectors
- Example: `{'00','01','10','11'}` is two-qubit standard basis
- Could also write `{'0','1'}*{'0','1'}`

Basis Literals

- Ordered list of basis vectors
- Example: `{'00','01','10','11'}` is two-qubit standard basis
- Could also write `{'0','1'}*{'0','1'}`
- Or `{'0','1'}**2`

Basis Literals

- Ordered list of basis vectors
- Example: `{'00','01','10','11'}` is two-qubit standard basis
- Could also write `{'0','1'}*{'0','1'}`
- Or `{'0','1'}**2`
- Must be an orthonormal basis
  - Example: `{'00','10',-'10'}` is invalid

Basis Translation Type Checking

Any b_{in} `>>` b_{out} requires $\mathrm{span}(b_{\text{in}}) = \mathrm{span}(b_{\text{out}})$.

Basis Translation Type Checking

Any b_{in} `>>` b_{out} requires $\mathrm{span}(b_{\text{in}}) = \mathrm{span}(b_{\text{out}})$.

✓ `{'0','1'} >> {'0','1'@90}`

Basis Translation Type Checking

Any b_{in} `>>` b_{out} requires $\text{span}(b_{\text{in}}) = \text{span}(b_{\text{out}})$.

✓ `{'0','1'} >> {'0','1'@90}`

✗ `{'0'} >> {'1'}`

Measurement

- `b.measure` measures in the basis `b`
- Measure in standard basis: `{'0','1'}.measure`

Measurement

- `b.measure` measures in the basis `b`
- Measure in standard basis: `{'0','1'}.measure`
- Measure in Bell basis:

```
{'00'+'11',
 '00'-'11',
 '10'+'01',
 '01'-'10'}.measure
```

Superdense Coding in Qwerty

```
message = bit[2](0b10)

@qpu
def superdense():
    bit0, bit1 = message
    alice, bob = '00' + '11'

    sent_to_bob = (alice | ({'0','1'} >> {'1','0'}
                            if bit0 else id)
                         | ({'1'} >> {-'1'}
                            if bit1 else id))

    recovered_message = (sent_to_bob * bob
                         | {'00'+'11', '00'-'11',
                            '10'+'01', '01'-'10'}
                           .measure)
    return recovered_message
```

Superdense Coding in Qwerty

```
message = bit[2](0b10)

@qpu
def superdense():
    bit0, bit1 = message
    alice, bob = '00' + '11'

    sent_to_bob = (alice | ({'0','1'} >> {'1','0'}
                             if bit0 else id)
                         | ({'1'} >> {-'1'}
                             if bit1 else id))

    recovered_message = (sent_to_bob * bob
                         | {'00'+'11', '00'-'11',
                            '10'+'01', '01'-'10'}
                           .measure)
    return recovered_message
```

Entangled pair

Superdense Coding in Qwerty

```
message = bit[2](0b10)

@qpu
def superdense():
    bit0, bit1 = message
    alice, bob = '00' + '11'

    sent_to_bob = (alice | ({'0','1'} >> {'1','0'}
                            if bit0 else id)
                         | ({'1'} >> {-'1'}
                            if bit1 else id))

    recovered_message = (sent_to_bob * bob
                         | {'00'+'11', '00'-'11',
                            '10'+'01', '01'-'10'}
                         .measure)
    return recovered_message
```

Entangled pair

Measure in Bell basis

Metaprogramming

metaQwerty Motivation

- Tedious:
```
{'0','1'}.measure
* {'0','1'}.measure
* {'0','1'}.measure
```

metaQwerty Motivation

- Tedious:
```
{'0','1'}.measure
* {'0','1'}.measure
* {'0','1'}.measure
```

- What about `{'0','1'}.measure**3`?

metaQwerty Motivation

- Tedious:
  ```
  {'0','1'}.measure
  * {'0','1'}.measure
  * {'0','1'}.measure
  ```
- What about `{'0','1'}.measure**3`?
- Or `std.measure**3`?

metaQwerty Motivation

- Tedious:
  ```
  {'0','1'}.measure
  * {'0','1'}.measure
  * {'0','1'}.measure
  ```
- What about `{'0','1'}.measure**3`?
- Or `std.measure**3`?
- Or even `measure**3`?

metaQwerty Motivation

- Tedious:
  ```
  {'0','1'}.measure
  * {'0','1'}.measure
  * {'0','1'}.measure
  ```
- What about `{'0','1'}.measure**3`?
- Or `std.measure**3`?
- Or even `measure**3`?
- *metaQwerty* expands to Qwerty

metaQwerty Prelude: Qubit Symbols

```
'0'.sym = __SYM_STD0__()
'1'.sym = __SYM_STD1__()
```

metaQwerty Prelude: Qubit Symbols

```
'0'.sym = __SYM_STD0__()
'1'.sym = __SYM_STD1__()
'p'.sym = '0' + '1'

'm'.sym = '0' + '1'@180
```

metaQwerty Prelude: Qubit Symbols

```
'0'.sym = __SYM_STD0__()
'1'.sym = __SYM_STD1__()
'p'.sym = '0' + '1'
'i'.sym = '0' + '1'@90
'm'.sym = '0' + '1'@180
'j'.sym = '0' + '1'@270
```

metaQwerty Prelude: Bases

```
std = {'0', '1'}
pm = {'p', 'm'}
ij = {'i', 'j'}
bell = {'00'+'11',
        '00'-'11',
        '10'+'01',
        '01'-'10'}
```

metaQwerty Prelude: Bases

```
std = {'0', '1'}
pm = {'p', 'm'}
ij = {'i', 'j'}
bell = {'00'+'11',
        '00'-'11',
        '10'+'01',
        '01'-'10'}

fourier[1] = pm
fourier[N] = fourier[N-1] // std.revolve
```

metaQwerty Prelude: Convenience Functions

```
id = {'0','1'} >> {'0','1'}
flip = {'0','1'} >> {'1','0'}
measure = std.measure
```

metaQwerty Example: N-Qubit Grover's

```
def grover2(oracle, num_iter):
    @qpu[[N]]
    def grover_iter(q):
        return (q | oracle.sign
                  | 'p'**N >> -'p'**N)

    @qpu[[N]]
    def kernel():
        return ('p'**N
                | (grover_iter
                   for i in range(num_iter))
                | measure**N)

    return kernel()
```

metaQwerty Example: N-Qubit Grover's

```
def grover2(oracle, num_iter):
    @qpu[[N]]
    def grover_iter(q):
        return (q | oracle.sign
                  | 'p'**N >> -'p'**N)

    @qpu[[N]]
    def kernel():
        return ('p'**N
                | (grover_iter
                   for i in range(num_iter))
                | measure**N)

    return kernel()
```

Polymorphism

metaQwerty Example: N-Qubit Grover's

```
def grover2(oracle, num_iter):
    @qpu[[N]]
    def grover_iter(q):
        return (q | oracle.sign
                  | 'p'**N >> -'p'**N)

    @qpu[[N]]
    def kernel():
        return ('p'**N
                | (grover_iter
                   for i in range(num_iter))
                | measure**N)

    return kernel()
```

Polymorphism

Macros

Predication

Predication Example

Flip right qubit if left is `'1'`:

```
flip if '1_' else id
```

Predication Example

Flip right qubit if left is `'1'`:

```
flip if '1_' else id
```

Basis pattern

Predication Example

Flip right qubit if left is `'1'`:

`flip if` `'1_'` **`else id`**

Basis pattern

Performs:

`'00'` $\mapsto$ `'00'`

`'01'` $\mapsto$ `'01'`

`'10'` $\mapsto$ `'11'`

`'11'` $\mapsto$ `'10'`

Another Predication Example

```
pm >> std if {'p_p', 'm_m'} else id
```

Performs:

'**p**p**p**'	↦	'**p**0**p**'	'ppm'	↦	'ppm'
'**p**m**p**'	↦	'**p**1**p**'	'pmm'	↦	'pmm'
'mpp'	↦	'mpp'	'**m**p**m**'	↦	'**m**0**m**'
'mmp'	↦	'mmp'	'**m**m**m**'	↦	'**m**1**m**'

Full Example: Bernstein–Vazirani

Bernstein-Vazirani Algorithm

Input: Black box for $f(x) = \underbrace{x_1 s_1}_{\text{AND}} \underbrace{\oplus}_{\text{XOR}} x_2 s_2 \oplus \cdots \oplus x_n s_n$

Output: Secret bitstring s used to build oracle

Bernstein-Vazirani Algorithm

Input: Black box for $f(x) = \underbrace{x_1 s_1}_{\text{AND}} \underbrace{\oplus}_{\text{XOR}} x_2 s_2 \oplus \cdots \oplus x_n s_n$

Output: Secret bitstring s used to build oracle

- Any classical algorithm needs to run the black box n times:

$$\begin{aligned} s_1 &= f(1000\cdots00) \\ s_2 &= f(0100\cdots00) \\ &\vdots \\ s_n &= f(0000\cdots01) \end{aligned}$$

Bernstein-Vazirani Algorithm

Input: Black box for $f(x) = \underbrace{x_1 s_1}_{\text{AND}} \underbrace{\oplus}_{\text{XOR}} x_2 s_2 \oplus \cdots \oplus x_n s_n$

Output: Secret bitstring s used to build oracle

- Any classical algorithm needs to run the black box n times:

$$\begin{aligned} s_1 &= f(1000\cdots00) \\ s_2 &= f(0100\cdots00) \\ &\vdots \\ s_n &= f(0000\cdots01) \end{aligned}$$

- **Quantum algorithm only needs 1 query!**

Bernstein–Vazirani in Qwerty

```
from qwerty import *

def bv(secret_string):










    return kernel()

secret_string = bit[4](0b1101)
print(bv(secret_string))
```

Bernstein–Vazirani in Qwerty

```
from qwerty import *

def bv(secret_string):
    @classical
    def f(x):
        return (secret_string & x).xor_reduce()







    return kernel()

secret_string = bit[4](0b1101)
print(bv(secret_string))
```

Bernstein–Vazirani in Qwerty

```
from qwerty import *

def bv(secret_string):
    @classical
    def f(x):
        return (secret_string & x).xor_reduce()

    @qpu[[N]]
    def kernel():
        return ('p'**N | f.sign
                       | pm**N >> std**N
                       | measure**N)

    return kernel()

secret_string = bit[4](0b1101)
print(bv(secret_string))
```

Bernstein–Vazirani in Qwerty

```
from qwerty import *

def bv(secret_string):
    @classical
    def f(x):
        return (secret_string & x).xor_reduce()

    @qpu[[N]]
    def kernel():
        return ('p'**N | f.sign
                       | pm**N >> std**N
                       | measure**N)

    return kernel()

secret_string = bit[4](0b1101)
print(bv(secret_string))
```

Bernstein–Vazirani in Qwerty

```
from qwerty import *

def bv(secret_string):
    @classical
    def f(x):
        return (secret_string & x).xor_reduce()

    @qpu[[N]]
    def kernel():
        return ('p'**N | f.sign
                       | pm**N >> std**N
                       | measure**N)

    return kernel()

secret_string = bit[4](0b[redacted])
print(bv(secret_string))
```

Tracing Bernstein–Vazirani in Qwerty

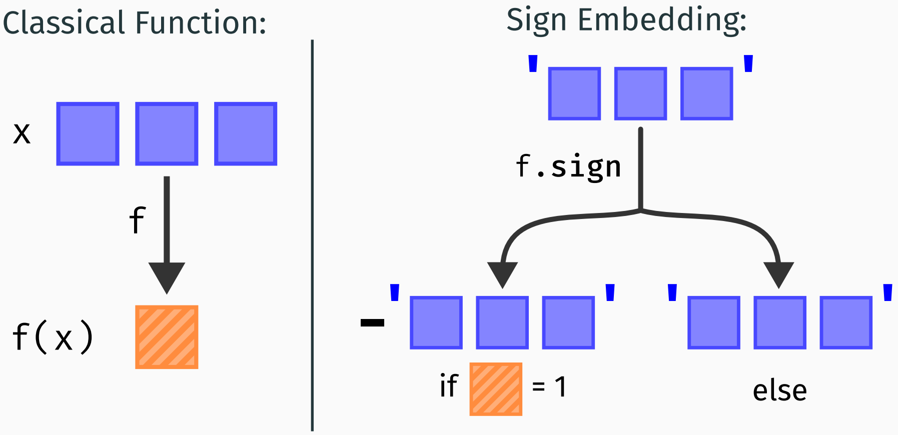
```
  ('0' + '1') * ('0' + '1')
* ('0' + '1') * ('0' + '1')
```

Tracing Bernstein–Vazirani in Qwerty

```
  ('0' + '1') * ('0' + '1')
* ('0' + '1') * ('0' + '1')
```

① ↓ f.sign

Sign Embedding



Bernstein–Vazirani Insight

```
f.sign == f_1.sign
          * f_2.sign
          * f_3.sign
          * f_4.sign
```

where

$$f_i(x_i) = \underbrace{x_i s_i}_{\text{AND}}$$

Tracing Bernstein–Vazirani in Qwerty

```
  ('0' + '1') * ('0' + '1')
* ('0' + '1') * ('0' + '1')
```

① ↓ `f.sign`

```
  ('0' + -'1') * ('0' + -'1')
* ('0' +  '1') * ('0' + -'1')
```

Tracing Bernstein–Vazirani in Qwerty

```
  ('0' + '1') * ('0' + '1')
* ('0' + '1') * ('0' + '1')
```

① ⬇ f.sign

```
      'm'      *      'm'
*     'p'      *      'm'
```

Tracing Bernstein–Vazirani in Qwerty

```
  ('0' + '1') * ('0' + '1')
* ('0' + '1') * ('0' + '1')
```

① f.sign

```
      'm'     *     'm'
*     'p'     *     'm'
```

② pm**4 >> std**4

```
      '1'     *     '1'
*     '0'     *     '1'
```

Tracing Bernstein–Vazirani in Qwerty

```
  ('0' + '1') * ('0' + '1')
* ('0' + '1') * ('0' + '1')
```

① ⬇ `f.sign`

```
      'm'      *      'm'
*     'p'      *      'm'
```

② ⬇ `pm**4 >> std**4`

```
      '1'      *      '1'
*     '0'      *      '1'
```

③ ⬇ `measure**4`

`bit[4](0b`**`1101`**`)`

More Examples in Paper

- Teleportation
- Quantum phase estimation (with tracing)
- Period finding (with tracing)
- Shor’s (order finding)

Conclusion

In this talk, I presented Qwerty, a *basis-oriented* quantum programming language that allows non-experts to reason about quantum computation without knowing bra–ket notation or circuit synthesis.

Conclusion

In this talk, I presented Qwerty, a *basis-oriented* quantum programming language that allows non-experts to reason about quantum computation without knowing bra–ket notation or circuit synthesis.

Compiler (ASDF) paper:

CGO ’25

Source code (GitHub):

`github.com/gt-tinker/qwerty`

Backup Slides

Fourier Basis Details

Fourier Basis: Motivation

QFT:

```
std**N >> fourier[N]
```

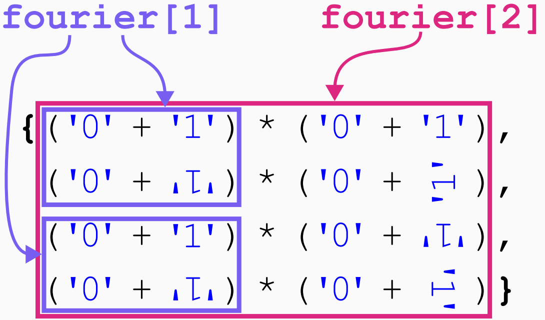
QFT†:

```
fourier[N] >> std**N
```

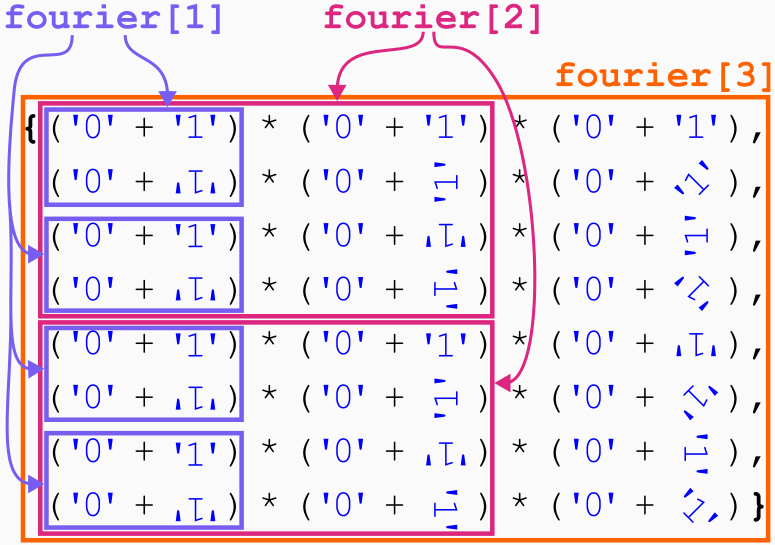
Fourier Basis

fourier[1]

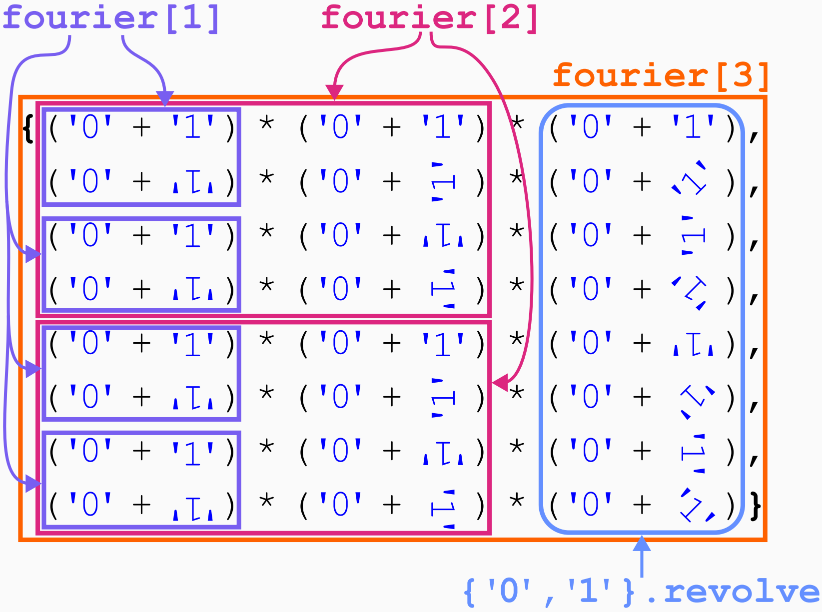
```
{ ('0' + '1'),
  ('0' + ,Ɩ,) }
```


fourier[1]
fourier[2]
{('0' + '1') * ('0' + '1'),
('0' + '1') * ('0' + '1'),
('0' + '1') * ('0' + '1'),
('0' + '1') * ('0' + '1')}

Fourier Basis



Fourier Basis



metaQwerty Prelude: Fourier Basis

```
fourier[1] = pm
fourier[N] = fourier[N-1] // std.revolve
```

metaQwerty Prelude: Fourier Basis

```
fourier[1] = pm
fourier[N] = fourier[N-1] // std.revolve
```

Basis generator

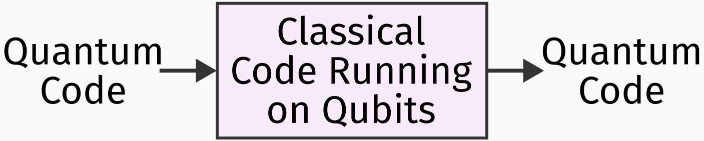
```
fourier[1] = pm
fourier[N] = fourier[N-1] // std.revolve
```

Basis generator

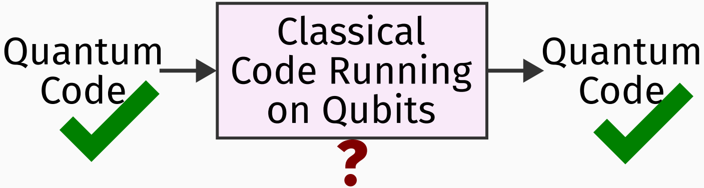
Apply basis generator

Embedding Classical Functions

Plugging the Oracle Hole



Plugging the Oracle Hole



Revisiting Grover's in Qwerty: Classical Embeddings

```
from qwerty import *

@classical
def oracle(x: bit[4]) -> bit:
  return x[0] & ~x[1] & x[2] & ~x[3]

@qpu
def grover_iter(q):
  return q | oracle.sign | 'pppp' >> -'pppp'

@qpu
def grover():
  return ('pppp' | grover_iter
                 | grover_iter
                 | grover_iter
                 | measure**4)

print(grover())
```

Revisiting Grover's in Qwerty: Classical Embeddings

```
from qwerty import *

@classical
def oracle(x: bit[4]) -> bit:
  return x[0] & ~x[1] & x[2] & ~x[3]

@qpu
def grover_iter(q):
  return q | oracle.sign | 'pppp' >> -'pppp'

@qpu
def grover():
  return ('pppp' | grover_iter
                 | grover_iter
                 | grover_iter
                 | measure**4)

print(grover())
```

Classical func. def.

Revisiting Grover’s in Qwerty: Classical Embeddings

```
from qwerty import *

@classical
def oracle(x: bit[4]) -> bit:
  return x[0] & ~x[1] & x[2] & ~x[3]

@qpu
def grover_iter(q):
  return q | oracle.sign | 'pppp' >> -'pppp'

@qpu
def grover():
  return ('pppp' | grover_iter
                 | grover_iter
                 | grover_iter
                 | measure**4)

print(grover())
```

Classical func. def.

Embedding

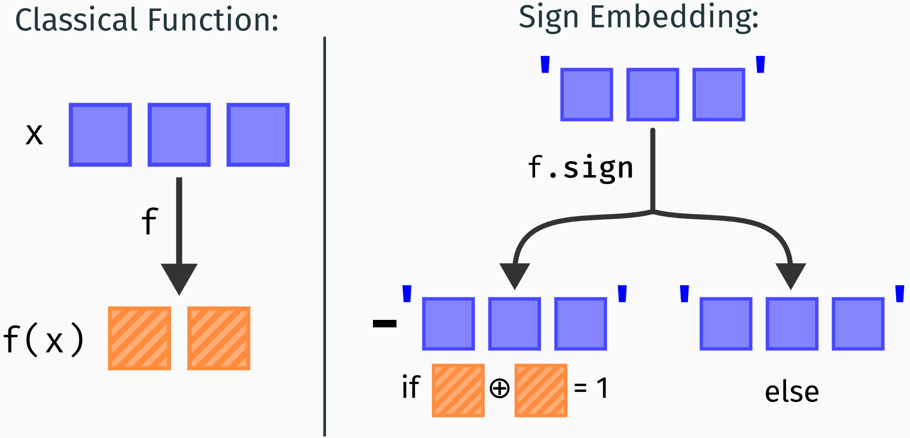
Revisiting Grover's in Qwerty: Classical Embeddings

```
from qwerty import *

@classical
def oracle(x: bit[4]) -> bit:
  return x[0] & ~x[1] & x[2] & ~x[3]

@qpu
def grover_iter(q):
  return q | oracle.sign | 'pppp' >> -'pppp'

@qpu
def grover():
  return ('pppp' | grover_iter
                 | grover_iter
                 | grover_iter
                 | measure**4)

print(grover())
```

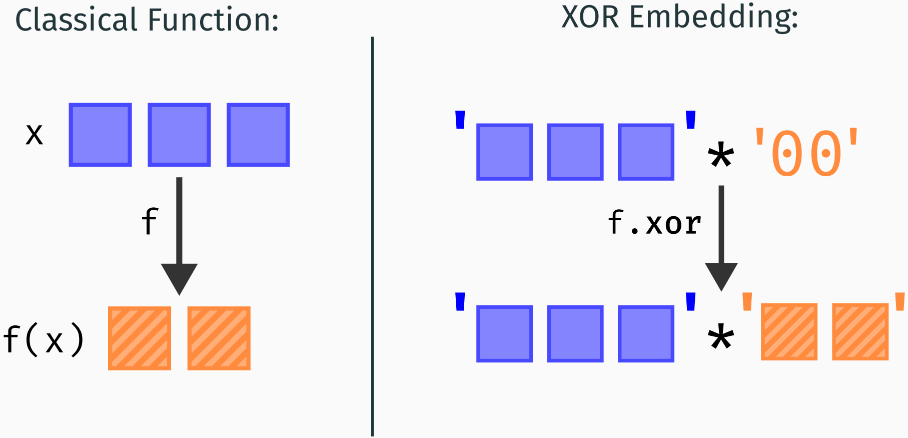
Classical func. def.

Embedding → 3 kinds

Sign Embedding

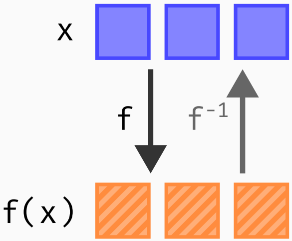


XOR Embedding



In-Place Embedding

Classical Function:



In-Place Embedding: