Notational Programming for Notebook Environments: A Case Study with Quantum Circuits

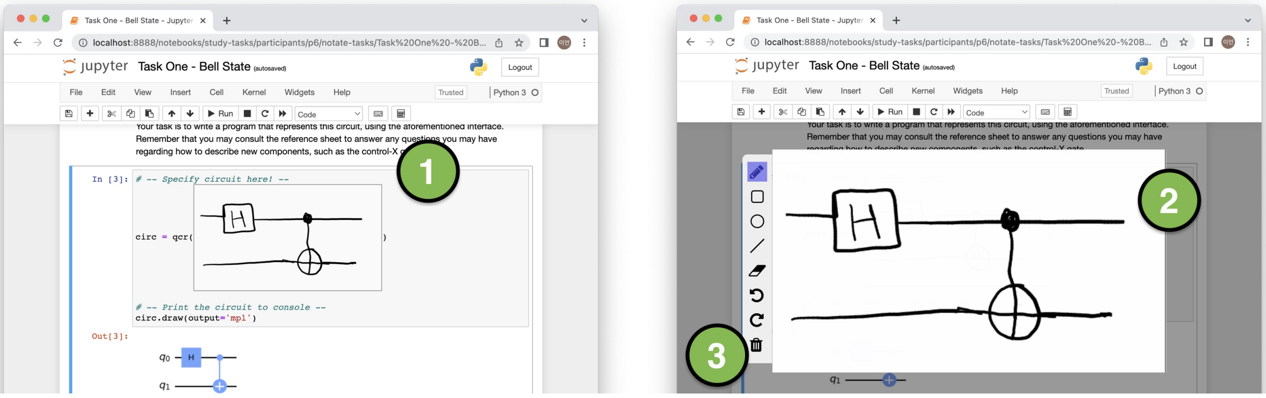
Ian Arawjo
Cornell University
Ithaca, NY, USA
iaa32@cornell.edu

Anthony J. DeArmas
Cornell University
Ithaca, NY, USA
ajd249@cornell.edu

Michael Roberts
Cornell University
Ithaca, NY, USA
mbr82@cornell.edu

Shrutarshi Basu
Harvard University
Cambridge, MA, USA
basus@seas.harvard.edu

Tapan Parikh
Cornell Tech
New York, NY, USA
tapan@represent.org


Task One - Bell State - Jupyter
localhost:8888/notebooks/study-tasks/participants/p6/notate-tasks/Task%20One%20-%20B...
jupyter Task One - Bell State (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
1
2
3
H
q_0
q_1

Your task is to write a program that represents this circuit, using the aforementioned interface. Remember that you may consult the reference sheet to answer any questions you may have regarding how to describe new components, such as the control-X

```
In [3]: # -- Specify circuit here! --

        circ = qcr(                    )

        # -- Print the circuit to console --
        circ.draw(output='mpl')
Out[3]:
```

Figure 1: The main interface to our system, embedded in a Jupyter notebook: (1) a canvas torn open inside a line of code in a cell; (2) fullscreen mode, accessed by touching or clicking on the canvas, with (3) a rudimentary toolbar.

ABSTRACT

We articulate a vision for computer programming that includes pen-based computing, a paradigm we term *notational programming*. Notational programming blurs contexts: certain typewritten variables can be referenced in handwritten notation and vice-versa. To illustrate this paradigm, we developed an extension, Notate, to computational notebooks which allows users to open drawing canvases within lines of code. As a case study, we explore quantum programming and designed a notation, Qaw, that extends quantum circuit notation with abstraction features, such as variable-sized wire bundles and recursion. Results from a usability study with novices suggest that users find our core interaction of implicit cross-context references intuitive, but suggests further improvements to debugging infrastructure, interface design, and recognition rates. Throughout, we discuss questions raised by the notational paradigm, including a shift from 'recognition' of notations to 'reconfiguration' of practices and values around programming, and from 'sketching' to writing and drawing, or what we call 'notating.'

CCS CONCEPTS

• **Human-centered computing** → **Interaction paradigms**; • **Software and its engineering** → **Development frameworks and environments**; • **Computer systems organization** → Quantum computing.

KEYWORDS

programming paradigms, pen-based interfaces, computational notebooks, quantum computing

ACM Reference Format:
Ian Arawjo, Anthony J. DeArmas, Michael Roberts, Shrutarshi Basu, and Tapan Parikh. 2022. Notational Programming for Notebook Environments: A Case Study with Quantum Circuits. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3526113.3545619

1 INTRODUCTION

Today, typewritten systems form the de-facto standard and dominant paradigm for computer programming. Yet, at the advent of programming, the earliest computer programming notations were handwritten, not typed. In the celebrated 1945 *First Report on the EDVAC,* for instance, John von Neumann equated diagrams to text and vice-versa [63]. In fact, notations were only serialized and called programming 'languages' when typewriter interfaces were appropriated for programming [4].

In this paper, we ask how recent advances in pen-based hardware and machine learning might reconfigure programming practice. We articulate one vision, a paradigm we call *notational programming*, which supports communications between handwritten and typewritten notations. Through this work, we seek to question what it means to "write" code –and to accomplish our vision, we argue, may not just entail the development of new user interfaces or improved recognition of existing notations like flow charts, but an active reconfiguration of cultural practices, representations, and values that have historically arisen around programming.

To explore notational programming, we designed an extension to Jupyter notebooks, Notate, that provides the ability to open drawing canvases within lines of code, allowing functions to accept canvas objects natively as arguments. Our architecture also passes these objects a reference to the local scope, enabling typewritten variables to be referred to in the handwritten context and vice-versa. We call this interaction *implicit cross-context references*, extending prior work on bimodal programming by further blurring territories between 'input' and 'output' [33].

To test a notational programming interface and implicit cross-context references in a concrete domain, we chose quantum computing (QC). This choice was strategic: programmers for QC, even when typing code, regularly translate between circuit diagrams and text [25]. An exploratory paper by Ashktorab et al. noted the potential for pen-based computing in QC spaces [5], but no such systems, to the best of our knowledge, exist so far. We introduce a toy notation, Qaw, that augments quantum circuit notation with abstraction features, such as custom gate definition, bundled wires, and recursion. We implemented an interpreter for a subset of Qaw using deep learning and classical computer vision techniques.

To explore the efficacy of our Notate and Qaw prototypes, we then ran a study with 12 programmers who were familiar with Python and computational notebooks but novices to quantum programming. Participants were given six circuits of increasing complexity and tasked with programming them into the machine. We found that almost all participants found our concept of implicit cross-context references intuitive; however, feedback suggests further improvements can be made to debugging infrastructure, interface design, and recognition rates. To validate our approach, we also compared Notate and Qaw to a typical typewritten workflow for quantum programming using the IBM Qiskit API. Results show that, for Python programmers, Qaw was comparable to Qiskit in terms of performance time, but suggest that further research is needed to understand the relative advantages of each approach.

To the best of our knowledge, this paper is the first system to explore a handwritten, diagrammatic paradigm for quantum computer programming (following the suggestion of sketch-based interfaces made in Ashktorab et al. [5]). It is also one of the few papers to explore (and take seriously) a handwritten paradigm of programming, which we define as notational programming. The rest of this paper is organized as follows: the front half covers related work (Section 2), a general description of a notational programming system (Section 3), and a case study with designing Qaw notation for quantum circuits (Section 4). The back half covers our evaluation of Notate and a subset of Qaw: usability study design (Section 5), findings (6), and comparison with a typewritten API (7). Finally, our discussion (8) serves to reflect on our design process, rationale, and comparison with graphical user interfaces (GUIs).

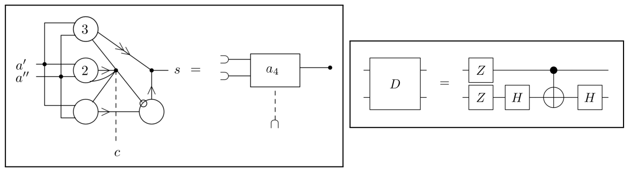


Figure 2: Circuit equalities from the advent of computing to quantum computing. Left: A circuit equality written by von Neumann in *First Draft of a Report on the EDVAC*, 1945 [63]. Right: a circuit equality from a quantum computing text.

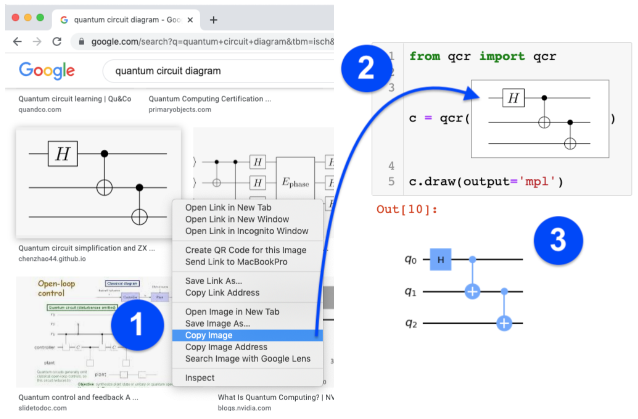
2 RELATED WORK

In this section we summarize prior work in programming and HCI and drawing-based interfaces related to our work. Interfaces for quantum computer programming will be covered in Section 4.

2.1 Programming Systems and HCI

A rich tradition at UIST and beyond focuses on developing novel interfaces for programming. One focus has been on systems for educational or novice users to make entry-level CS more accessible. These include block-based GUI environments, tangible programming with physical objects, or manipulatives in virtual reality (e.g., [66–68, 72]). More recently, a growing community of researchers explore intersections between the fields of computer programming and human-computer interaction, or PL+HCI. Some examples of such work include enabling computers to complete unfinished programs [29, 50], constructively critiquing the design of popular PLs [11, 71], adapting usability methods for introducing new features to existing languages [10], understanding task-switching between languages [31], and using machine learning or crowdsourcing to support code generation [46, 55, 62].

Some programming environments seek to explicitly or implicitly blend "visual" and "textual" representations. Max/Msp and various game engines such as Unity and Godot, for instance, foreground flow diagrams as their main programming interface but retain the ability to customize blocks with textual (typewritten) code in languages such as JavaScript and Lua. Rarely, however, are visuals interspersed within typewritten code [6]. One contemporary exception is the computational notebook paradigm popularized by the iPython notebook platform, which has received major attention by HCI researchers [9, 32, 37, 39, 54, 64, 65, 69]. Work in this area includes "bidirectional" coding, where "visual" and "textual" modalities are mixed in the form of click-and-drag GUIs, and edits

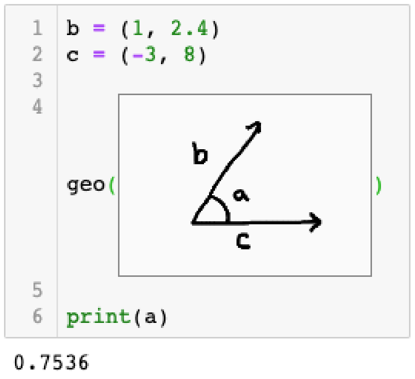

quantum circuit diagram - Goo
google.com/search?q=quantum+circuit+diagram&tbm=isch&
Google
quantum circuit diagram
Quantum circuit learning | Qu&Co
quandco.com
Quantum Computing Certification ...
primaryobjects.com
Quantum circuit simplification and ZX ...
chenzhao44.github.io
Open-loop control
Quantum control and feedback A ...
slidetodoc.com
What Is Quantum Computing? | NV
blogs.nvidia.com
Open Link in New Tab
Open Link in New Window
Open Link in Incognito Window
Create QR Code for this Image
Send Link to MacBookPro
Save Link As...
Copy Link Address
Open Image in New Tab
Save Image As...
Copy Image
Copy Image Address
Search Image with Google Lens
Inspect
1
2
3
H
q_0
q_1
q_2

```
from qcr import qcr

c = qcr(                              )

c.draw(output='mpl')
Out[10]:
```

Figure 3: A user copies an image of a quantum circuit from search results (1) and pastes it directly into a function call (2). The user runs the cell and views output (3), verifying that the interpretation is correct.



```
b = (1, 2.4)
c = (-3, 8)


geo(                              )

print(a)
0.7536
```

Figure 4: In a code cell, a user draws a diagram to calculate the value of angle a between two 2d vectors, b and c, defined as tuples in Python code. The Interpreter geo takes a Canvas and (implicitly) a reference to the local scope. Interpreting the diagram, it associates label a with an angle, realizes that a is not set, and declares it as a new variable in the host scope.

to one affect the other [33, 69]. Other work explores the integration of GUIs inside computational notebooks for visualizing and navigating data [2, 39, 49]. An early precursor to this type of work is the "heterogenous visual languages" vision of Erwig & Meyer [20], which is perhaps closest to the vision we will lay out here, albeit without the focus on pen input.

2.2 Pen-based interfaces for programming

A long tradition in programming tools involves 'sketching,' starting from Iverson's SketchPad and continuing to sketch recognition of diagrams and pen gestures [3, 13, 19, 41, 60]. Within HCI, sketching interfaces have been applied to support UI designers, such as in James Landay & coauthors' SILK and DENIM systems in the mid-90s to early 00s [40, 41, 44], as well as work by Ellen Yi-Luen Do, Mark Gross, Tracy Hammond and Randall Davis [13, 28, 30, 35]. Some of this work has sought to make sketching more interactive, offering tight sketch-interpretation feedback loops where shape gestures are successively recognized and/or beautified [19, 56]. Other systems convert handwritten diagrams into code within unidirectional workflows from early-stage sketches to textual code [45, 59]. One such example is Li et al.'s AlgoSketch, which supported recognition of code-like lines of freehand mathematics notation [42]. But while these and other systems have converted handwritten notation into computer programs [17, 42], to the best of our knowledge, no drawing interface has been embedded within a typewritten programming environment, while allowing for implicit communication between handwritten notation and textual code.

3 WHAT IS NOTATIONAL PROGRAMMING?

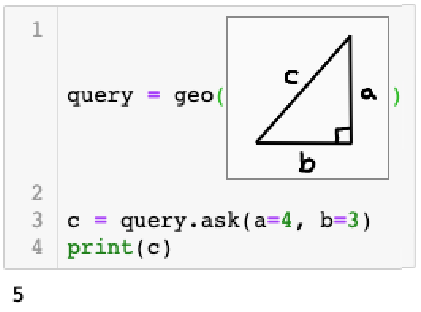
Here, we define the key features, principles, and rationale behind the notational programming paradigm. Our interface, depicting drawing canvases inside typewritten code, is shown in Figure 1. Users may draw on the canvas (1), resize it by dragging the corner, or click/touch it to open up fullscreen view (2), which presents a rudimentary toolbar (3). Canvases in a notebook cell move in response to text edits in the editor (e.g., newlines) and can be deleted, copied, and pasted alongside textual code, analogous to the "interactive visual syntax" GUIs of Andersen et al. [2]. Our interface also allows users to paste in images from outside the notebook to instantiate a new canvas. Figure 3 depicts an example of copying a quantum circuit from Google Image search that is interpreted into an IBM Qiskit QuantumCircuit object.

To illustrate a simple workflow, consider the "notational program" in Figure 4. Here, a user has specified two vectors b and c as 2-tuples in Python code. Below these declarations, the user has drawn a diagram of the kind found in an introductory geometry class, depicting these vectors, writing an angle symbol between them, and labelling the angle ɑ. To declare how their notation should be interpreted, they wrap their drawing canvas in a call to geo(). The geo interpreter then segments and recognizes portions of the drawing, matches labeled parts of the diagrams to information already in the current Python context, solves for the values of undefined variables (here, ɑ), and binds them in the host scope.

This example illustrates what we call *implicit cross-context references*: the typewritten b and c becomes b and c, resp., while the handwritten ɑ implicitly declares a Python variable a in the typewritten context. Note that the meaning of a diagram need not be literal: c in fact points in a different direction than what is drawn.

3.1 Definition and principles

Now that we have built some intuition, we provide a general definition. A notational programming system consists of three components: a *host environment*, a *pen-based interface*, and a *communication protocol* by which they interact. By host environment, we refer to a typewritten or drag-n-drop IDE with a corresponding "host language." For this paper, the Jupyter notebook interface and Python are the host environment. By *notational programming interface*, we mean a system where:



```
query = geo( )

c = query.ask(a=4, b=3)
print(c)
```

Figure 5: In a code cell, a user draws a diagram without pre-specifying values for sides *a*, *b* and *c*. The diagram returns a queryable object. By setting some parameters (here, the ask method), the object returns the value of the remaining unspecified length. Note that this example is meant to express the possibilities afforded by notational programming, not argue the quality of this particular example's API design.

(1) users can **interact** with drawing canvases as first-class pieces of "code" embedded inside the host environment (copy and paste, drop in, delete, etc.)
(2) users can **draw on** or otherwise edit the canvases using drawing features, with a stylus, touch, mouse, etc.
(3) the system facilitates a **communication protocol** between the typewritten and handwritten contexts

A communication protocol specifies how the host environment "sees" image canvases embedded inside of it: what underlying types or objects they represent. In our implementation, the Python environment reads the canvas as an image object (a NumPy array) extended with some additional metadata, such as strokes, pressure data and timestamps. Importantly, the metadata includes a snapshot of the *local scope* captured at the point of execution (when the line of code with the canvas is read by the Python kernel). For brevity, we shall call this image-plus-metadata a Canvas object.

At the most basic level, one could use the notational programming system to set a variable directly equal to an image, or otherwise use it in a function call, without having to first save image data to a file.[1] The Canvas can, moreover, be something users create or edit using standard drawing tools. The intention of a notational programming system, however, is not simply the ease of importing images, but on handwritten notation as a *first-class element* when defining computation. To allow this, a notational programming system facilitates cross-context communication between labels in a handwritten *notation* (a handwritten system of marks, signs, graphics, or characters with a syntax and semantics, such as math, music, state diagrams, etc) with typewritten labels (whether variable names, functions, classes, etc) in the host scope. By cross-context, we mean that not only can handwritten notation reference typewritten variables, but later typewritten notation can reference handwritten variables. We use "variables" broadly to mean any named object in the host scope: functions, classes, etc.

[1]This is similar to drag-n-drop functionality in Mathematica: https://reference.wolfram.com/language/howto/GetAnImageIntoTheWolframSystem.html.

Practically, in order to interpret a notation in a particular execution context, one must write a notation Interpreter. This step is akin to defining a typed literal macro [49], albeit with a handwritten notation recognizer instead of a textual lexer. The process of interpretation can be enumerated into steps, roughly:

(1) **recognition**: computer vision process visually recognizes the notation, syntactically; often this requires a segmentation step where symbols are extracted from "the rest" of the drawing and associated with parts of it
(2) **semantic parser**: the recognized syntactic object is parsed in the notation's semantics (potentially throwing errors or warnings, say for type mismatches or ambiguities)
(3) **communication policy**: informally, a set of read/write rules between the host scope and the interpreter that specifies what variable names may be "read" into the notational context, how they should be translated,[2] and what typewritten variables may be declared or changed during the interpretation that are carried into the host scope. More formally, a read policy would specify both the domain of valid names and the expected types of the referenced variables.

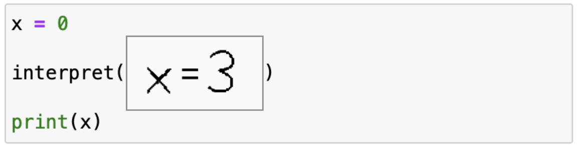
Similar to macros for a typewritten notation [48], over the course of its execution, an Interpreter may:

(1) Read certain variables (as in names) in the host scope
(2) Modify existing variables in the host scope
(3) Declare new variables in the host scope and bind them
(4) Return a value (like a normal function)

An Interpreter that implicitly modifies or reads the local scope acts differently than a typical function, because (at least for Python) it may violate the scoping rules of the host language. For example, the normal Python code:

```
x = 0
def foo(img):
    x = recognize_symbol(img)
foo(Image.load("handwritten_3.png"))
print(x)
```

outputs 0 to the console, because outer variable `x` cannot be set within `foo` (without the `global` keyword). However, the code:



```
x = 0
interpret( )
print(x)
```

for some interpreter `interpret()` would print 3 (assuming, of course, the recognition step was successful). Here we violate Python's scoping rules so that the Interpreter may act similarly to a line of typewritten code –which has access, implicitly, to variables defined in the local/host scope.

Finally, an object returned by an Interpreter may not be a direct value, but require certain parameters in order to specify its value (as in, a lambda function). Consider a triangle with labels 𝔞, 𝔟, and 𝔠 that are not defined in the host environment (Figure 5). The interpreter geo returns an object "with holes" –that is, the meaning is indeterminate until it receives values for (some of) the undefined parameters, akin to a lambda function. For instance, we might use a method `obj.set(a=4)` to set 𝔞 to length 4. Here, the object would

[2]For instance, we may define a policy whereby Greek letters like θ declared in the notational context are accessible in later Python code by referencing the name `theta`.

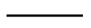
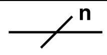
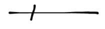
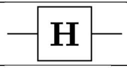
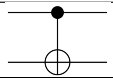
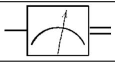
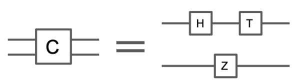
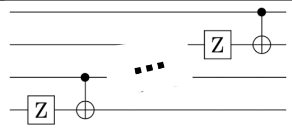
Notation	Term	Use	Representation in Qaw
	wire, qubit or "qubit line"	Stands for a single qubit (quantum-bit). Read from left to right.	(same)
	quantum register (or qubit bundle)	Stands for n qubit lines bundled together. Often used in more informal definitions of circuits.	 (without the n)
	gate	Stands for an operation (here, H) on the qubit(s) input into it (line(s) attached to left of box).	(same, but only capital letters accepted)
	controlled X-gate	Stands for a controlled-X operation on two qubits.	(same)
$\lvert 0\rangle$	ket	Initializes the qubit line to its right (here, to state 0). Sometimes a complex state like $\lvert\psi\rangle$ is written.	(same, but only 0, 1, +, or - currently supported)
	measure	Measures a qubit, collapsing the quantum state to a binary value.	–| (line with a stopper)
 (for instance)	assignment operation or subcircuit definition	Defines a subcircuit of the given name (here, C). The subcircuit can be used as a gate in later circuits.	With Notate, accomplished by setting a typewritten letter `A-C` equal to a drawing of the circuit wrapped in a `qcr()` call.
 (for instance)	ellipses operation (used informally)	Informal. The ellipses stands for repeating a pattern across n qubit lines. Pattern is inferred from surrounding context. Sometimes paired with parametrized gates.	Accomplished via recursive definition, which uses both assignment and slash-wire notation for bundling qubits. See Appendix A section *A.0.8* for example.

Table 1: Some common elements of notation that practitioners use to write quantum circuits. For a full description of notation included in Qaw, see Appendix A. For an intro to quantum computing, see the Qiskit textbook [34].

returns another object where `a=4`. This object would still need `b` or `c` to be defined, to infer the last side. The return object will typically need to do some unification and constraint solving in order to fill in these fields.

4 CASE STUDY: THE QAW QUANTUM CIRCUIT NOTATION

Having described the notational programming paradigm in the previous section, we now narrow our scope to explore one potential application domain: quantum programming. This section reviews our motivations, design methodology, and specification for the Qaw (quantum-draw) notation and provides two examples. To aid readers less familiar with quantum computing (QC), Table 1 lists some common notation for quantum circuits, along with names, uses, and corresponding notation in Qaw. A full accounting of the notation included in Qaw appears in Appendix A. For an introduction to quantum computing, see the Qiskit textbook [34].

Quantum circuits are used to describe algorithms run on quantum computers. The general workflow of a researcher developing a new quantum algorithm is split into roughly two steps:

(1) *Pencil and paper.* A programmer plans their algorithm by calculating in quantum mechanics and linear algebra notation, and draws a quantum circuit.
(2) *Typing code and debugging.* The programmer translates their circuit into a typewritten programming language/API, outputs a diagram representation and inspects it, runs the code and observes the outcome, and edits and debugs.

Current software for quantum programming primarily supports step 2 of this process [5]. Typewritten approaches range from APIs in existing languages (e.g., Python for IBM Qiskit and Google Cirq [24, 34]) to entirely new languages (e.g., Microsoft Q# [61]). Researchers have also developed drag-n-drop graphical user interfaces, aiming to make quantum computing easier for novices (e.g., IBM Composer and Quirk [5, 21]). We were motivated to choose quantum computing for our case study after noticing how circuit diagrams remain a central feature of QC APIs and often appear side-to-side with typewritten code in many resources.[3]

4.1 Notation Design Process

One of the first steps in designing notational programming interface involves designing a notation for a particular domain. To produce the Qaw notation, we conducted a survey of quantum programming resources. We surveyed circuits as they appear in papers on quantum algorithms, online tutorials and wikis, sketches in blog posts and class notes, example code, and textbooks. Given that one of the limitations of contemporary GUIs for QC is their lack of abstraction tools, we particularly paid attention to how authors handle abstraction in their circuit diagrams. We found several elements for denoting abstraction:

- Slashes bundle an abstract number of qubit lines, usually with a parameter n written above the slash
- Sub-circuits are declared using an assignment operator =

[3]In Google Cirq, for instance, a circuit diagram is output as ASCII text to the terminal [24]; in IBM Qiskit, a diagram is drawn to a Jupyter plot [34]; Quipper outputs diagrams [25] and QuECT embeds ASCII quantum circuits into existing PLs [8].

- Ellipses (...) are used to imply a repeating pattern within or across qubit and bit lines (e.g., last row of Table 1)
- Parameters appear either as arguments in parentheses or exponentials to a gate name
- A less common but powerful operation is recursive circuit definition, for instance in Rennela & Staton [53, p. 18]

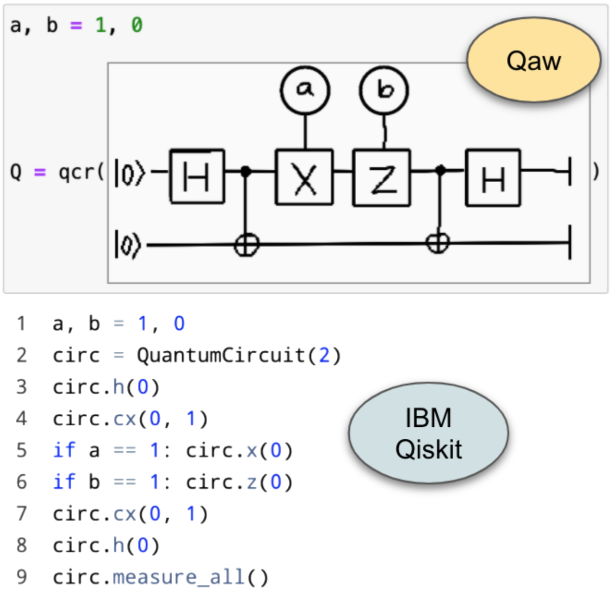
We aimed to incorporate many of these prior conventions while designing our notation with an eye towards simplicity and reducing the effort required to hand-write elements. For instance, in Qaw one does not need to write a size parameter n for wire bundles above a slash, except in cases where one wishes to use the parameter elsewhere, needs to distinguish it from another, or wishes to implicitly define its size by inheriting from a typewritten variable. Nor do they need to explicitly write the tensor product in a gate (e.g., the $^{\otimes n}$ in $H^{\otimes n}$), since the type of gate may be inferred from the type of the input. Our choices were also governed by recognition accuracy –e.g., we opted not to include ellipses (as an option for suggesting repeated segments of circuits) since ellipses may be more prone to recognition error and ambiguity. Instead, the power of ellipses is obtained through recursive circuit definition.

While designing this notation, the first author paper prototyped how one would apply the notation to implement real quantum algorithms by handwriting solutions to tutorials in IBM Qiskit and Microsoft Q#, alongside the Python and Q# solutions to these tasks. They applied an iterative design process to amend the notation, reducing effort in favor of brevity where possible (e.g., the choice to depict the measure symbol as a capped output line, –|), or extending it (in the case of measures that then control later qubit gates, used in a quantum teleportation circuit). Nevertheless, just as programming languages like JavaScript are never "final," so too do we expect notations to evolve and change as time goes on and more communities come into contact with the technology [36]. We leave a full description of our current iteration of the Qaw notation and its components to Appendix A, for interested readers.

4.2 Examples: Superdense Coding & Grover's Algorithm

To illustrate how Qaw works in practice, we wrote a small "notational program" for superdense coding (Figure 6), a common example quantum algorithm [34, 51]. Here, a user has specified bits a and b in typewritten code. They then drew a diagram that uses these variables to control gates X and Z, which essentially functions as an `if` statement –if a, then apply gate X; otherwise, let the qubit pass through this part untouched. Measure notation –capped ends of output wires –indicate to measure both qubits. A later "run" method (not pictured) would then run the circuit Q on a quantum computer, observing the output. The equivalent Python code using Qiskit is depicted at the bottom of the figure.

For a more complex example, consider the general circuit for Grover's Algorithm, as presented in the Qiskit textbook (Figure 7, top). This "abstract" circuit uses a common, albeit informal abstraction, the slashed-wire or "wire bundle" notation, to represent n inputs succinctly. The circuit may be coded in the current iteration of our Qaw notation using a similar slash (Fig. 7, bottom), here



```
a, b = 1, 0
Q = qcr(
```

```
a, b = 1, 0
circ = QuantumCircuit(2)
circ.h(0)
circ.cx(0, 1)
if a == 1: circ.x(0)
if b == 1: circ.z(0)
circ.cx(0, 1)
circ.h(0)
circ.measure_all()
```

Figure 6: A common circuit for superdense coding, handwritten in Qaw (top) with Qiskit code for comparison. Notable features: 1) circles mark classical control bits *a* and *b* 2) kets initialize qubit lines, 3) stoppered outputs represent 'measure' operations. Here *a*, *b* variables are implicitly referenced in the handwritten context as classical bits that control gates.

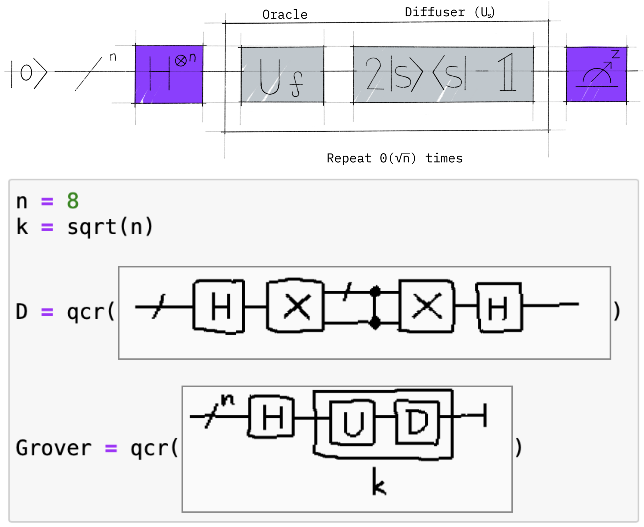
where the diffusion subcircuit D is also written using slash abstractions. To generate the same abstract circuit in Qiskit and Python requires using loops or recursion across n inputs.

4.3 Implementation

We implemented an interpreter `qcr` for a subset of the Qaw notation, using a combination of deep learning and classical sketch recognition techniques. The `qcr` function takes a `Canvas` and outputs either (1) an IBM Qiskit `QuantumCircuit` object or (b) an abstract wrapper over a Qiskit `QuantumCircuit`, called `AbstractQuantumCircuit`, which needs parameters (e.g., n for number of input wires) to generate a "concrete" `QuantumCircuit`. The abstraction is necessary as Qiskit does not support specifying circuits with abstract elements. Including Notate, our full system took about a year for the first author to build, including (re)training of the ML model and an iterative process to improve the heuristic part of the algorithm. The architecture of our recognition system is illustrated in Figure 8.

5 USABILITY EVALUATION

Using this implementation, we proceeded to run a usability test of our Notate interface with a subset of the Qaw notation. Our goals were to investigate how novices would use and perceive a notational programming interface in order to solve a series of QC tasks. We wondered especially about conceptual understandings of our core concept, issues around mode-switching between typing and drawing, and values participants might hold around different types



```
n = 8
k = sqrt(n)

D = qcr(      )

Grover = qcr(      )
```

Figure 7: Top: a way to write Grover's algorithm, from the Qiskit textbook, depicting the slash notation used in many quantum computing resources [34]. Bottom: the above circuit "coded" in Notate with a version of the Qaw notation, where $\mathfrak{D}$ is diffusion circuit and $\mathfrak{U}$ is the oracle (to be defined). The $\mathfrak{D}$ circuit is a solution to Task 3 in our user study.

of coding practices. Here we describe our study design, materials, and participants. We also validated our approach by comparing completion of the same tasks with a typewritten API, which we will discuss in Section 7.

5.0.1 **Task Design and Scaffolding.** Since our system targeted translations between diagrams and typewritten code, we settled on examining the "translation work" [4] users perform when programming quantum circuits into the machine. We designed a progression of tasks focused on a subset of the Qaw notation, designed to (1) introduce novices to quantum circuits and (2) focus on our handwritten references concept, where custom gates, defined as typewritten Python variables with capital letters A-C, may be referred to in handwritten diagrams within the same scope. Much like a traditional API, the Qaw notation includes an array of complex components with nuances that could not be fully introduced within a 2 hour time-frame. Choosing to recognize only a subset also increases accuracy and reduces development costs.

We designed our tasks to lead up to asking participants to program a recursive circuit structurally similar to the Quantum Fourier Transform (QFT), omitting the parametrization and final multi-swap gate. Along the way, we chose some circuits specifically for their relationship to real quantum algorithms: Task 1 is a Bell State; while Task 3 is the *n*-ary diffusion subcircuit of Grover's Algorithm. Other circuits were chosen for pragmatic or scaffolding reasons: Task 2 is a four-line circuit that we expected would require more time drawing than typing; Task 4 introduces the idea of defining subcircuits; and Task 5 introduces recursive definition. Task 6 tests all concepts that users had been taught across Tasks 1-5 (gates, controlled gates, slash-wires, subcircuits, and recursive definition). Our designs of Task 5 and 6 were meant to provide harder programming problems than could be solved by the copying of an example diagram. Further descriptions of our tasks, rationales and example solutions are presented in Appendix B.

5.0.2 **Participants.** We recruited 12 participants (18-27 years old, median 20; 6 male; 6 female) who all self-reported prior experience in Python and using computational notebooks, but no prior knowledge of quantum computer programming.[4] These participants had been randomly selected from a full pool of 24 participants (the remainder selected into our typewritten condition, discussed in Section 7). Given that quantum programming is rather niche, we anticipated that we could not recruit enough in-lab participants with prior expertise; even if we could, expert participants may be biased towards the quantum programming interface they are familiar with [10, 68]. Of those who participated, eleven were undergraduates, and one was a PhD student. Nine majored in CS, with others from Biology, Engineering, and Information Science fields.

5.0.3 **Experiment Design and Procedure.** After written consent, a member of the research team introduced participants to a Microsoft Surface tablet running the Jupyter notebook environment. All participants used the same Surface PC. Participants completed a tutorial, followed by six tasks of increasing complexity with an optional 5 min. break after the third task. Following the tasks, participants were asked to complete a Likert post-survey and a semi-structured interview. The post-survey asked for Likert ratings from 1 (strongly disagree) to 5 (strongly agree) for five questions, listed in Table 2. Each session was capped at 2 hours and participants were compensated $30 in cash for their time.

5.0.4 **Data collection, Setup and Materials.** We asked to record the screen and microphone for the duration the study. A data logger captured user interactions with the Jupyter notebook, such as code cell edits, executions, tracebacks and toggling fullscreen mode. In addition, a researcher typed timestamped observational notes of the participant's interactions, with guidance especially to focus on anything not captured by the screen –e.g. shifting their posture, jotting on scratch paper, or moving the PC.

Participants were given a blank piece of scratch paper and a reference sheet. The sheet included circuit elements they would encounter during the tasks, and was made to mimic API documentation, since participants could not search online. Participants were told they may ask the researcher for a hint if they get stuck; and researchers were allowed to provide a hint if participants seemed to be stuck (e.g., repeating themselves due to confusions around an error). For the Notate condition, since our goal was not to test the accuracy of the recognizer, in the event of a recognition error on a correct (final) solution, the experimenter would let participants know their solution was correct and let them move on. Our tutorial, tasks, and materials are available in the Supplementary Material.

6 FINDINGS

All Notate participants were able to complete the first five tasks, while nine were able to complete the last task within the allotted

[4]Our screening criteria were: *"Participants must have prior experience in Python (taken a class, workshop, etc.), have at least cursory/passing knowledge of Jupyter notebooks, have no prior knowledge of quantum computing, and be comfortable drawing by hand."*

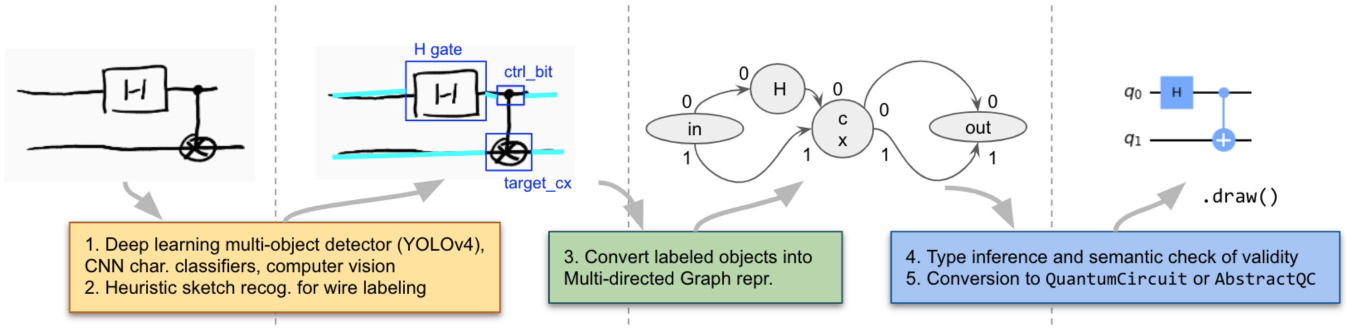

H gate
ctrl_bit
target_cx
1. Deep learning multi-object detector (YOLOv4), CNN char. classifiers, computer vision
2. Heuristic sketch recog. for wire labeling
3. Convert labeled objects into Multi-directed Graph repr.
in
H
c x
out
4. Type inference and semantic check of validity
5. Conversion to QuantumCircuit or AbstractQC
q_0
q_1
.draw()

Figure 8: The process of interpreting a handwritten quantum circuit in our system. ***(Drawing is P14's solution to Task 1.)***

time (exceptions: P4, P6, P19). As shown in Table 2, post-survey results indicate non-normal distributions and high variances, indicating that there was a difference in how users adjusted to the interface. We affinity diagrammed our qualitative data and identified three major clusters of findings: conceptual understandings, error handling and debugging, and general usage patterns with the Surface device and Notate interface.

6.1 Conceptual (mis)understandings

Notate participant's conceptual (mis)understandings could be broken down into three kinds: first, understandings about our core concept of implicit communication between typewritten and handwritten notations; second, understanding the Qaw notation and applying it successfully; and third, maintaining conceptual boundaries between what they considered "coding" versus "drawing."

6.1.1 **Understanding communication between typewritten and handwritten contexts.** By far, participants had few, if any, difficulties understanding our core interaction of referencing typewritten variables within handwritten notation. They used it effectively to reference smaller circuits, typewritten as Python variables such as `A`, inside larger circuits, by handwriting $\mathscr{A}$ (Figure 9). Notate participants would rarely bring up this interaction until pressed with a specific question by the interviewer. P19 recounts:

> *I didn't have any trouble with it. For the most part, it was able to recognize my handwriting and the variable that I typed.... was the same variable that I wrote down. [...] I guess it's the same intuition [as normal programming practice]. It's just instead of typing out a variable that you're referencing later, you're just writing it.*

Three participants, to our surprise, even thought that in-line drawing canvases were an existing feature of Jupyter notebooks which they were simply unaware of (P4, P5, P11; e.g. *"Maybe this is like a feature in Jupyter notebooks?"*). These participants were otherwise familiar with Jupyter and had used it before. Said P5:

> *I wasn't sure if [the canvas] was a built-in library, or some library that already existed? Or if that's part of it? [Interviewer: Part of the Jupyter interface?] Yeah, that produces an image and then qcr() interprets the image.*

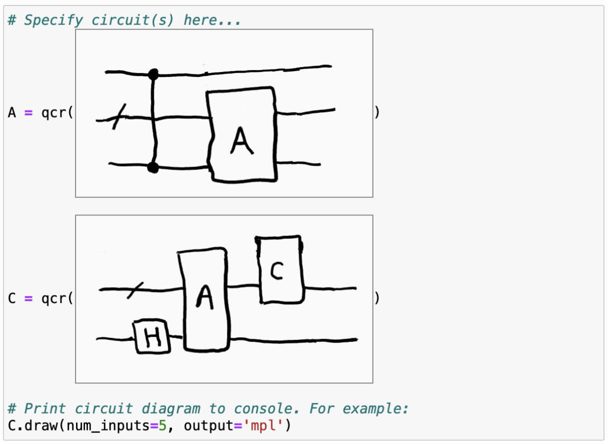
A few participants described an initial hesitation followed by quickly becoming accustomed (P17: *"When I first read about it, it felt very foreign... [but] that idea seemed less foreign and sort of familiar once I did a couple of practices"*). For two participants, one of the main conceptual difficultiues stemmed from recursive definition. These users either did not notice the typewritten `C` (due to small font) in our Task 5 example diagram, or assumed the C gate had a different function. Said P11, who was observed scrolling quickly through the Task 5 text: *"Maybe the C is just like, 'lambda function abstracted away.' [Like C] just means recursion. And I didn't realize that the C was actually on the outside."* Examples of recursive definition are shown in Figure 9.

6.1.2 **Understanding the Qaw notation and abstractions.** Common conceptual errors of syntax across users are pictured in Figure 10. A few users also tried to slash multiple wires at a time, a feature that was unsupported by our implementation since it can lead to semantic ambiguities. Some of these confusions may merely be learning hiccups, but they may also suggest further extensions to the notation to support the varied ways people convey information (e.g., the third example in Fig. 10). When encountering these errors, the debugger would either print a warning, or raise an Exception with an error message. Common error messages included semantic "input / output size mismatch" errors when gates had different numbers of inputs than outputs (e.g., forgetting to draw the output wires to a gate). For Tasks 5 and 6, some participants also encountered "maximum recursion depth exceeded" errors that indicated their recursive circuit definition never terminated.

We anticipated that many participants would struggle on Tasks 5 and 6 due to the presence of recursion (a concept possibly rarely used in most participant's programming in Python), requiring a complex understanding of Qaw abstractions. This, indeed, was what happened for *some* participants: for Task 5, both the median time *and the standard deviation* was approx. 12 minutes; for Task 6, median time was 27 minutes, st. dev 15.5 mins (for a full accounting, see Table 3). Closer examination of Task 5 reveals that two participants were able to complete the task in *a little over 1 minute,* and one about 1.5 min; while the longest two took about *30 min and longer.* Post-interview data indicates a possible reason for the faster participants: three reported that they felt comfortable with recursion and had taken (or were currently taking) the functional programming class at our institution. P21, for instance, a CS major, completed Task 5 in 5min 16s. When asked if anything in her background might have contributed towards her ability to solve

Number	Question	Median	Mean	St dev.	Shapiro-Wilk (W, p)
Q1	The interface was easy to use.	4.0	3.58	1.1645	0.8596, p=0.0483**
Q2	The interface was enjoyable.	4.0	3.50	1.0000	0.8226, p=0.0171**
Q3	When I made a mistake/error, I found it easy to correct.	3.0	3.17	1.4668	0.8384, p=0.0265**
Q4	I felt confident using the interface.	4.0	3.75	1.1382	0.8512, p=0.0380**
Q5	When I completed the tasks, I felt like a programmer.	3.5	3.17	1.4035	0.9056, p=0.1874

Table 2: Post-survey Likert results for our Notate user study. Shapiro-Wilk tests indicate non-normal distributions at $p<0.05$ for all questions except Q5; hence, we report medians. Variances are high, consistent with programming studies [10].



```
# Specify circuit(s) here...
A = qcr(                                   )

C = qcr(                                   )

# Print circuit diagram to console. For example:
C.draw(num_inputs=5, output='mpl')
```

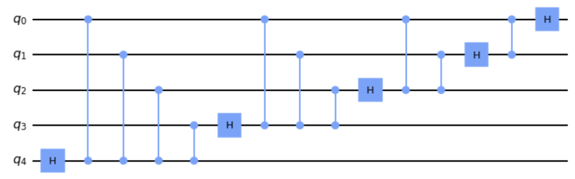


Figure 9: P5's solution to Task 6, generating a pattern similar to the the body of the quantum fourier transform. Circuits $\mathcal{A}$ and C are defined recursively using Qaw slash notation and implicit cross-context references.

the tasks, she said *"I understand recursion and know how to apply it."* Her conceptual issues seemed less to do with recursion than with nuances of the notation, such as where slashes could go.

6.1.3 "This doesn't feel like programming at all": Negotiating boundaries between "coding" and "drawing".

"Coding begins with the drawing of the flow diagrams..."
–Goldstine & von Neumann, 1947 [22, p. 20]

One of our goals in conducting this work was to further explore how programmers negotiate boundaries between what practices constitute "programming" or "coding" and which do not, paralleling the historic separation between the "textual" and the "visual" and the dominance of the term "language" deriving from early adoption of the typewriter [4]. When presented with a series of tasks which effectively forefronts handwritten drawings in programming practice, we wondered whether it would destabilize or challenge participant's notions of programming.

Some Notate participants said that notational programming and especially recursive notation was unlike anything they had never encountered before (P11, a TA for an upper-level CS class: *"I don't really think this compares to any other kind of programming that I've done in the past, like... It's completely different"*) or struggled to compare it to their prior experiences. Participants would often erect boundaries between their experience using the drawing interface and what they considered "coding." Said P21, when asked to compare her prior coding experience to the study:

> *This is definitely less coding; it was more drawing. For me coding is like, writing out... I guess, like this bottom line [points to C.draw()], like actually typing it up?*

Participants appeared to associate the keyboard or "typing" with "coding" – often (somewhat ironically) resorting to the verb "writing" to describe how coding differed from "drawing" –in order to exclude drawing from the category of coding (P19: *"It [the study] is not what I was expecting... I thought I would have to write code"*; P7: *"When I'm programming, I don't have to think about my handwriting... I just have to think about, you know, writing stuff"*). For them, "writing code" was a practice inexorably attached to the keyboard.

Yet, cracks in participants' conceptual boundary-making could also occur. P7 began by claiming "this doesn't feel like programming at all," but as he tried to defend his point, ended up questioning how one would define programming, even asking the interviewer how they would define it. This provides evidence that, above and beyond technical concerns, notational programming interfaces may call into question participants' ideas and values around what "programming" and "writing" code entails, especially as they sit with the concept for longer periods of time.

6.2 Error handling and debugging

In Notate, unlike a typical coding environment, errors may not be syntactic/semantic –errors caused by the user –but "recognition errors," faults of the AI computer vision algorithm.[5] Common to usability studies of notation recognition systems [47], during our studies we observed high variability in recognition rates for some Notate participants versus others. Our post-survey provides some support for this claim: the question with the lowest score and highest variance is "When I made a mistake/error, I found it easy to

[5]Technically, a "recognition error" can also occur in typewritten coding if the syntax/-grammar parser has a bug.

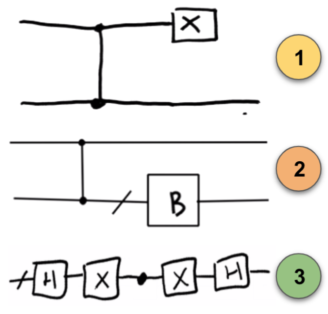


Figure 10: Three common "syntax errors" made by participants: (1) missing an output wire to a gate; (2) slashing an output wire of a control gate; (3) trying to use a single dot to represent a multi-Controlled Z gate in Task 3. In future iterations, we can imagine supporting some of these styles.

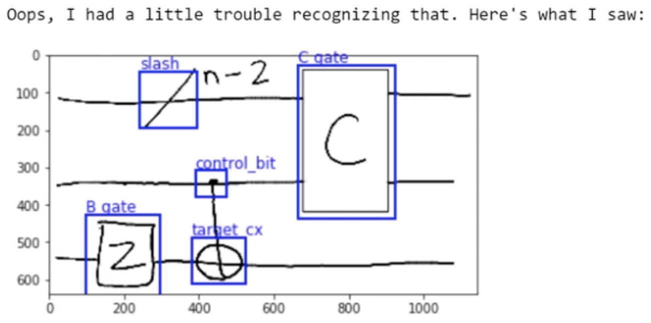

```
Oops, I had a little trouble recognizing that. Here's what I saw:
```

Figure 11: During Task 5, P8 encounters an error plot thrown by the AI recognizer (below the code cell). Noticing that a Z gate was mistaken for a B, P8 responds by erasing and rewriting her Z, then guessing the open corner was also an issue (*"Maybe it's also this thing over here?"*) and filling it in. She runs the cell; it throws another error plot. The researcher present remarks on the $n-2$ as a conceptual misunderstanding –this notation was only used in a tutorial to explain how a recursive definition unrolls. P8 erases it and runs the cell, leading to the expected output (*"Alright! Looks good."*).

correct" (Table 2). And indeed, those who struggled with the recognizer earlier or for longer durations may have rated the interface poorer. For instance, P14, who generally wrote in a very sketchy style, grew frustrated with the recognizer and gave the interface all 1's in her post-survey, the lowest score across participants. By contrast, P21, who rarely encountered a recognition error, rated the interface all 4's. In addition, comparisons with other pen-based interfaces may also be involved: for instance, one participant, in the post-interview, talked about their dislike of the Microsoft Surface, over the iPad & Apple Pencil interface that they were familiar with.

6.2.1 Conflations between semantic and recognition errors.

In practice, the additional uncertainty of recognition errors meant that conceptual misunderstandings of the Qaw notation –where the user is learning, through trial and error, how to apply the syntax –were often confounded by recognition errors or issues parsing (or trusting) error messages.

If it had trouble, the recognition part of the interpreter would throw a plot of what the AI saw (Figure 11); but in practice, this could not catch errors when the AI *thought* it had parsed the drawing correctly, but the semantic parser then threw an error from the misrecognized circuit. This led to situations where participants interpreted recognition errors as semantic or syntax errors and vice-versa. For instance, while completing task 3, P7 struggled with the recognizer. They at first interpreted a recognition error as a semantic one, then tried a bunch more drawings before returning to a drawing semantically equivalent to their original solution –only this time, the recognizer worked, outputting the interpreted circuit:

> *Oh, come on, I did that the first time. [Frustrated, he looks at the output. It doesn't match the solution.] Is that circuit... Damnit. [pause] Oh, that's supposed to happen. Hey yo, no, I don't want to write that stuff [realizing he will have to draw a second line of gates].*

Here, P7 is frustrated in realizing that what they thought was a semantic error was really the fault of the AI. This causes them to momentarily doubt the current, *correct* output –blaming the recognizer rather than their specification –before realizing that the AI was correct this time (*"oh, that's supposed to happen"*), and reattributing blame to themselves (*"no, I don't want to write that stuff"*). Other participants would remark that a major similarity between coding in Python and in Notate was that error messages in Python were not always helpful. For instance, P3 mentions this similarity, along with the new source of "blame":

> *[In regular coding], the error you get is supposed to be helpful, and it's just not, or [it's] straight up wrong... But, being able to blame it on the image processing was kind of interesting. Like, "oh, it could be an error with me, or it could be an error with the computer."*

Consistent with other work in HCI on improvisation and repair, sometimes these frictions were unexpectedly "productive" [38, p. 4]: for the wrong reasons, recognition breakdowns could end up nudging user's behavior in the right way. In other words, users could read *recognition* errors as a sign their circuit was *semantically* invalid –when it in fact was –and then stare at and amend their drawing towards the correct solution.

6.2.2 "It doesn't like my handwriting": Modifying drawings in response to errors.

A common pattern, especially if recognition errors happened early on, involved participants amending their drawing practices to suit what they perceived the AI could understand, such as starting from freehand drawing to resorting to the rectangle and line tools. The sooner the recognizer failed, the faster and more extreme the amendments to their practices. P17, for instance, triggered a recognition error (with plot) on the Tutorial, the only participant to have done so. She then noticed the rectangle tool and began using rect and line tools for the rest the study. At one point she writes a fast Z, then erases it and rewrites it slowly, possibly worried about the AI's interpretation. Participants who rarely triggered a recognition error, by contrast, almost never used the rect, line, or circle tools in the toolbar.

6.3 Interactions with hardware and software

Many Notate participants would touch the screen while mode-switching between pen and keyboard. An especially popular mode of interaction was tapping in-line canvases to enter fullscreen mode, and tapping the background of fullscreen mode to close it. The Ctrl-\ shortcut to place a canvas at the mouse cursor seemed to be successful, with some participants only hesitating on Task 1 to remember the shortcut and the term `qcr()` (resolving the confusion by either using the reference sheet or looking back at the Tutorial). Strikingly, we did not encounter a case where participants, after Task 1, forgot to wrap their canvas in the `qcr()` function.[6] One issue with the Ctrl-\ shortcut was that, likely due to the weight and size of the felt Surface keyboard, some participants accidently pressed Ctrl-+ (plus) when trying to use it, which enlarged the size of the browser content.

Many Notate participants preferred to use fullscreen drawing mode, with few writing directly on the in-line canvases. One exception was P8, who resized canvases to be almost the width of the code cell. So too would participants, especially for Task 5 and 6, use the scratch paper extensively, sometimes to write solutions preemptively on the paper before copying them onto a canvas. Their behaviors here may have to do with worries over the software interface, such as accidentally resizing the canvas, or hardware, such as the glass screen and how participants perceived the fidelity of the pen. Early in the study, P11 put the screen flat on the table, with the felt keyboard flat below it, but the keyboard got in the way, since it was near their elbow, leading to them returning the device to upright (stand) mode. Other participants would remark that they might have tried a "tablet" mode, e.g. P3: *"I would have been fine, every once in a while, when I had to type like qcr() just doing that on a touch screen... because the keyboard was almost entirely unused."* Two participants asked for a "lasso" tool, referencing the Notability app they had used on their iPad. They wanted the lasso tool to move around parts of their drawings or copy parts of drawings within a canvas. Some participants mentioned having, or being familiar with, the iPad Pro and Apple Pencil; only one mentioned prior familiarity with a Microsoft Surface.

7 COMPARISON WITH A TYPEWRITTEN API

To assess the real-world performance of our system, and to validate that our system wasn't significantly worse when compared to typewritten coding, we conducted a comparison of our interface with an alternate condition where participants had to use the IBM Qiskit API and Python to solve the same tasks. We chose Qiskit because it is one of the most popular quantum programming APIs. We also use this comparison to explore various tradeoffs between the two conditions regarding task types and workflows.

7.1 Participants

We recruited 12 additional participants (20-26 yrs age, median 21.5; 8 male, 3 female, 1 unspecified) who had been randomly selected from the full pool of 24 participants across studies, matching a between-group design.[7] The study had the same timeframe of 2 hours and compensation of $30, with Python-equivalent tasks and materials –a reference sheet to emulate API documentation, a piece of scratch paper, and a tutorial. All task explanatory text was altered to present the same conceptual information (as much as possible), except in typewritten code.[8] Examples in tasks were presented as screenshots and could not be copied & pasted, to retain fairness across conditions. All Qiskit participants used the same Microsoft Surface as the Notate users except one, who used a MacBook Pro 2013. Participants could similarly ask for hints and, if they seemed stuck, the researcher present could provide help. We altered the name of the Qiskit API to 'qcirc' to eliminate potential biases around perceptions that a corporate entity had designed the interface. Task notebooks for both conditions are in our Supplementary Material.

7.2 Task by Task Performance

All Qiskit participants completed all tasks, although P13 requested significant help on Task 6 from the researcher present, resulting in pair programming the solution. Since a raw "time to task completion" metric does not factor in potential variability in reading the task text, which varies across conditions, here we estimate start time by the moment the user begins to work in the editor on the problem –editing the code cell to add a canvas or type code. End time is marked by the last execution of a code cell containing a correct solution, before starting on the next task.

Shapiro-Wilk tests of normality indicate that Qiskit completion time data is not normally distributed for all six tasks (p<0.01 for every task except four, which is p<0.02). For the Notate task times, none of the Shapiro-Wilk tests were significant, with only Task 2 near significance at p=0.059<0.1. For post-survey subjective measures, a Shapiro-Wilk test reveals similar non-normality at p<0.05 for all except one sample (Notate condition in Q5); and at p<0.01 for all Qiskit post-survey questions. Because of these violations of normality and our between-subjects design, we report non-parametric tests to compare conditions: Mann-Whitney U to test for differences between distributions, Levine's test with the median (a.k.a. the Brown-Forsythe test) for differences in variances, and Cliff's δ for non-parametric effect size. We report median task times and other statistics in Table 3. Below, we unpack these findings with interpretations from qualitative data.

7.2.1 **Tutorial and Task 1:** No significant differences found. Medians and st devs. are close, suggesting that pen and keyboard input is comparable for entering a simple Bell State circuit.

7.2.2 **Task 2:** indicates a trend towards the Qiskit condition (p=.06), but does not reach significance. We included Task 2 to test the intuitive hypothesis that larger "concrete" circuits –that is, circuits without abstractions, with several gates and control lines that go in exact places –would be easier to solve via the keyboard.

7.2.3 **Task 3:** significant in favor of Notate p=0.04<0.05, with a large effect size (Cliff's δ=−0.5). Task 3 introduces a new abstraction notation, the slash-wire, that participants have likely never seen

[6] In the Tutorial it had been explained that canvases are read internally as images, so possibly participants drew from their Python knowledge to understand `qcr` as a function taking an image argument.

[7] Eight were undergraduates, three PhD/masters students, and one unspecified; six majored in CS, the others in Engineering and Information Science fields.

[8] E.g., where Task 4 in the Notate study introduces subcircuits with an example of use, the Qiskit study presents example code to accomplish the exact same thing.

Task	Notate ($\tilde{x}$, s)	Qiskit ($\tilde{x}$, s)	MWU (U1, p)	Levene w/ med. (W, p)	Cliff's δ (effect size)
Tutorial	173.03s, 63.66s	150.19s, 87.66s	89.00, p=0.3408	0.75, p=0.3957	0.24 (small)
Task 1	59.27s, 25.71s	52.23s, 45.81s	82.00, p=0.5834	0.43, p=0.5202	0.14 (negligible)
Task 2	126.16s, 87.01s	*77.81s, 62.48s*	105.00, *p=0.0606**	0.51, p=0.4837	0.46 (medium)
Task 3	**265.79s, 176.85s**	422.07s, 464.89s	36.00, **p=0.0404****	0.80, p=0.3813	-0.50 (large)
Task 4	*200.28s, 78.84s*	233.71s, 284.03s	43.00, *p=0.0998**	2.58, p=0.1225	-0.40 (medium)
Task 5	721.54s, 722.15s	323.35s, 298.75s	83.00, p=0.5444	5.57, **p=0.0276****	0.15 (small)
Task 6	1619.33s, 925.31s	**386.58s, 796.97s**	125.00, **p=0.0024*****	1.61, p=0.2181	0.74 (large)

Table 3: Task times ($\tilde{x}$=median, s=st. dev) and comparison statistics (Mann-Whitney U, Levene with median) across conditions. (*=p<0.1 indicating a potential trend, **=p<0.05 indicating significant, ***=p<0.01 indicating highly significant)

before. The solution is also non-trivial (Fig. 7, subcircuit `D`), and could have benefited from an extension to the notation whereby a dot captures a bundled multi-controlled Z gate (Fig. 10, #3). In both conditions, we did not include the multi-Controlled Z component in the task's tutorial explicitly, to emulate participants searching an API to solve the task. We did, however, introduce the slash-notation (in Notate) and (in Qiskit) remind participants of the "for i in range(n)" abstraction in Python for accomplishing similar looping over n inputs. Our observations suggest that the multi-Controlled Z gate in Qiskit was a major pain point in the API, yielding confusion for participants both on where to apply it (trying to apply it within a for loop, initially) and how to enter qubit indices as an argument.

7.2.4 **Task 4:** suggests a slight trend in favor of Notate (p=0.09), but does not reach significance. Observations of the Qiskit condition indicate a potential pain point in the API regarding using custom subcircuits, with some participants appearing confused about the difference between output of `.to_instruction()` and subcircuit objects. In Qiskit, one had to cast their subcircuit to an instruction, then use the `.append` command with a list of indices as an argument; while in Notate, one could directly use their subcircuit by drawing a letter $\mathcal{A}$ within a gate of their larger circuit.

7.2.5 **Task 5:** does not reach significance. However, Levene's test shows the two distributions have significantly different variances at p<0.03. In 6.1.2, we noted that the fastest three Notate users took under 1.5 minutes; the longest over 30 mins; here, the fastest Qiskit user took about 3.5 mins; while the longest took 18.5 min.

7.2.6 **Task 6:** significant at p<0.01 in favor of Qiskit, with a large effect size (Cliff's δ=0.71).[9] The relative speed of Qiskit users on Task 6 is as hypothesized by our choices in the task design. However, many Qiskit users were observed copying their solutions from Task 5 into Task 6, then amending inner calls, possibly accounting for the size of the difference.[10] In comparison, Notate participants, although they could copy canvases *within* a notebook, could not copy canvases from one notebook into another.

7.2.7 **Likert Survey.** Post-survey results indicate that Qiskit participants rated the interface as easier to use and more enjoyable compared to Notate (p<0.03; median=5 vs. 4 for both Q1 and Q2), while other metrics did not reach significance. For "When I made a mistake/error, I found it easy to correct," a Levene's test with median yields a significant (p=0.018<0.05) difference in variances, with higher variance for Notate (which we discussed in 6.2). The high ratings for ease-of-use are somewhat unsurprising: after all, we had recruited participants experienced with Python and Jupyter notebooks, and the Qiskit condition did not ask them to do much more than use the interface they were already accustomed to. Our choice of the word "interface" could also have led to confusion, as our post-interview data suggests that what some Qiskit participants were rating were Jupyter notebooks. Participants may also be inclined to blame themselves; for instance, P23 appeared to be struggling with Python and error messages, but in the post-interview did not attribute his frustrations to the interface, even when pressed.

7.3 Other observations

By contrast to Notate participants' reliance on touch and pen interactions and frequent use of the scratch pad on harder tasks, the Qiskit participants rarely used the scratch paper or touched the screen, and only one of them used the stylus to scroll with the scrollbar. Their means of interaction were the fold-out keyboard and trackpad of the Surface, akin to typical programming practice. Sometimes, a participant would be seen ready to write something on the scratch paper (pencil up), but then would return to the keyboard. A few participants used trackpad gestures to zoom out, such that they could see the entire notebook without scrolling.

Across all participants, Qiskit users relied heavily on the provided example code snippets, sometimes copying example code almost verbatim before amending it for the task. As might be anticipated, the types of errors Qiskit participants encountered were also fundamentally different than Notate ones. Qiskit user errors centered around indexing and array-out-of-bounds errors (missing an index, going over by one, etc.). Users typically responded by revising the index argument, sometimes seemingly guessing in multiple quick edit-run-revise cycles until the cell ran without incident or gave the right output. By virtue of the notation, indexing errors were entirely absent from the Notate condition.[11]

Near the end of the post-interview, we asked Qiskit participants an interview question regarding a hypothetical drawing interface

[9]For the three Notate participants who could not finish Task 6, we include their times in the analysis without edit, for they took at least 10 minutes or longer.
[10]P1, P23 and P24 copied the entire code cell; P15, P18 and P20 their entire recursive function; P9 a few lines; P10, P16 flipped rapidly between browser tabs to manually copy parts, including the base case; and P13 referenced it. Only P2 and P20 did not reference Task 5.

[11]The idea that cumbersome indexing disappears in a diagrammatic notation aligns with the rationale of Penrose and Coecke & Kissinger, who designed index-free notations for tensors and quantum mechanics, respectively [12, 52].

(they were not introduced to the Notate interface). Participants generally hypothesized that they would prefer to type code instead. When asked to expand, however, it materialized that their belief stemmed from difficulties conceptualizing how drawing could define abstract circuits (i.e. over *n* inputs), with many assuming a tedious copying of the entire n-ary diagram in Task 6 (e.g., P1: *"[Drawing] by hand?... I think that that would get out of hand so quickly"*). And, when pressed about how to express abstraction visually, some Qiskit participants appeared at a loss for words. They could not imagine how that would be done. However, as a few participants continued talking, they appeared to grow uncertain of the solidness of their boundaries between programming and drawing. P10 for instance, without being aware of the Notate condition or ever seeing the interface, said:

> *[For] one of those abstract ones [circuits]... it'd have to be some like combination of drawing with like, notation, or something... But once you have that, then you're moving back into, like, the 'code territory'... If we were to do this completely, like in a 'no code' way, I'd probably have to completely draw [the circuit], right? But once you start thinking about ways to save time on that, like creating notation to define this abstractness, or the repeatedness... Technically, it's code, right?*[12]

When drawings have notation, they "move into," almost *invade*, the "territory" of code [15]. Since notational programming is, by design, meant to dissolve the "territories" of textual, keyboard-centric IDEs and handwritten drawings/notation, were such systems widely adopted, it is possible that the material assumptions underlying the concepts of "code" and "programming" would shift.

7.4 Limitations of evaluation

Our study had numerous limitations. By far the biggest limitation is that Notate participants were novices, and could not conceptualize how such an interface might fit into a real quantum programming workflow. Future participatory design work with expert quantum programmers may enrich and amend our design. Second, our choice to recruit only those already experienced with Python and Jupyter notebooks may have biased them towards those familiar interfaces [10, 27]. Had we chosen participants with no knowledge of Python and Jupyter, it is an open question how the interfaces would compare.[13] Third, we acknowledge that our comparison is limited to a typewritten API which might be improved in the future (say, to make using custom subcircuits more intuitive), and comparisons to other typewritten APIs may yield different results. Finally, because of our task design, Notate users did not encounter workflows that involved equal ratios of handwritten and typewritten coding. In practice, we envision more equal cooperation between input modalities. Future studies might explore tasks where participants learn both a typewritten API and a corresponding handwritten notation.

[12]During this study, the audio transcript reveals that the interviewer never mentioned the term "notation": the participant came up with it on their own.

[13]There may also be no best design for a comparative study, however. According to Lieberman, comparative studies are only preferable when "changing the variable doesn't change the paradigm of interaction... when the alternatives being tested are radically different from each other, you've got a problem" [43]. Greenberg & Buxton echoed Lieberman's concerns, arguing that running comparative usability studies may (sometimes) be "harmful" for evaluating interfaces that challenge entrenched practices or norms [27]. They outline situations where either the interface is on the cusp of feasibility, but still too prone to errors; or where participants hold strong biases towards existing practices.

8 DISCUSSION

Overall, our findings show that Notate users found our core interaction of implicit cross-context references intuitive. Moreover, all Notate users were able to apply Qaw abstractions to solve tasks 3-5, and most were able to solve the final task, involving double recursive definitions and at least one subcircuit, within the allotted time. In addition, although Notate participants were introduced to an entirely new notation, the slash-wire, in Task 3 –compared to Qiskit participants who used familiar `for` loops –they were able to complete the task significantly faster. This is all the more surprising since, usually, a portion of Notate participants' task time was spent wrangling recognition errors. Comparison of task times for Notate vs. Qiskit conditions also provides evidence for a longstanding contention by programming researchers studying "visual" vs. "textual" notations: that which is "better" depends on the task at hand, how the design of the notation affords or resists encoding a particular solution, and the background and preferences of who is trying to apply it [26, 68]. Taken collectively, these findings support our "heterogenous" vision of notational programming –for designs that mix modalities, instead of demanding one for all time [20, 58]. We now reflect on future directions, rationale behind some design choices, and differences with GUIs.

8.1 Future directions and reflections on process

Our qualitative findings for Notate revealed that, while participants could learn and apply Qaw abstractions, there was much room for improvement to the interface design. Recognition rates, drawing features (such as the lasso tool), task reference material, and especially debugging feedback could all be improved. The infrastructure and standards around typewritten environments –linters, syntax highlighting, debugging feedback, etc. –has evolved over decades, and it stands to reason that handwritten programming could benefit from similar innovations. In Notate, only recognition errors 'threw plots' in their tracebacks, but UX-AI transparency principles [23] suggest that seemingly-semantic errors should also throw plots visualizing what the AI saw. Future work might explore best principles of building debugging toolchains for notational programming.

We also believe improvements can be made to our notation design process. Much of our work was akin to research-through-design, viewing our design artifact as an outcome "that can transform the world from its current state to a preferred state" [73, p. 493] –relating to how participants currently associate "coding" with the keyboard, versus (our intention) broadening this conception to include pen-based input. Here, the *process* of trying to design and implement a notational programming system may itself be a contribution, and offer suggestions for future practice. One improvement could be, before implementation, to run a Wizard of Oz (WoZ) study to examine how participants deploy a notation in practice, with the intent to fold their feedback into a more finalized specification. Another, more futuristic design is to leverage "evolving AI" [70] so that a distributed notational system would evolve its notation in response to how users actually use it, just as a non-computational notation like Feynman diagrams evolved as it came into contact with varied communities [36]. Relatedly, a future paper might center the communication protocol mediating the two contexts, and develop tools and guidelines for helping design domain-specific notations and interpreters.[14] Past work on multi-domain sketch recognition by Alvarado & Davis required a (typewritten) hierarchical shape description language to specify new domains [1]; here we might imagine combining deep learning-based recognition with programming-by-example techniques.

One additional question was raised by the "trick" many Qiskit participants used to solve Task 6 significantly faster than their Notate counterparts: the copying of one solution into a new notebook, followed by a short edit-run-revise cycle. Future designs might support copying Canvas objects not just across notebooks, but facilitate copying only parts of the drawing, or editing a finished drawing. Also, this raises a question of how one might design notations with localized copy-and-paste operations in mind.

8.2 Rationale behind turns of phrase

The astute reader might wonder why we used terms like "notational programming system" throughout this piece, instead of more common phrases like "sketch-based interfaces." Developers of sketching interfaces typically aim to support early-stage design processes and often take a user-centered design approach, aiming to recognize rather than reconfigure existing practice. For instance, Buxton defines sketching as any design process where the output is quick, plentiful, disposable, ambiguous, and with minimal detail [7]. Supporting sketching is an important area, as decades of rich research can attest. However, we preferred to use terms like "notational" and "pen-based" to, in part, clarify that we do not intend notational programming to support or augment early-stage sketching; i.e., it is not a replacement for paper and pen. Rather, notating is intended to be closer to the end product of user's thought processes, more like typewritten code. Said differently, it is our intent that handwritten input requires a certain precision, just as typing on a keyboard does. That does not mean we should dismiss poor recognition rates, but rather –taking principles from AI research –calibrate user expectations appropriately [23].

Another key break with our approach was a focus on notation design and the 'reconfiguration' of user practices present in the PL community but often avoided in the framing of sketch recognition research. Per its name, sketch recognition research tends to proceed from a user-centered design perspective that focuses on recognizing existing notations and design practices (e.g., molecular diagrams, flow charts, or UI sketches), rather than taking a cultural-historical approach which conceives of writing practices as produced through a dialectic between human needs and material affordances [4, 16]. We perceive the goal of notational programming as not only recognition, but asking how we might *reconfigure* and extend existing notations –or indeed create new ones –in dialogue with new computational powers.

[14]For instance, although we have chosen an "implicit" binding protocol here –where a subset of typewritten variable names are implicitly imported into the handwritten context and vice-versa –perhaps some developers may favor more "explicit" bindings, involving passing more arguments to the interpreter.

8.3 Handwriting interfaces vs. GUIs

A recurring question some people have asked, when first hearing of our system, is: "why not use a GUI?" Surely, they reason, an embedded GUI for the case of quantum circuits is preferable to our solution –less prone to ambiguous errors, more immediate, editable, etc. And for certain purposes they may be right.[15] Our design was instead meant to be partly critical, i.e. "design that asks carefully crafted questions and makes us think" [18, p. 58]: we purposely avoided the tight feedback loops present in on-line sketch-based interfaces [57], seeking to challenge participants' norms and values around programming (if a feedback loop isn't required, then, technically, they could be writing significant parts of their program on a piece of paper). Our goal was not universally faster or better products [18, p. 58], but to cause users to pause, to provoke or question their typical "ways of doing" programming.[16] From our interviews, it seems that some participants' understandings of coding were indeed broached or called into question after interacting with our system. Nevertheless, there may be practical benefits to our vision of the notational paradigm that we now highlight: its shifting of front-end to back-end implementations, its altering of the "posture" of programmers, and its fluidity.

First, a single drawing interface (and protocol for reading its contents) means that implementers need not develop, embed, and test a new GUI widget for each new domain. Yes, new interpreters will need to be defined, but those methods need not unduly concern themselves with the particulars of the *front-end* interface. Although today it can be costly and time-consuming to iterate ML models [70], as these processes are streamlined, we can imagine future support tools that ease the process of developing said recognizers, say with transfer learning and few-shot examples.

Second, a GUI often –although not exclusively –assumes the familiar mouse/trackpad and keyboard setup. The way that laptops and desktop PCs constrain the body is often an assumed, and not reflected on, aspect of programming practice. Notational programming may change the paradigm of interaction towards pen-centric input, such that, hypothetically, one could be writing significant parts of a program on an e-Ink tablet. The body (posture, movements, even aspects of cognition) is constrained differently based on these setups, and some of our participants even reflected on this.

Third, and unlike domain-specific GUIs, because the core data for the notational paradigm are images, they can be copy and pasted and passed around at will, using existing, out-of-the-box infrastructure available on all operating systems. The image effectively is the program, or at least part of it. Were there an abundance of community-built interpreters, one could copy images from whiteboards, paper, or online resources, that then load them directly into data structures within a typewritten workflow. Like Mol's characterization of the Zimbabwe Bush Pump, we might say the notational programming interface aims to be a "fluid technology," in that an image affords "a flexibility that allows it to travel almost anywhere" [14, p. 226]. What it sacrifices for a GUI's "firmness," in Mol's terms, it may make up for in versatility and mobility.

[15]For instance, we would encourage IBM integrating Qaw abstractions like the slash-wire into their quantum circuit GUI Composer.

[16]Computer "coding" and "programming" are terms with a relatively recent history; their meanings have never been pregiven, fixed or static but have evolved over time.

9 CONCLUSION

In this paper, we explored a prototype notational programming system embedded in notebook environment. We introduced several principles of notational programming regarding how a host environment (here, typewritten language and IDE) communicates with a pen-based interface for handwritten notation. As a case study, we then developed an abstract notation to writing quantum circuits, Qaw, and built a deep learning-powered interpreter of a subset of the Qaw notation. Interestingly, three Notate participants seemed to assume our in-line canvases shipped as a standard feature of Jupyter notebooks, and nearly all had no trouble grasping the concept of referencing typewritten variables in handwritten code, suggesting that future users would find our core concept intuitive. Such blending between the "textual" and the "visual" might also work towards shifting cultural values and boundary work around what "programming" entails. However, more work is needed on the infrastructure supporting notational programming –debugging tools, and designs that manage or mitigate the mode-switching between keyboard and pen.

ACKNOWLEDGMENTS

The first author would like to thank friends and colleagues whose conversations over the years contributed to the ideas in this paper: Sha Xin Wei, Dietrich Geisler, Kimberly Baum, and Matthew Law.

REFERENCES

[1] Christine Alvarado and Randall Davis. 2007. SketchREAD: A Multi-Domain Sketch Recognition Engine. In *ACM SIGGRAPH 2007 Courses* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 34–es. https://doi.org/10.1145/1281500.1281545

[2] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (Nov 2020), 28 pages. https://doi.org/10.1145/3428290

[3] Georg Apitz and François Guimbretière. 2004. CrossY: A Crossing-Based Drawing Application. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology* (Santa Fe, NM, USA) *(UIST '04)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1029632.1029635

[4] Ian Arawjo. 2020. To Write Code: The Cultural Fabrication of Programming Notation and Practice. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3313831.3376731

[5] Zahra Ashktorab, Justin D. Weisz, and Maryam Ashoori. 2019. Thinking Too Classically: Research Topics in Human-Quantum Computer Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300486

[6] Marat Boshernitsan and Michael Sean Downes. 2004. *Visual programming languages: A survey.* Citeseer.

[7] Bill Buxton. 2010. *Sketching user experiences: getting the design right and the right design.* Morgan kaufmann.

[8] Arnab Chakraborty. 2011. QuECT: a new quantum programming paradigm. *arXiv preprint arXiv:1104.0497* (2011).

[9] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376729

[10] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A Myers. 2021. PLIERS: a process that integrates user-centered methods into programming language design. *ACM Transactions on Computer-Human Interaction (TOCHI)* 28, 4 (2021), 1–53.

[11] Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. 2021. Does the Bronze Garbage Collector Make Rust Easier to Use? A Controlled Experiment. *CoRR* abs/2110.01098 (2021). arXiv:2110.01098 https://arxiv.org/abs/2110.01098

[12] Bob Coecke and Aleks Kissinger. 2017. *Picturing Quantum Processes.* Cambridge University Press.

[13] Randall Davis. 2007. Magic paper: Sketch-understanding research. *Computer* 40, 9 (2007), 34–41.

[14] Marianne De Laet and Annemarie Mol. 2000. The Zimbabwe bush pump: Mechanics of a fluid technology. *Social studies of science* 30, 2 (2000), 225–263.

[15] Gilles Deleuze and Félix Guattari. 1987. A thousand plateaus, trans. Brian Massumi.

[16] Paul Dourish. 2017. *The stuff of bits: An essay on the materialities of information.* MIT Press.

[17] Wenxiao Du. 2012. *Code Runner: Solution for Recognition and Execution of Handwritten Code.* Technical Report. Stanford University. 1–5 pages.

[18] Anthony Dunne and Fiona Raby. 2001. *Design noir: The secret life of electronic objects.* Springer Science & Business Media.

[19] Thomas O Ellis, John F Heafner, and William L Sibley. 1969. *The GRAIL Project: An experiment in man-machine communications.* Technical Report. RAND Corporation.

[20] Martin Erwig and Bernd Meyer. 1995. Heterogeneous visual languages-integrating visual and textual programming. In *Proceedings of Symposium on Visual Languages.* IEEE, 318–325.

[21] Craig Gidney. 2017. Quirk: Quantum Circuit Simulator. https://algassert.com/quirk.

[22] Herman H. Goldstine and John Von Neumann. 1947. *Planning and coding of problems for an electronic computing instrument.* Technical Report. Moore School of Electrical Engineering, University of Pennsylvania.

[23] Google. 2019. People + AI Guidebook. https://pair.withgoogle.com/guidebook/.

[24] Google Quantum AI. 2022. Cirq. https://quantumai.google/cirq.

[25] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. https://doi.org/10.1145/2491956.2462177

[26] Thomas RG Green and Marian Petre. 1992. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, Vol. 57. Citeseer.

[27] Saul Greenberg and Bill Buxton. 2008. Usability Evaluation Considered Harmful (Some of the Time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy) *(CHI '08)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/1357054.1357074

[28] Mark D. Gross and Ellen Yi-Luen Do. 1996. Ambiguous Intentions: A Paper-like Interface for Creative Design. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology* (Seattle, Washington, USA) *(UIST '96)*. Association for Computing Machinery, New York, NY, USA, 183–192. https://doi.org/10.1145/237091.237119

[29] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[30] Tracy Hammond and Randall Davis. 2007. LADDER, a Sketching Language for User Interface Developers. In *ACM SIGGRAPH 2007 Courses* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 35–es. https://doi.org/10.1145/1281500.1281546

[31] Rebecca L. Hao and Elena L. Glassman. 2020. Approaching Polyglot Programming: What Can We Learn from Bilingualism Studies?. In *10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019) (OpenAccess Series in Informatics (OASIcs), Vol. 76)*, Sarah Chasins, Elena L. Glassman, and Joshua Sunshine (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:7. https://doi.org/10.4230/OASIcs.PLATEAU.2019.1

[32] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[33] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925

[34] IBM. 2022. Qiskit Textbook: Learn Quantum Computation using Qiskit. https://qiskit.org/textbook/preface.html.

[35] Gabe Johnson, Mark D Gross, Jason Hong, and Ellen Yi-Luen Do. 2008. Computational Support for Sketching in Design: A Review. *Human–Computer Interaction* 2, 1 (2008), 1–93.

[36] David Kaiser. 2009. *Drawing theories apart.* University of Chicago Press.

[37] DaYe Kang, Tony Ho, Nicolai Marquardt, Bilge Mutlu, and Andrea Bianchi. 2021. ToonNote: Improving Communication in Computational Notebooks Using Interactive Data Comics. In *Proceedings of the 2021 CHI Conference on*

Human Factors in Computing Systems (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 727, 14 pages. https://doi.org/10.1145/3411764.3445434

[38] Laewoo Kang and Steven Jackson. 2021. Tech-Art-Theory: Improvisational Methods for HCI Learning and Teaching. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1, Article 82 (Apr 2021), 25 pages. https://doi.org/10.1145/3449156

[39] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

[40] James A. Landay and Brad A. Myers. 1995. Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '95)*. ACM Press/Addison-Wesley Publishing Co., USA, 43–50. https://doi.org/10.1145/223904.223910

[41] James A. Landay and Brad A. Myers. 2001. Sketching interfaces: toward more human interface design. *Computer* 34, 3 (2001), 56–64. https://doi.org/10.1109/2.910894

[42] Chuanjun Li, Timothy S Miller, Robert C Zeleznik, and Joseph J LaViola Jr. 2008. AlgoSketch: Algorithm Sketching and Interactive Computation. *SBIM* 8 (2008), 175–182.

[43] Henry Lieberman. 2003. The Tyranny of Evaluation. *ACM CHI Fringe* (2003).

[44] James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. 2000. DENIM: Finding a Tighter Fit between Tools and Practice for Web Site Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (The Hague, The Netherlands) *(CHI '00)*. Association for Computing Machinery, New York, NY, USA, 510–517. https://doi.org/10.1145/332040.332486

[45] Microsoft. 2018. Sketch2Code. https://www.microsoft.com/en-us/ai/ai-lab-sketch2code.

[46] Dhawal Mujumdar, Manuel Kallenbach, Brandon Liu, and Björn Hartmann. 2011. Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI EA '11)*. Association for Computing Machinery, New York, NY, USA, 1525–1530. https://doi.org/10.1145/1979742.1979802

[47] Theresa O'Connell, Chuanjun Li, Timothy S. Miller, Robert C. Zeleznik, and Joseph J. LaViola. 2009. A Usability Evaluation of AlgoSketch: A Pen-Based Application for Mathematics. In *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling* (New Orleans, Louisiana) *(SBIM '09)*. Association for Computing Machinery, New York, NY, USA, 149–157. https://doi.org/10.1145/1572741.1572767

[48] Cyrus Omar and Jonathan Aldrich. 2018. Reasonably Programmable Literal Notation. *Proc. ACM Program. Lang.* 2, ICFP, Article 106 (Jul 2018), 32 pages. https://doi.org/10.1145/3236801

[49] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059

[50] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan 2019), 32 pages. https://doi.org/10.1145/3290327

[51] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894

[52] Roger Penrose. 1971. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications* 1 (1971), 221–244.

[53] Mathys Rennela and Sam Staton. 2017. Classical Control, Quantum Circuits and Linear Logic in Enriched Category Theory. *CoRR* abs/1711.05159 (2017). arXiv:1711.05159 http://arxiv.org/abs/1711.05159

[54] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

[55] Mark Santolucito. 2021. Human-in-the-Loop Program Synthesis for Live Coding. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design* (Virtual, Republic of Korea) *(FARM 2021)*. Association for Computing Machinery, New York, NY, USA, 47–53. https://doi.org/10.1145/3471872.3472972

[56] Nazmus Saquib et al. 2020. *Embodied mathematics by interactive sketching.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[57] Nazmus Saquib, Rubaiat Habib Kazi, Li-yi Wei, Gloria Mark, and Deb Roy. 2021. Constructing Embodied Algebra by Sketching. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 428, 16 pages. https://doi.org/10.1145/3411764.3445460

[58] Lucy Suchman. 2002. Located accountabilities in technology production. *Scandinavian journal of information systems* 14, 2 (2002), 7.

[59] Sarah Suleri, Vinoth Pandian Sermuga Pandian, Svetlana Shishkovets, and Matthias Jarke. 2019. Eve: A Sketch-Based Software Prototyping Workbench. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI EA '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3290607.3312994

[60] Ivan E. Sutherland. 1963. *Sketchpad, a Man-Machine Graphical Communication System.* Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

[61] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q# Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. 1–10.

[62] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[63] John Von Neumann. 1993. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing* 15, 4 (1993), 27–75.

[64] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 39 (Nov 2019), 30 pages. https://doi.org/10.1145/3359141

[65] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. 2022. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *ACM Trans. Comput.-Hum. Interact.* 29, 2, Article 17 (Jan 2022), 33 pages. https://doi.org/10.1145/3489465

[66] Danli Wang, Yang Zhang, Tianyuan Gu, Liang He, and Hongan Wang. 2012. E-Block: A Tangible Programming Tool for Children. In *Adjunct Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (Cambridge, Massachusetts, USA) *(UIST Adjunct Proceedings '12)*. Association for Computing Machinery, New York, NY, USA, 71–72. https://doi.org/10.1145/2380296.2380327

[67] David Weintrop. 2019. Block-Based Programming in Computer Science Education. *Commun. ACM* 62, 8 (Jul 2019), 22–25. https://doi.org/10.1145/3341221

[68] Kirsten N. Whitley. 1997. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing* 8, 1 (1997), 109–142.

[69] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 152–165. https://doi.org/10.1145/3379337.3415851

[70] Qian Yang, Aaron Steinfeld, Carolyn Rosé, and John Zimmerman. 2020. Re-Examining Whether, Why, and How Human-AI Interaction Is Uniquely Difficult to Design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376301

[71] Anna Zeng and Will Crichton. 2019. Identifying barriers to adoption for Rust through online discourse. *arXiv preprint arXiv:1901.01001* (2019).

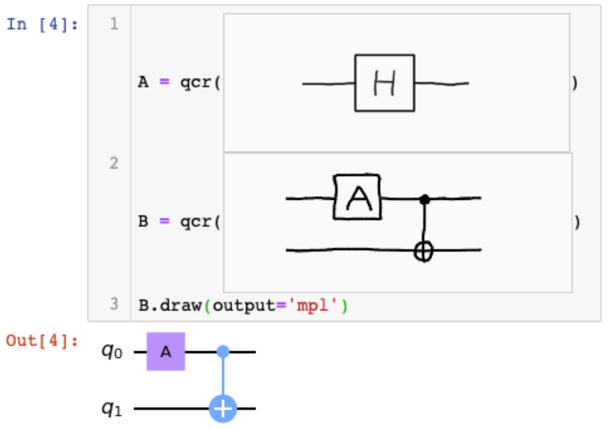
[72] Lei Zhang and Steve Oney. 2020. FlowMatic: An Immersive Authoring Tool for Creating Interactive Scenes in Virtual Reality. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/3379337.3415824

[73] John Zimmerman, Jodi Forlizzi, and Shelley Evenson. 2007. Research through Design as a Method for Interaction Design Research in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 493–502. https://doi.org/10.1145/1240624.1240704

A ELEMENTS OF QAW NOTATION

Here we describe components and operations of the current iteration of the Qaw notation. Since Qaw is handwritten, diagrams have been drawn to represent each operation. In practice, one must implement a suitable interpreter for the notation, conceptually similar to a parser for a typewritten language. Our implementation leverages a combination of deep learning and traditional computer vision techniques, but we omit those details here. This section describes the components that make up quantum circuit notation and how they may be combined to denote larger circuits. For each component we note to what extent it was implemented for the user study discussed in the main text.

A.0.1 **Circuit Definition.** Similar to logic circuits, quantum circuits are built from quantum logic gates (drawn as blocks), connected by wires (lines). Qaw facilitates abstraction by allowing blocks to represent either single gates, or a larger defined circuit. This allows users to build up complex circuits from simpler ones. For instance (draw output produced from Qiskit API):



```
In [4]:
A = qcr(        )
B = qcr(        )
B.draw(output='mpl')
Out[4]:
```

Implementation in study: One can type a variable A, B, or C and set it equal to a circuit, then use that variable in later handwritten gates. Although not used in the tasks, letters D-F and U were also supported. One could not handwrite the left-hand-side and equals symbol, however.

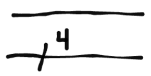
A.0.2 **Connection and Initialization.** The wires connecting gates may represent either traditional bits, or qubits, and are differentiated by their starting inputs. A traditional bit wire starts with circles, while a qubit wire starts with a *ket* ($|0\rangle$). This ket notation is standard in the quantum computing literature, with $|0\rangle$, $|1\rangle$, $|+\rangle$, and $|-\rangle$ being common examples. See Figure 6 for an example. Qaw supports common Greek symbols ψ, ϕ and ω with an expected `complex` type, such that one might type `psi` (or use the symbol) in Python code to initialize a qubit within the handwritten context. If the wire initialized is a bundle (see below), then this value would be repeated over the argument. A list of complex numbers would similarly spread onto the qubits associated with the init wire.

Implementation in study: Not implemented. A version of our multi-object detector parsed kets $|0\rangle$ and $|1\rangle$, although the feature was removed for the user study, to raise recognition rates.

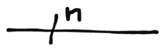
A.0.3 **Measure.** Like in a logic circuit, the value of a traditional bit can be read directly from the wire representing it. But the value of a qubit must be explicitly *measured*. The notation for measure is a stopper to an output, −|. The usual icon for measure is cumbersome to draw, and potentially prone to recognition errors; here, we alter traditional notation for brevity, since measures are common. If one needs to control later gates based on the output of a measure, as in the quantum teleportation circuit [25], they may write −|-|− (double-slash), where the right-hand line now carries a classical bit.

Implementation in study: Not supported.

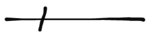
A.0.4 **Wire bundling and bundle types.** Qaw supports bundling wires together with slash notation. Wire bundles are "concrete" if they specify a exact (integer) number for the "size" of the bundle (number of wires):



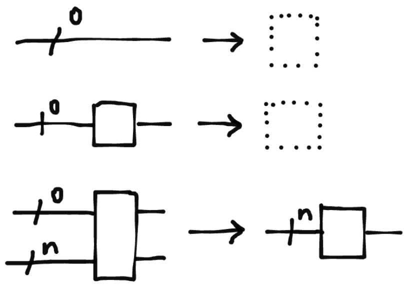
Wire bundles are "abstract" or "variable-sized" if their size is specified with a variable like *n*:



In Qaw, we allow for (and encourage) omitting a parameter name above the slash, where otherwise there would be no ambiguity. The below is equivalent to the above:

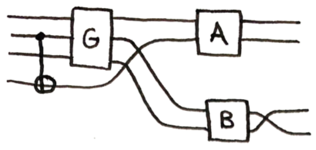


A slash into a common gate such as H, X, Y, or Z "spreads" the gate across all *n* qubits in the bundle, as one may see in Task 3. The spreading performs the tensor product of the matrix, e.g. $H^{\otimes n}$, without the user having to write the cumbersome and redundant $\otimes n$. In addition, for induction purposes, we allow for a "zero-size" bundle that vanishes following special rewrite rules:



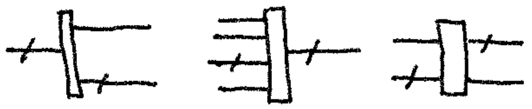
Implementation in study: Support for circuits with at most one input bundle in their input wires. Support for zero-sized bundles upon recursive definition. No support for writing symbols above the slash.

A.0.5 **Swapping wires.** Because the notation is handwritten, wires do not have to be strictly straight lines. Rather, lines can curve and cross as needed to represent swapping wires:

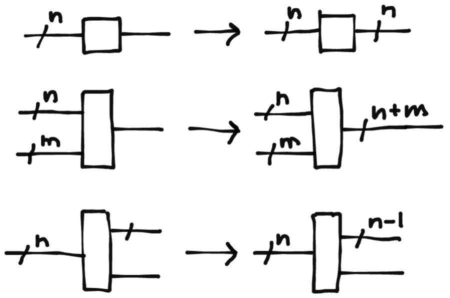


Implementation in study: Supported.

A.0.6 **Splitters, bundlers, and rearrangers.** Much of the power of our notation rests in the combination of variable-sized wire bundles and recursion. Splitters, bundlers, and rearrangers function in a way similar to destructuring lists in functional programming languages (think `hd::tl` in ML-family languages, or `car` and `cdr` in Lisp derivatives). These abstractions are represented by blank gates (identity gate) with *at most one* bundle in their output. From left to right: *splitters* pulling off individual wires from bundles; *bundlers* take a number of input wires and bundle them together; and *rearrangers* combine features of both:



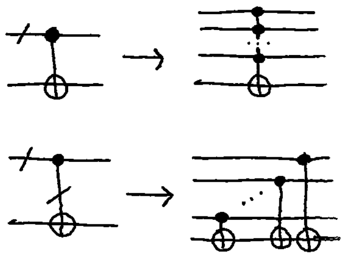
Splitters, bundlers, and rearrangers may be used implicitly in gate definitions. The sizes of the output wires of unitary gates are inferred from the input and need not always be explicitly specified. For example, here are some inference rules for output sizes given the inputs:



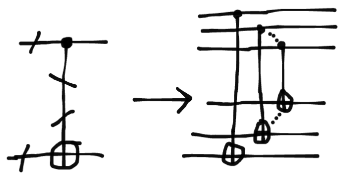
Output sizes can only be inferred in cases where there is no ambiguity. For instance, the last example with two output wires would be in error if the top output wire was not slashed to indicate its abstract size: there would be ambiguity about which output wire was the abstract one.

Implementation in study: Supported for at most one wire bundle in the inputs. Our parser performs size inference checks that threw errors or warnings when a mismatch or ambiguity occurred.

A.0.7 **Control bit patterns.** Quantum circuits often use one bit (or qubit) to control another. This is represented by intersecting the two wires and circling the intersection. These intersections may represent gates that specify exactly how the control operates (e.g. CNOT, Toffoli). The slash notation discussed above allows Qaw to easily generalize control gates to common circuit patterns. For example, we can easily represent higher-order control gates (top), or repeat 2-qubit gates in a staircase pattern by slashing the control line in the direction of the staircase (bottom):

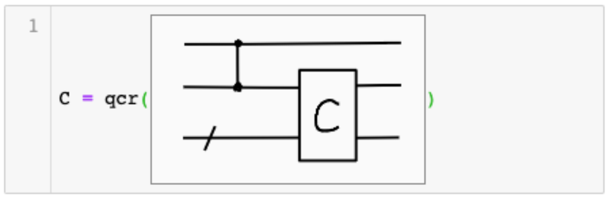


If both wires are bundles, one can specify different staircase patterns by applying multiple slashes on the control line:

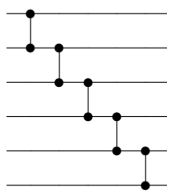


Implementation in study: Basic control gates CX and CZ were supported. Controlling a custom gate `A`, `B`, `C` or common gates H, Z, X, Y, and T were also supported, including for multiple control bits; however, none of the tasks asked for this. Feeding a wire bundle into a CX and CZ gate to represent multi-Controlled Z and X gates was supported. Slashing the control line itself to generate staircase patterns was not supported. This was intentional, as supporting slashing of a control wire might have reduced the difficulty of Task 6.

A.0.8 **Recursive definition and pattern matching.** Circuits can be defined recursively, with base cases identified from pattern matching. The number of input wires to the recursive use of a circuit (right-hand side) must be less than the number in a left-hand-side definition. In the context of a typewritten environment, one can simply type the left-hand-side as a gate name referred to in its definition:


```
C = qcr(                )
```

Note that in the above example, the base case is implicit from the recursive part of the definition, since the bottom wire bundle, as well as the reference to *C*, vanishes when its size = 0. The code above generates the pattern:



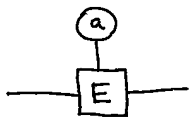
for *n*>1 inputs. In general, for ease of definition we define the *default* base case of any circuit *C* as the identity gate. This default allows the base case to go unspecified in many recursive definitions, saving space and effort.

Implementation in study: Supported, when left-hand-side is a typewritten variable A, B, or C. No current support for handwritten left-hand-sides or alternative base cases.

A.0.9 **Repeat section of circuit k times.** Repetition of subsections of a circuit is indicated by drawing a box around the region one wishes to repeat. Below the box, one may either write a number, or lowercase characters such as k to import values from the typewritten context (Fig. 7). Note that if one writes the size of a bundle, say n, they may also use this inner value elsewhere in the diagram, including in a repeated section (i.e., to make a circuit with n inputs repeat a subsection n times).

Implementation in study: Not supported.

A.0.10 **Controlling gates with classical bits.** One may control gates with classical bits by drawing a circle (unattached to a qubit line) and controlling the gate with a vertical line. Inside the circle one may indicate 0 or 1 or, more interestingly, lowercase alphabetic symbols defined in typewritten code, a, b or c, as 0 or 1 integers. E.g.:



Implementation in study: A version of our implementation supported this operation, but we removed it for the study.

A.0.11 **Reversal / uncomputation.** Uncomputation, an operation often used to clean up ancilla qubits, is indicated by writing a line above a custom gate symbol such as $\mathcal{A}$. The line should not be touching the gate box.

Implementation in study: Not supported.

A.1 Current limitations

Where Qaw succeeds is constructing parts of circuits which are not dependent on classical data (e.g., teleportation). Where it currently struggles or fails is when classical data is encoded into qubit operations (e.g., encoding the oracle in a 3-SAT problem).[17] or when parametrized operations are repeated over an argument, as in:

```python
# Implement an approximate Hadamard
theta = np.pi/np.sqrt(2)
for j in range(n):
    qc.rx(theta/n,qr[0])
    qc.rz(theta/n,qr[0])
```

Future iterations of the notation might address these issues. However, it may also be possible that typewritten coding is the best solution for particular subtasks. By designing notational programming to support "heterogenous" input methods, we intend for users to switch between pen and keyboard interactions where they deem appropriate.

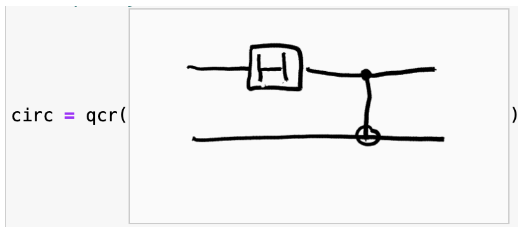
[17] Although we can imagine some extensions. The implementation of certain oracles, e.g. for Grover's algorithm, may be generated from classical functions; thus, in an extension to Qaw, the user might be able to define a classical function f in Python and then reference it in place of the U in Fig. 7.

B TASK RATIONALE AND SOLUTIONS

For completeness, we list our six Tasks with the solutions, written in Qaw notation. We provide rationale for each task, including what concepts it introduces or tests. Each example solution was drawn by one of our participants, depicting a range of styles. Where circuits use abstract notation, we print example output for a specific n inputs.

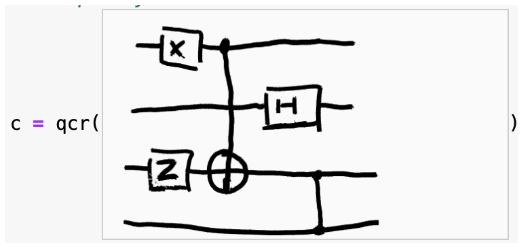
B.1 Task 1

The first task is a simple Bell State, $|\phi^+\rangle$, chosen because it is the "hello world" circuit of quantum computing. Participant P12's solution is simply a copy of the example diagram:


```
circ = qcr(                                  )
```

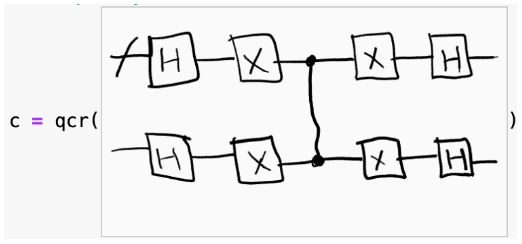
B.2 Task 2

The second task involves copying a more complex "concrete" circuit with four qubits, two control lines and three gates. The rationale was to test the hypothesis that typewritten coding would be faster for such larger concrete circuits, since they may require more time to draw manually. The circuit itself has no particular meaning outside of the task. P4 drew the following:

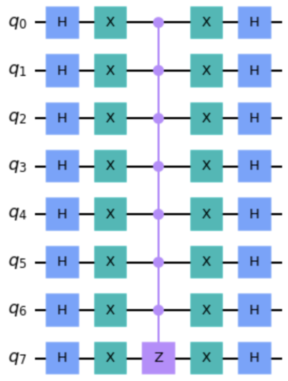

```
c = qcr(                                  )
```

B.3 Task 3

The third task is the generalized diffusion subcircuit of Grover's Algorithm, a famous quantum search algorithm. The diffusion subcircuit was chosen as a vehicle for introducing the slash-wire Qaw notation, as it required the slash-wire to represent arbitrary n inputs, but it was simple in structure enough to not require much more knowledge beyond that. The Multi-Controlled-Z gate in the middle also provided a opportunity to emulate "searching the documentation": in both conditions' task text, we did not explicitly mention or explain this element of the solution. The component was instead included on the reference sheet, so that participants had to use the "documentation" to solve the task. P6's solution:

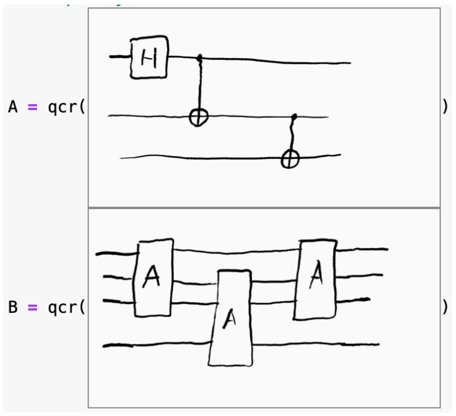


Setting `num_inputs=8`, the notation interpreter prints:



B.4 Task 4

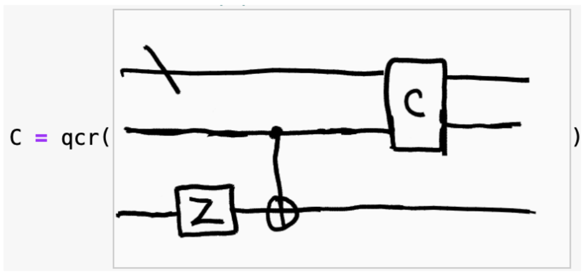
The four task is a circuit that uses a subcircuit, defined in a previous line. The circuit serves as a vehicle to introduce participants to the idea of subcircuits and, in the Notate condition, implicit cross-context referencing. The circuit is simple and has no particular meaning outside the task. For Task 4, P3 wrote:



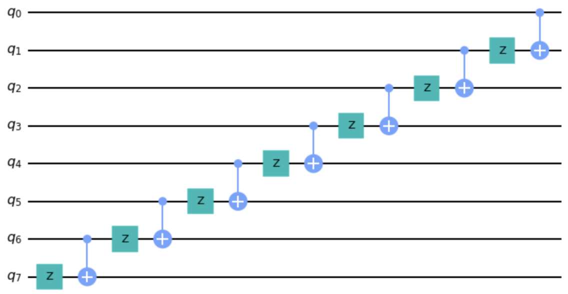
B.5 Task 5

The fifth task asks participants to generate an upward staircase pattern of Z and Controlled-X gates. The pattern had to be defined for n>1 inputs. The rationale was to introduce Notate users to recursive definition, also requiring an implicit cross-context reference. The circuit has no particular meaning outside the task, but was meant to setup the concepts necessary for participants to solve Task Six.

P7's solution:



Setting `num_inputs=8`, the notation interpreter prints:



B.6 Task 6

The sixth and final task asks participants to generate an upward pattern of H and Controlled-Z gates that mirrors the structure of the body of the generalized Quantum Fourier Transform (QFT) circuit for n inputs. The rationale was to provide a harder task for Notate participants that tested every concept they had learned before: gates and control gates, the slash-wire notation, subcircuit definition, and recursive definition. We did not expect all participants to be able to solve this task. P5's solution was depicted in Fig. 9.