Wordplay: Accessible, Multilingual, Interactive Typography

Amy J Ko
The Information School
University of Washington
Seattle, Washington, USA
ajko@uw.edu

Carlos Aldana Lira
Middle Tennessee State University
Murfreesboro, Tennessee, USA
carlos.aldana.lira@gmail.com

Isabel Amaya
University of Washington
Seattle, Washington, USA
iamaya@uw.edu

Abstract

Educational programming languages (EPLs) are rarely designed to be both accessible and multilingual. We describe a 30-month community-engaged case study to surface design challenges at this intersection, creating Wordplay, an accessible, multilingual platform for youth to program interactive typography. Wordplay combines functional programming, multilingual text, multimodal editors, time travel debugging, and teacher- and youth-centered community governance. Across five 2-hour focus group sessions, a group of 6 multilingual students and teachers affirmed many of the platform's design choices, but reinforced that design at the margins was unfinished, including support for limited internet access, decade-old devices, and high turnover of device use by students with different access, language, and attentional needs. The group also highlighted open source platforms like GitHub as unsuitable for engaging youth. These findings suggest that EPLs that are both accessible and language-inclusive are feasible, but that there remain many design tensions between language design, learnability, accessibility, culture, and governance.

CCS Concepts

• **Social and professional topics** → **Computing education**; • **Software and its engineering** → **Development frameworks and environments**; **Compilers**.

Keywords

programming language, computing education

ACM Reference Format:
Amy J Ko, Carlos Aldana Lira, and Isabel Amaya. 2025. Wordplay: Accessible, Multilingual, Interactive Typography. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3706598.3713196

1 Introduction

Programming languages are a foundational way people interact with computers. But, they can only play this role because people *learn* them. From *Pascal* and *BASIC* in the 1970s to the countless research prototypes in the 1980s and 1990s, such as *LogoBlocks*, *ToonTalk*, and *Squeak*, to ubiquitous block-based platforms like *Scratch* and *Snap!*, educational programming languages (EPLs) have been a key resource for enabling this learning [38] over past decades.

Today, however, EPLs have become critical infrastructure for global computing education reform movements in primary and secondary education. Countries worldwide have adopted computer science (CS) learning standards, created teacher learning pathways, developed curricula, and taught classes, striving to broaden participation in computing. These efforts depend tightly on an ecosystem of EPLs and the network of for-profit, not-for-profit, and research teams who maintain them.

While these reform movements have made some progress, they have not been equitable, with schools lacking in funding, most youth lacking access, and pedagogy overlooking diverse learner needs [63]. EPLs, as critical infrastructure, are part of these inequities, determining who can and cannot participate in computing education, and who ends up creating our digital worlds. Two equity gaps in particular, and their intersection, have been broadly overlooked in EPL design:

- Most EPLs are inaccessible to youth who are blind or low vision (BLV), deaf or hard of hearing (DHH), or neurodivergent in that they require youth to be able to use a keyboard, mouse, and/or touchscreen, are not screen-readable, and provide limited flexibility over how programs are presented, executed, explained, or shared.
- Most EPLs are illegible to youth who cannot read English or do not know cultural ideas from Western civilization since syntax, libraries, and documentation are primarily in English and use Western metaphors.

Consequently, many youth are either partly or wholly excluded from learning to code. For example, some disability data estimates that about 15% of students in U.S. public schools have speech, language, or vision impairments [31]. Globally, roughly 10% of youth are fluent in English [50]. If we want future generations globally to have literacy in programmable media, then EPLs must be designed without assuming ability, culture, or language fluency.

The question is how. Prior work has explored support for BVI learners [45, 60]. For instance, Quorum offers a more screen-readable syntax that avoids punctuation, places keywords first to streamline screen readability, and provides sonifications for graphical output [69]. Others have examined how to make code editors more screen readable, primarily by giving structural navigation [5, 21, 48, 49, 49, 61, 64, 67]. Others have explored more accessible tangible and audio output [37, 58].

Prior work has also separately explored supporting English language learners. Some EPLs, for example, have been designed with distinct grammars for multiple natural languages [29]. Some support numerous localizations of their interfaces and keywords [62]. But, most work has focused on instruction and not EPL design.

CHI '25, Yokohama, Japan

ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3713196

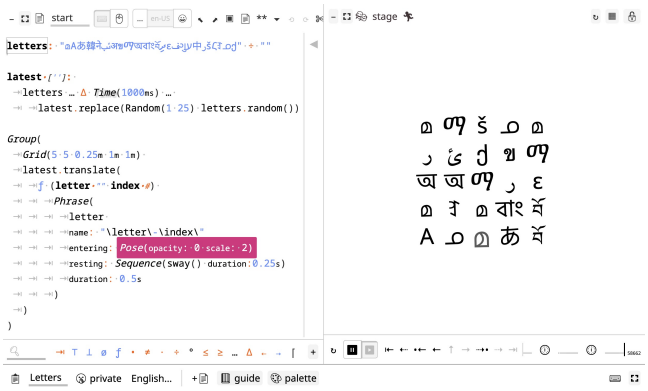

Figure 1: *Wordplay*: accessible, multilingual, interactive typography.

There is evidence that English primacy harms learning [3, 24], and that while instruction in students' first languages promotes learning, engagement, and belonging [2, 20, 22, 39, 46, 52, 66, 75], English primacy in programming and learning materials limits their impact [42].

Little prior work, however, has focused on accessibility *and* language inclusion in tandem, likely due to the tensions between these goals. For example, block-based platforms like Scratch [62] can be rendered in multiple languages but require a pointing device, excluding learners without sight or unable to use a pointing device. Screen-readable EPLs like Quorum [69] carefully design syntax around English to optimize for screen readability, but in doing so require English fluency, or considerable labor and design complexity in curating grammars for every natural language. Creating both accessible and multilingual EPLs is *partly* a matter of engineering, but there may also be design tensions that cannot obviously be addressed through better building. For example, embracing multilingualism can create accessibility barriers, as text-to-speech in screen readers is primarily monolingual and supporting multiple languages in code may increase interface verboseness and complexity.

This raises two questions:

- RQ1: How might EPLs be designed and implemented to be both multilingual *and* accessible?
- RQ2: What design, engineering, and pedagogical challenges arise in designing at this intersection?

To answer these questions, we designed and engineered *Wordplay* (Figure 1), an EPL for creating accessible, multilingual, interactive typography. Our design process involved 30 months of community engagement with teachers and youth, particularly those fluent in non-English languages and/or with disabilities. Throughout our community engagement, our joint goal of multilingual accessibility grew to include an interconnected set of justice-centered goals for *Wordplay* to be *accessible*, *liberatory*, *transparent*, *cultural*, *obtainable*, *democratic*, and *enduring*. Our case, therefore, led to innovations, but also surfaced design tensions that arose in striving for this broader set of community-informed goals.

In subsequent sections, we situate this effort in prior work, describe our design process and *Wordplay*'s design and implementation, and detail technical tradeoffs that surfaced. We then describe youth and teacher critiques from five 2-hour focus groups led by undergraduates, with a group of six teachers, middle school students, and undergraduate students evaluating *Wordplay* against the seven goals above. Our contributions are 1) a novel EPL design that attempts to be both accessible and multilingual and 2) insights into design, engineering, and pedagogical tradeoffs in designing at this intersection.

2 Background

A complete history of EPLs is outside the scope of this paper. Here, we focus on a cross-section of prior work on the design of EPLs, particularly concerning accessibility and language inclusion.

Debates about EPL design have been ongoing for decades. In 1996, for example, McIver and Conway debated introductory EPL design principles [47], contending EPL designs tended to have "grammatical traps," excessive "cleverness," inconsistencies with learners' prior knowledge, and more. In 2005, Kelleher found a broad range of innovations among hundreds of EPLs [38], including systems that provided direct instruction, simplified code writing, alternatives to text, new programming paradigms, and improved visibility of program execution. Stefik and Hanenberg argued EPL discourse focused too much on finding the "best" language rather than recognizing that languages have different purposes and that claims about fitness for a purpose ought to be grounded in evidence rather than ideological stances [68]. Ko et al. complemented these debates, observing that most learning challenges in EPLs stem from algorithm design, finding code to reuse, learning how to reuse it, combining code to reuse, comprehending program failures, and gathering information to debug [40].

While dominant discourse about EPLs has largely neglected accessibility and language inclusion, there has been notable progress on both. For accessibility, examinations into the learning experiences of blind programmers have found numerous accessibility issues with IDEs, including debuggers lacking screen reader support, syntax highlighting not being read, and access barriers only becoming visible after overcoming steep learning curves [4]. Studies with teachers of blind and vision-impaired youth have found that most students needed to use plain text editors because of poor screen reader support, with the exception of Quorum [69][30]. Numerous works have explored new designs of EPL to overcome these limitations, with most focusing on improving the screen readability of IDEs, programming languages, and code [25, 49]. For example, Ehtesham-Ul-Haque et al. offered a screen-readable grid layout for editing conventional code [21], Milne offered an interaction technique for screen reading block-based code [48], and others have described systems for navigating and editing the code structure via keyboard input [5, 64]. Some works have explored systems for speech-based input to support learners with motor impairments [7], audio game design environments [37], and live-coding support for mixed ability groups [55]. Though these works explore support for diverse human abilities, all are in English, and most rely on English grammar and speech technologies to enhance accessibility, ignoring language diversity.

Work on language inclusion has primarily examined pedagogical remedies for EPL designs that are language exclusive [42]. Jacob et al. found that tailoring learning materials to students' language fluency changed their perceptions of CS and their self-efficacy for learning [34]. Vogel found that positioning computing education critically through a translingual lens enabled language-minoritized students to affirm their language practices and resources but also led youth to question the importance of English in computational tools [74]; she also examined how narrowly localizing EPLs overlooks the broader ways that learners translanguage with code, using their language assets to navigate gatekeeping in computing [75]. These findings are supported by large-scale log studies, which show that learning to code in learners' first languages is strongly associated with faster learning [20].

EPL innovations that account for diversity in language fluency are nascent. Early inquiries into how learners use language to express computational ideas [54], for example, were strictly concerned with English grammar. More recent works have focused on the localization of EPLs. Goswami and Pal, for example, explored the translation of programming languages into Bengali [22], and Perera explored automated keyword translation [56]. Hermans has made the most notable progress, designing *Hedy*, an EPL grammatically localized into more than a dozen languages, including right-to-left scripts [29]. She has also examined the technical challenges of supporting a large diversity of grammars and identified 12 aspects of EPLs that can be localized: grammar, keywords, identifiers, numerals, punctuation, reading order, and errors [70]. Finally, Piech and Abu-El-Haija explored the limits of machine translation of code for learning, finding that translation is feasible but with errors and primarily only for generating one-time translations of relatively static content rather than learners' ever-evolving code [57]. Though these explorations suggest how EPLs might support learners' multiple language fluencies through translation from one language to another, none suggest how to support multiple languages at once, or what implications multilingual support might have on screen readability, speech input, or other access supports.

Despite progress on EPL accessibility and language inclusion, to our knowledge, no work examines their intersection: systems that have explored accessibility in depth (e.g., Quorum [69]) have been monolingual, and systems that have explored multilingual learner support (e.g., Hedy) have not prioritized accessibility [29]. The closest work at this interaction is pedagogical: work on Universal Design for Learning (UDL) focuses on teachers' broad needs to support English-language learners and students with disabilities. Israel et al. examine challenges at this intersection in computing education [32] and teachers' struggles to consistently and meaningfully incorporate UDL principles in their teaching [33]. In these works, Israel emphasizes the importance of jointly engaging students' cultures, languages, and abilities and highlights the lack of programming tools and platforms that do, but she does not detail designs that might make such support possible.

Design methods for EPLs often overlook accessibility and language inclusion. Coblenz et al. offer PLIERS, a human-centered process for programming language design focusing on task needs but not ability or language fluency needs [15]. The Cognitive Dimensions of Notations framework [8], used to analytically evaluate prototypes of languages like *C#* [14], *LabVIEW*, and *Prograph* [23], makes no mention of language or ability.

Several justice-centered design frameworks offer high level guidance on design. For example, Harrington et al. offer rich insights into conducting community-engaged research at the intersection of race, disability, and accessibility [28]. Costanza-Chock's *Design Justice* [16] builds upon and critiques participatory and human-centered design paradigms, examining how design can center community needs, values, and agency. Work on youth participation in co-design highlights youth engagement opportunities and the need for ongoing critical reflection about their participation [9].

Given the lack of prior work on the intersection of ability and language inclusion, there are two possibilities this paper explores. One is that creating accessible, multilingual EPL is a small matter of careful design and engineering. The other is that there are wicked problems [17] lurking at this intersection that only become visible when attempting this design and engineering.

3 *Wordplay*: Process, Design, and Implementation

In this section, we describe *Wordplay*'s design and implementation and the process we followed to arrive at its current state. Comprehensive specification is infeasible here[1], so we focus on key design processes, design choices, and emergent tradeoffs to enable readers to implement similar language designs and as context for the community critiques from our five 2-hour focus groups in the next section. We divide our discussion into 1) design process and then design choices about 2) programming language, 3) tools, and 4) community governance.

3.1 Design Process

The project's motivations were eclectic. The first author had personal goals to design, build, and deploy discoveries from her prior research. We also had research goals of jointly exploring accessibility and language. The first author also had advocacy goals for equitable K-12 computing education reform in coalition with students, teachers, and school leaders. Though mixed, the first author felt these motivations were well-aligned with a vision of more equitable programmable media for learning, even though they complicated methodological purity.

There is no one way to design a programming language, and we found many methods helpful. While there is no space to articulate in detail all of the methods and data we used to support our formative design process, summarizing them here will give context for the approaches we used to design Wordplay, helping the reader assess whose voices shaped the platform. From March 2022 to the time of writing this paper (30 months), we have engaged in:

- Syntax and semantics prototyping [15] and evaluation with direct stakeholders.
- Cognitive Dimensions analyses [8] of the closeness of mapping, consistency, error-proneness, hidden dependencies, visibility, and more.
- Expert consultations with EPL designers, including a week-long research retreat.

[1]The platform is available for review at *wordplay.dev*, and implementation and specification at *github.com/wordplaydev*

- Literature reviews on accessibility, translanguaging, PL design, and justice.
- Artifact analysis [72] of accessibility and localization features of popular EPL.
- Following design justice principles [16], community building with mixed-ability, multilingual communities of youth and teachers, including classroom visits, co-teaching, researcher-practitioner partnerships, and student-led open source contribution workflows.
- Designerly ways of knowing [18], supported by aesthetics of parsimony and whimsy.
- Domain expertise [59], primarily the first author's 25+ years of human-centered design, study, and evaluation of programming languages and tools and 15+ years in studying computing education.

The first author was the primary designer and developer of *Wordplay* in its first year, engaging in these design methods and synthesizing perspectives into design choices with community feedback. Her positionality was complex: she is multiracial, gender marginalized, and a community organizer of educators and youth, and these roles shaped the design work throughout the first year, within and beyond the methods above. While many of these encounters were with a global community — primarily experts — most were localized to a region of the United States with racial, ethnic, language, and class diversity, stemming from the region's commitments to serving refugee and immigrant communities. This had tradeoffs: our teacher partners were steeped in the challenges of multilingual learning, and that shaped what we learned, but our insights were also tied to the idiosyncrasies of locally controlled U.S. school districts.

In the second year, this design effort shifted to systematically engage teachers and students. We taught two courses, including a summer high school course on computing and culture to roughly 20 multilingual students of color and a 1-hour unit to two 7th-grade classrooms with multilingual youth. We also organized an ongoing open-source community that has engaged more than 150 high school and college students. Finally, we consulted with 2 middle school teachers (of which, one later participated in the focus group) to explore the complexities of using the platform in schools.

Across this period of design and interaction, these interactions led to multiple redesigns, new language features, localization features, and changes to community governance. They also led to us broadening our goals out from only multilingual accessibility to a set of justice-centered EPL design goals that interact with accessibility and language inclusion:

- **Accessible**. Teachers were unsure how to meet IEP ("individualized education plan") requirements in computer science classrooms with inaccessible EPLs and found that accessible EPLs were not sufficiently multilingual.
- **Liberatory**. Teachers and students reinforced that EPLs should be as much about the "what" and "why" of computation as the "how," enabling students to critically examine what computing should be used for.
- **Transparent**. Teachers reinforced how poorly debugging tools and documentation enabled students to understand how programs evaluate, particularly limiting English-language learners' programming self-efficacy.
- **Cultural**. Teachers lamented how few curricula and tools were focused on students' lived experiences, languages, cultures, and family values.
- **Obtainable**. Teachers worried that platforms were often incompatible with the devices they or their students had access to and that students lacked internet connectivity at home.
- **Democratic**. Students and teachers wanted an ongoing voice in shaping the platform's design.
- **Enduring**. Teachers worried about platforms that were only supported for a few years, leading them to lose any investments they made in instructional design.

Throughout the rest of this section and the later evaluation section, we describe and summatively evaluate Wordplay against these design goals through a focus group.

3.2 Programming Language

This section describes *Wordplay*'s core programming language design choices concerning multilingual accessibility.

3.2.1 ***Decision****: Interactive, accessible, textual output.* In our design process, youth surfaced in many media as culturally relevant, including images, music, and games. These media, of course, are well supported by other EPLs but pose the ability and language inclusion challenges we discussed earlier. One class of media surfaced, however, that did not: text. Youth and teachers described the cultural significance of culturally rooted poetry, song lyrics, passages from books of faith, memetic phrases, and emojis. This led us to explore a programming language that would facilitate multilingual, accessible textual expression in rich, interactive, and typographic ways resonating with sighted youth's interests in graphics and games while remaining straightforwardly describable to BVI learners.

Given this focus, a *Wordplay* program's output is a scene graph of Unicode glyphs. The core output concept is a `Stage`, which is a list of `Phrases` or `Groups` of `Phrases`. Each `Phrase` takes a multilingual text or rich text value, and `Phrases`, `Groups`, and `Stages` have optional styling inputs, like fonts, colors, shadows, transformations, and animations. Each output can be rendered as typographic or text-to-speech output in any language supported by machine translation and speech synthesis. Speech output includes a rich description of styling. Interactive output is essentially a sequence of `Stage` values rendered over time, optionally described. Figure 2 exemplifies these concepts, rendering a sequence of animated lyrics from the Radiohead song *Airbag*, generated by machine translating an English program into Spanish. To date, youth have used this media to create programs such as animated, ethnically rooted lullabies, folk dancing Christmas tree emojis, chatbots that teach non-English languages, animated cacophonies of Arabic onomatopoeia, and word puzzles that celebrate the beauty of the grammatical diversity of youth's first languages[2].

[2]We do not show these examples as *Wordplay* projects are private by default.

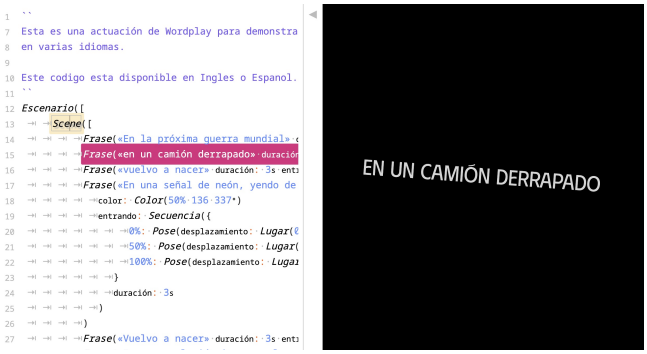

Figure 2: Phrases, Groups, and Stages are used to compose optimally animated and interactive typographic scenes.

```
["cat" "dog" "reptile" "fish" "rodent"].filter(f(pet) pet.has("e"))
```

Figure 3: A *Wordplay* program that finds pet species with the vowel 'e' in them.

While textual output serves ability and language inclusion goals, there are tradeoffs. Youth who want to create images, pixel art, or audio cannot. There are also accessibility limitations in Unicode, such as the lack of non-English descriptions of its 260K+ glyphs that constrain text-to-speech output of symbols to English. Youth can also create visuals as typography, but such compositions also need to be described so the text does not entirely escape the access limitations of visual media.

3.2.2 ***Decision****: Functional style.* One foundational choice in programming language design is how to handle *state*. Should programs be allowed to create and modify variables (the "imperative" style), or should programs only create values from declarative expressions ("functional" style)? After exploring tradeoffs, we decided that *Wordplay* would be purely functional. This choice stemmed from two observations. First, declarative programs "localize" computation, in that all code that affects what is computed is contained within an expression rather than distributed as state modifications across a program's logic. This serves *transparency*, potentially simplifying youths' comprehension of program evaluation, particularly youth with disabilities who may face more significant burdens of code navigation. Second, there was a design intuition that declarative code might better align with the need for description for screen reading. It might simplify localization by reducing the volume of language constructs needing translation.

Being functional, *Wordplay* has no mutable variables or data structures. All values are *immutable*, and computation produces only new values derived from prior values. For example, Figure 3 shows an expression that filters a list of text values to find the pet species names that have "e" in them (evaluating to `["reptile", "rodent"]`).

Though this choice potentially enables these and other benefits described below, it has clear tradeoffs. Learners with prior knowledge of imperative styles will have less knowledge to transfer and even have interfering knowledge. Teacher professional learning

```
"clickety".repeat(1 ... δ Key() ... . + 1)
```

Figure 4: A *Wordplay* program that renders a keyboard emoji the number of times a key has been pressed.

opportunities, including computer science coursework, also tend to use imperative styles, with few exceptions [65], necessitating significant investments in curriculum and teacher learning.

3.2.3 ***Decision****. Reactive expressions.* Functional styles often constrain interactivity because programs respond to user input by changing state. This contradicts many of the project goals of being *cultural* and *liberatory* in that youth usually view computing as interesting only when programs interact with the world [36]. Prior work has addressed this with *functional reactivity* [19, 76], where reactions to input are not event handlers that modify state (e.g., "on click, modify state") but expressions that define the current value of a program based on new input and optional prior values. This treats a program as a recurrence relation on a stream of inputs.

Wordplay implements this with *input streams* and *derived streams*. For example, in the program shown in Figure 4, a program reacts to a `Key` stream of keyboard events and produces a derived stream of the number of key presses that have happened previously. The `δ Key()` expression checks whether that stream has a new value, like an event handler in an event-driven language, and the `1 ... δ Key() ... . + 1` derived stream keeps a running tally of keyboard events. In screen reading, *Wordplay* describes this expression in English as "*reaction, starts one, when key stream changes, previous value plus one.*" Each time the stream changes, the program reevaluates based on the changed streams, producing a new final text value and repeating the word "clickety." This is similar to how a spreadsheet reevaluates, but *Wordplay* programs are a tree of expressions with value streams from sensors, networks, and people instead of a grid of cells.

Wordplay has built-in support for many inputs, including pointer positions and presses, keyboard events, camera sensor frames, microphone amplitude and frequency, `Webpage`s, text from a `Chat` box, collisions between `Phrase`s and `Group`s, and a `Choice` stream, which is a combination of keyboard, pointer, or speech input for selectable graphical content. All of these can be combined with reactions to create rich interactive programs. For example, Figure 5 shows a student example of a video stream that translates a `Camera` stream into ASCII symbols based on brightness, with a dynamic `description` for screen readers.

Reactivity has tradeoffs. One benefit is that reactions are local to what they influence, possibly enhancing *transparency*. This contrasts with imperative event handling, where changes to output are distributed as state changes throughout a program. There is a risk, however, that the concept of "handling" of events may be more closely mapped to how people reason about events in the world as conditions and actions [8].

3.2.4 ***Decision****: Strong, inferred types.* "Strongly typed" means that the types of values must be known and compatible with one

Figure 5: An accessible Unicode video renderer, written by a student.

```
f repeat(text count)
  count > 0 ?
    text + repeat(text count - 1)
    text
repeat('meow' 5)
```

Figure 6: Type inference is necessary in some instances, such as recursion.

another; "inferred" means that *Wordplay* can deduce types without a learner specifying them. For example, the previous section's examples do not specify types; they are implied through literals and function declarations. However, sometimes types must be specified. In Figure 6, for instance, *Wordplay*'s type inference cannot determine the types for the recursive function's inputs, so we must declare them.

Wordplay's type system also supports units on number values and unit arithmetic. For example, a `Time` stream provides a series of new time values at a specified frequency with the unit `ms`. This means that the expression `Time() + 1` is invalid since `1` is missing the unit `ms`. *Wordplay*'s libraries use units like meters to determine distances on `Stage`, degrees for hues in colors, and kilograms for the mass of objects in physics simulations. There are built-in conversions between the world's unit systems, which can be used with the conversion syntax (e.g., `1m ⟶ #ft`).

From an ability and language perspective, this decision has two benefits. First, making types explicit is critical to localization and description of code, reducing ambiguity about meaning. Second, it potentially shifts the work of debugging runtime failures to edit-time type errors, improving *transparency* [27]. The tradeoff is that creators must comprehend *type errors* instead of failures, which can be challenging.

3.2.5 ***Decision**: Symbolic keywords.* One visible tension between EPL design and multilingualism is *keywords*, which are predominantly in English. EPLs offering multiple grammars, like Hedy [29] and Scratch [62], avoid this tension, but such design precludes the use of multiple languages at once. For Wordplay, this raised a critical design question: should Wordplay use words to demarcate programming language constructs, and if so, how might they support multilingualism? If not, what should be used instead of words?

To answer this question, we drew upon work that ties *translanguaging* to decolonization [77]. Translanguaging is the idea that multilingual youth use and combine resources from their many language fluencies to communicate, even when they do not have "complete" mastery of a named language. Wei and García argue that such "weaving together" of language across formal language boundaries is a decolonizing act because it de-centers the language of colonizers and creates *translanguaging spaces* in which youth are affirmed for their creative and constructive use of multiple language assets.

In this spirit, we aspired for Wordplay to be a translanguaging space that encourages such weaving. Using words from a specific language would deter this. However, even using reserved words from languages of learners' choice, especially in multi-lingual combination, risked increasing syntactic complexity, reducing legibility, and de-centering learner-defined names and text in their code. Therefore, we instead chose 15 symbols and symbol pairs from a diversity of natural languages. This created a multilingual symbolic canvas on which to layer learners' own choices of multilingual names and text:

- The `·` to represent a data type declaration, for composite data structures (e.g., `·Cat(name·"" breed·"")`).
- The symbols `ø` for a none literal, `#` for numbers, `[]` for lists, `{}` for sets, `{:}` for key-value dictionaries, and `⊤` and `⊥` for Booleans.
- The `:` represents the binding of a value to a name for later reference by name.
- The `?` to represent conditional expressions, the shorthand syntax `??` for coalescing values that are possibly "none" to some default (e.g., `choice ?? 'default'`, and the `???` as a match expression to allow for conditionals on non-binary values (like a "switch" statement.
- The Dutch florin `ƒ` to declare functions and parentheses `()` to denote the evaluation of a function with a list of inputs. The `()` also represents a block of subexpressions (e.g., `(name: 'Ai' hi: 'hola' "\name\")`).

- The right arrow `→` to represent conversions from one type to another (e.g., `“hello” → []` evaluates to `[‘h’ ‘e’ ‘l’ ‘l’ ‘o’]`).
- The Greek `δ` represents a change predicate on stream values and `...` to denote derived streams.
- The forward slash `/` represents a language tag on a name, a text literal, or documentation and commas to segment them (e.g., `house/en,casa/es`).
- Backticks `` ` `` represent rich text documentation (with a secondary notation for markup omitted here for brevity).

We hoped these specific symbols would realize a translanguaging space. First, they de-center English relative to other EPLs while elevating youths' languages of choice, as the symbols "blend in" with learners' own word choices in code. Second, our symbols come from many colonizers rather than just one, as there are delimiters from English, Latin, Greek, Dutch, Swedish, French, and more. We hoped this diversity of language origins represented an opportunity to translanguage for learners who recognize their meaning and a blank canvas on which to project meaning for learners who do not. This choice potentially imposes comprehension burdens, but it does so for all learners, not just those lacking English fluency, creating a more level field for learning.

3.2.6 ***Decision****: Multilingual text and numerals. Wordplay* also attempts to globalize text and numbers by:

- Supporting text delimiters from all scripts: (e.g., `“”`, `‘’`, `„“`, `„`, `‹›`, `«»`, `「」`).
- Allowing text, names, and documentation to have language-tagged alternatives, as in `"hi!"/en“hola!„/es` and `count/en,contar/es`.
- Supporting numerals from all scripts, enabling mixed-numeral arithmetic like `1 ÷ III`.

These choices permit several language inclusion features:

- *Wordplay*'s libraries and any code that learners write can have any number of names in any written language.
- *Wordplay*'s editor can "skin" code in any language, enabling multilingual classrooms to switch between views of programs in any combination of languages.
- *Wordplay* programs can generate multilingual *output*, where text literals are selected at runtime by preferred language(s) without using a localization library.
- Language metadata on all code and output enables screen readers to choose appropriate text-to-speech engines.

Multilingual support has clear tradeoffs: it complicates syntax and pedagogy by introducing another concept to learn but also helps monolingual youth be aware that the world is multilingual.

3.2.7 ***Decision****: Conflicts, not errors.* Prior work has shown that error messages often do not contain the information learners need to address them and frequently frame the learner as the cause of the error [6]; moreover, errors are predominantly in English. To address these gaps, *Wordplay* frames errors as "conflicts" between language constructs and bug fixes as "resolutions" to the conflict. This rhetorical shift aims to avoid implying blame and frames defects as inconsistencies that only a learner can resolve. For example,

Figure 7: A conflict between a function definition and a function evaluation.

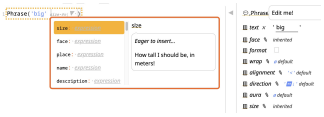

Figure 8: Editing programs via text editing, menus, and a palette.

in Figure 7, the function `square` expects a value, but the evaluate expression `square()` does not provide one. Instead of an error like "*TypeError: square expected 1 arguments, got 0*" when running `square()`, *Wordplay* says: *"I can't evaluate* `square` *without the input named* `num`*"*. This provides the information needed while positioning the conflict as a disagreement only the learner can resolve, similar to prior work [41].

Wordplay has localized versions of all conflicts. Because each message is connected to a particular language construct, they also contain links to all relevant documentation and tutorials.

3.3 Tools and Platform

Wordplay is also a platform. Here, we discuss how our programming language choices interact with the accessibility and localization of its platform features.

3.3.1 ***Decision****: A progressive web application.* A critical decision for accessibility and language inclusion is that *Wordplay* is a web-only, zero-install platform. We chose this primarily for our design goal of *obtainability*: web standards are stable, and browsers increasingly comply with them. No installation means teachers can use it without significant IT approvals or strict hardware requirements. It is also "progressive", meaning most functionality is available offline after download.

3.3.2 ***Decision****: Multimodal editing.* In designing *Wordplay*'s editor, we started from the premise that all abilities must be viable ways to read and write code. This meant supporting input devices that rely on fine motor performance with fingers but also speech, haptic, gaze, and switch inputs. To support these modes, *Wordplay*'s editor represents programs as *both* a text buffer and an abstract syntax tree, building upon prior work on hybrid and multi-modality code editors [44, 78]. This enables several types of input:

- Text editing (see Figure 8), as if the program were a text buffer. This includes insertions, deletions, copy and paste, and autocomplete but with rich visual and auditory annotations. This mode permits syntax errors.
- Block editing (see Figure 9), as in *Scratch* or *Snap!*, but with full keyboard accessibility, a cursor, copy and paste, and OS-level accessibility features, such as speech input and Braille

Figure 9: Editing programs via blocks, drag-and-drop, and copy-and-paste prevents syntax errors but constrains editing.

output. Leaf nodes are text fields or drop-downs, and other nodes are containers representing expressions. Syntax errors are impossible. When switching from text with syntax errors to blocks, errors are represented as a list of unparseable tokens.

- Menu-based editing in both (see Figure 8), where a node in the tree or a position between nodes can be selected, and a set of transformations are shown. This mode is slower but allows switch and gaze input more efficiently.
- Bidirectional editing of typographic output, allowing properties of `Phrase`, `Group`s, and `Stage`s to be edited via GUI. Palette edits immediately update the program and its output; output itself can also be dragged and edited, updating the program bidirectionally.

Unlike most existing editors, all modes are WCAG 2.1 compliant, keyboard accessible, and always available. When a node is selected with a screen reader active, it and its type are described in the selected natural language. When the editor cursor is between nodes, adjacent nodes are defined. The escape key allows navigation to parent nodes, enabling navigation of the program as a text buffer or tree. Because all code is declarative, descriptions of computation tend to more directly describe the purpose of computation than they would in an imperative style.

We have not yet designed native speech input. Large language models may be one resource for enabling this, but there remain questions about what a speech interface would allow to be spoken: tokens would be slow; language constructs would be faster but less reliable, requiring fluid error correction; descriptions of high-level program behavior would be most error-prone and would need to work well across all natural languages.

3.3.3 ***Decision****: Multilingual editing.* *Wordplay*'s editor is locale-aware, leveraging *Wordplay*'s ability to alias names, text, and documentation with language tags. The editor can be shown in "literal" mode, showing all encoded translations in code; "localized" mode (Figure 10), showing code in the selected locales; and "symbolic" mode, preferring emojis and symbol names when available. These modes enable a diversity of multilingual learner and teacher collaborations, a key feature teachers request.

Programs can also be localized on demand. Choosing a source and target language, the editor extracts the program's names, text, and documentation, splits names by camel case and underscores, and then sends the text to a machine translation engine, obtaining "best guess" translations, then repairs any tokenization or syntax errors introduced, and finally embeds the new translation in the program. While the translations are rarely perfect, they allow quick translations for teachers or peers, facilitating multilingual learning.

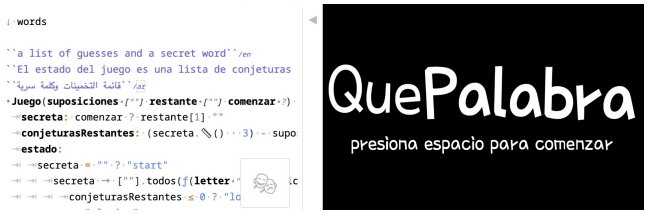

Figure 10: Programs can be rendered in one or more particular locales or with a preference for symbolic names. Here, a program is shown in Spanish, except for the comment the cursor is in.

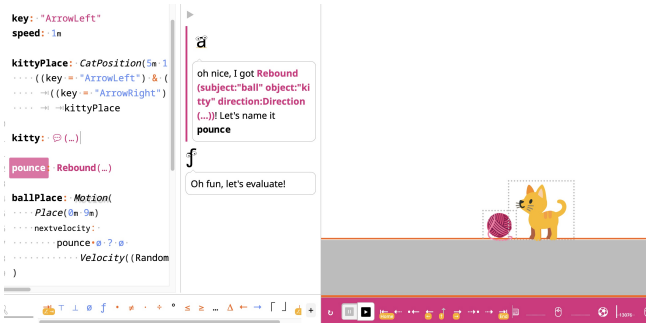

Figure 11: Programs can be paused and stepped forward or backward on demand. Here, the player controls the cat chasing a ball and is inspecting the reevaluation where a collision happened (the "pounce" `Collision` stream was not `ø`).

3.3.4 ***Decision****: Immediate and reversible evaluation.* *Wordplay*'s runtime has three central accessibility and language inclusion features.

The first is control over when programs are reevaluated. By default, evaluation is immediate, reevaluating after any editing pause or input stream changes, updating the new value on stage. Reevaluations are animated in the editor, with input stream expressions "popping" and audibly ticking and active animations highlighted. Immediacy intends to improve *accessibility* and *transparency*. For example, if someone uses a non-visual medium for output, there is immediate feedback about the effect of a program edit as sound or haptics. But, if a learner's attentional needs demand focus or a teacher wants to withhold output for pedagogical reasons, evaluation can be paused.

Second, all steps of program evaluation are described in the current locale. For example, in Figure 11, the assignment of a new `Rebound` value to `pounce` is displayed and screen readable.

Finally, program evaluation is completely and instantaneously reversible to any point of evaluation (see Figure 11). This is possible only because *Wordplay* is purely functional: by remembering input values, any state of a program's evaluation can be quickly recreated. This enables *Wordplay* to support arbitrary scrubbing of evaluation history, including step forward, backward, to a previous input or evaluation of an expression, and to run a program slowly forward and back. This affords direct control over *time*, which allows learners, regardless of their abilities or language fluency, to control when

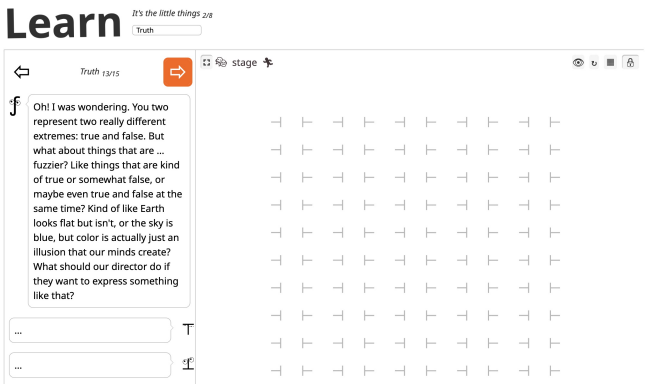

Figure 12: Wordplay supports learning via dramatized, personified tutorials, standalone documentation, and contextual help. Here, the main protagonist, *f*, asks true and false how they think about things that are not exactly true or false. They have no words.

and how program evaluation proceeds, including in reverse if they missed a step or need something repeated, to juxtapose a before and after state, or to switch to a different language mid-evaluation to see a step's explanation in another language.

3.3.5 ***Decision****: Multilingual, multi-paradigm learning. Wordplay* offers three different forms of learning content, all localized and WCAG 2.1 compliant.

The first is a tutorial (Figure 12), framed as a seven-act dramatic play about co-dependency[3], where a troupe of language constructs, each with a distinct personality, work together with a learner to put on typographic plays on stage. The core translation challenge here is finding different cultural metaphors, personalities, explanations, and alternative puns to convey ideas. Teams of multilingual undergraduates maintain these translations, attempting to write at a middle-school reading level in culturally relevant ways, starting from a machine translation.

A second form is a multilingual documentation platform presenting only the facts of the language design. This was created in partnership with autistic students who reported finding the dramatic content distracting, confusing, and unnecessary and wanted the shortest path to the facts.

The third form is contextual help. The editor provides annotations in the margin describing the current node selection or position, linking to documentation or relevant portions of the tutorial to reveal more about the constructs selected or adjacent. This supports learners who wish to tinker and learn in context [13], deferring learning until necessary.

3.3.6 ***Decision****: Sharing is multilingual and moderated.* Youth advocated that *Wordplay* deviate from conventional EPL goals of virality. Learners and teachers can create private galleries of projects, share those galleries with their friends or classes, and limit access to trusted groups. This is done partly in compliance with the global laws on youth privacy (e.g., FERPA, GDRP) but also because of

[3]Astute readers will note the pun.

Figure 13: A *Wordplay* meetup where students network, engage in peer support, and collaboratively design and engineer new features and bug fixes.

student views that their expressions are situated: youth create for their friends, families, and peers, with knowledge of their interests, cultures, languages, and abilities, and so decontextualizing their creation would subvert the socially-situated nature of their creations.

Youth may share their projects publicly but with constraints. The program may not contain anything that appears to be personally identifiable information (PII). Content warnings are shown by default until moderated by a community of trained undergraduate moderators. Finally, there is nowhere to see all public projects on the site: they can be shared by links on external sites only.

3.4 Governance

Wordplay is governed by teachers, students, and undergraduates, facilitated by researchers. In practice, this means more than just making the project open-source. It has meant:

- Forming and striving to sustain a student and teacher advisory council with rotating membership.
- Organizing an undergraduate design and development community with onboarding materials, online community, weekly meetups, and student facilitators. (See Figure 13).
- Cultivating partnerships with secondary CS teachers with multilingual students and students with disabilities.
- Consulting with K-12 teachers at teacher conferences like the CS Teachers Association annual conference.
- Facilitating design discussions in GitHub issues about complex design tensions between youth goals and programming language design constraints.

These have all required defining and communicating project leadership as an ongoing facilitation of co-design to reconcile design tensions, renegotiate power, and mutually teach and learn the community's many forms of expertise.

4 Evaluation

While community input was central throughout our design process, as the platform reached a viable release candidate status, we sought more structured, independent feedback to answer RQ2: what design, engineering, and pedagogical challenges arise? To answer this question, we conducted a structured focus group with six participants, including middle-school youth, teachers, and undergraduates, spanning five 2-hour sessions. Aside from research method guidance, the sessions were entirely designed and run by two undergraduate researchers (the second and third author).

Table 1: Self-reported demographic data.

Participant ID	*Group*	*Ethnicity*	*Gender*	*Languages*
MST1	Middle-school teacher	White	Woman	English, some Spanish
MST2	Middle-school teacher	White	Man	English
MSS1	Middle-school teacher	Indian	Girl	English, Hindi, Punjabi
MSS2	Middle-school student	Indian	Girl	English, Malayalam
MSS3	Middle-school student	(not provided)	Boy	English, Spanish, French, Mandarin
UG1	Undergraduate	Asian	Man	English, some Mandarin

4.1 Participants

We recruited 2 middle-school teachers, 3 middle-school students, and 1 undergraduate who taught or completed computing courses in educational institutions in the metropolitan area of a large city (see Table 1). To recruit, we worked with one middle-school CS teacher participating in the first author's lab as part of a program for engaging teachers in research, and who we had previously consulted on Wordplay's design, but who had not analyzed it in detail. The teacher sent a mass email to parents of students she had previously taught and to a mailing list for middle-school computing teachers; we relied on her established relationships for rapport, freeing youth to critique more openly. We also sent a message to recruit undergraduates to the project's Discord server. All communication reinforced the accessibility and language inclusion goals. We asked all interested participants to complete an interest form, including space for participants to volunteer demographics. Due to the risk of stigma, we did not ask students to disclose disability identities; none were disclosed. We provided snacks, adults were compensated with cash, and youth received gift cards. Experience with Wordplay varied, with some having used it briefly in a class and others having contributed to its design and implementation.

4.2 Sessions

For each session, we devised questions to prompt participants' discussion of EPL and *Wordplay* through the lens of one or two of the justice-centered design goals described above. We additionally devised probes to clarify or further draw out participants' responses. Thus, our prompts guided the session's discussion while our probes afforded exploration of participants' critiques. This highly scaffolded approach helped us adapt to participants' diverse experiences with EPL, multilingualism, and accessibility and better understand their feedback on *Wordplay*. For example, session #4 included the following prompts about *transparency*:

- *Remember when you were confused about what was happening in your code? What made you feel confused?*
- *Think back to when you or a student had trouble debugging code. Looking back on it, what helped you to debug that code?*
- *What would you add to our definition of the term transparent?*
- *Imagine you could create the ideal programming platform to help students understand what is happening in their code. What would that programming platform look like?*

We then asked participants to brainstorm design choices for their ideal EPL on sticky notes. After waiting 10 to 15 minutes, we asked them to read and discuss their choices aloud. Then, we presented *Wordplay*'s design choices and, for each choice, asked:

- *What about this design choice would help you be confident that you knew how your code worked? What are the shortcomings of this design choice?*
- *What about this design choice would help you debug your students' code? What about it would make debugging more difficult?*
- *What about this design choice helps you understand and evaluate the Wordplay language? What are the shortcomings of this design choice?*
- *We are trying to make Wordplay more transparent. Based on what we have talked about today, what have we missed? Is there anything you wanted to say but did not get the opportunity to speak?*

Each session was structured similarly. Between prompts, we presented instructional material about EPL designed to scaffold participants' thinking about accessibility and multilingualism while amplifying their knowledge as insider experts. After each session, we revised prompts according to participants' feedback about the session format. The sessions were audio-recorded and transcribed with participants' consent.

4.3 Analysis

Our unit of analysis was the group's collective critiques rather than individuals. Our primary data included audio transcripts and, secondarily, participants' sticky note brainstorms. The third author's notes from each session supplemented these sources. Following recommendations from Hammer and Berland [26], our goal was not to measure agreement for quantitative analysis but rather generate claims about the data for future investigation. To do this, the facilitators independently and inductively coded the data, identifying potential claims about the group's critiques, and iteratively compared and resolved disagreements. The second author then wrote summaries of the group's design critiques, linking critiques to data. The facilitators then iteratively refined this initial list of critiques until they reached a consensus, resulting in the set of group design critiques below.

4.4 Results

Here we present the group's 13 design critiques, by session, with supporting evidence. We summarize them in Table 2 for reference.

4.4.1 ***Session #1****: Obtainable.* This design goal aims to ensure that learners of any financial means can access EPLs. This means an EPL must work on slow, public, or shared devices and in contexts with slow or unreliable Internet connections. While this session was two hours long, we recorded only the first hour due to a hardware

Table 2: Focus group principles and critiques

Principle: *An EPL should…*	**Critique**: *Wordplay…*
…be usable without internet access	…is only partially usable without internet access
…be available on all devices	…doesn't function equally well on all screen sizes or devices
…be customizable for diverse access needs	…is not customizable enough for all access needs
…be inclusive of neurodivergent learners	…should present information at different levels of detail
…be governed by diverse teachers and students	…needs more diverse youth governance
…minimize barriers to contributions	…should not require GitHub use to contribute design ideas
…be enduring.	…needs sustainable funding without compromising values
…help learners solve it on their own	…errors should teach and empower
…let learners "see under the hood"	…understanding how it is built shouldn't require reading code
…describe "why" a program behaves as it does	…code should be more explainable
…be multicultural	…should have more multicultural examples
…be multilingual	…should support collaboration in multiple languages
…center learners as decision-makers	…should frame learners as shaping the future of computing

failure, so we did not have complete data on the group's evaluation of *Wordplay*.

Participants broadly agreed with the *obtainable* goal.

***Critique**: An EPL should be usable without Internet access.* While participants appreciated that Wordplay could be used partially without internet access, they were concerned that Wordplay would not be fully obtainable without having full offline support. For example, when describing potential barriers to obtaining an EPL, MST1 recalled her teaching experiences in an environment without reliable Internet access, expressing that not all EPLs serve these environments:

> *MST1*: I worked at a school in Thailand where we had Internet for an hour a day, and then it would be gone, and so it was so nice to find options that you could download for offline use where you didn't have to be connected …

Additionally, participants highlighted that *Wordplay* required an Internet connection to start for local use and suggested that the platform be fully downloadable for offline use.

Participants' sticky notes reinforced this view, arguing that EPLs should be "*able to be accessed offline w/o taking a ton of device space*," and provide a "*downloadable version so teacher[s] can run [it] on a local classroom web server*". Other critiques problematized Internet access, such as the critiques that "*project[s] [be] saved offline as a file*".

These critiques reflect the many well known digital divide gaps in education: rural communities globally tend to have no or poor access to the internet [1]. But, they also interact with multilingualism; for example, in the United States, rural communities are increasingly multilingual [53], suggesting that supporting multilingual learners requires accounting for their lack of internet access.

***Critique**: An EPL should be available on all devices.* The group believed that device support should be universal, recognizing that most EPLs require access to a device, but not every learner or school possesses an appropriate device. For example, when discussing potential barriers to obtaining an EPL, MSS1 explained different devices afforded access to different EPL platforms:

> *MSS1*: My old school, we only [had] Apple iPads for elementary and middle and, I think, even up to high-school, I don't really know. But then when I came to [my current school], they had Microsoft ThinkPads, so it was more obtainable to get better programming apps.

Similarly, MST1 shared how schools often do not have access to the newest hardware, restricting access to newer EPL:

> *MST1*: Whenever I see anything on a phone, I'm like, "How compatible is it with older versions of the operating system?" I think there seems to be an assumption that everyone has the newest phone and can upgrade to the newest thing, and in schools, I mean even in well-resourced schools, we're usually keeping the same devices for 4 years, up to, say, 10 years.

In brainstorms, participants suggested strategies for compatibility, affirming that an EPL should be "*accessible to all devices despite the model, age, or company the device is from*" and "*made for laptops/PC but also can be used on all devices.*" Other participants suggested that an EPL be able to "run on browsers/computers with less RAM or computing power" or "*low power [and] old devices*", ensure features are "*independent [of] OS*", distribute a "*cross-platform stand-alone version*", and be "*accessible in browser (for community spaces like libraries).*"

Though these critique suggest EPL should strive for maximal device support, this foregrounds existing digital divide tensions. For example, rural communities tend to be lower income communities, reducing access to modern hardware and software [1]. And yet, many accessibility and language features rely on standards that are supported only in more modern operating systems and hardware.

4.4.2 ***Session #2**: Accessible. Accessible* implies that EPL be usable with any input and output media learners and teachers can perceive and comprehend, including keyboards, pointers, screens, speech input, tactile output, and more. Participants affirmed this goal, but they also questioned how it might be realized pedagogically.

***Critique**: An EPL should be customizable to accommodate a diversity of access needs.* The group viewed customizability as a form

of accessibility when it came to supporting teaching and learning. For example, UG1 argued that an accessible EPL must be flexible in its input-output modalities:

> *UG1*: I think for it to be truly accessible, it has to be able to have some flexibility to switch between these like modes of input and modes of output so that it can actually be customizable by everyone.

Participants elaborated on UG1's critique for classroom management. For example, MST1 viewed customizability as necessary for pedagogy:

> *MST1*: It's great that [an EPL] supports lots of different kinds of input and output, but do they work seamlessly? … We have mixed needs within a group that might be sharing the same computer, so is there a way to like — you know, I'm working with [MSS2], and I want to do mouse input, and now we're pair programming, and it's her turn, and she wants to switch it over to audio input. Is there methods to do that, that are reasonable in a short timeframe that students can make those changes quickly and independently, and they all kind of seamlessly work together?

Drawing from personal experience with accessibility features, MST2 suggested that settings be embedded and configured in the EPL:

> *MST2*: One more thought that I had while you were talking about [the] teacher shared device conundrum. You could develop a scripting language that would be able to set the accessibility settings from a program, and then you can store that in a profile database. And so, the student can come in and load their profile. Boom, and it sets all their accessibilities.

Participants affirmed these points during brainstorming, writing that there should be "*one EPL that can support many needs*," that an EPL provide "*different forms of input + output + supports/customizations [that] work seamlessly*," and that it ensures that "*changing settings adapt[ing] to each person's needs can be done easily*" or that "*settings can be changed to accommodate all needs*," and that it features an "*easily accessible menu for accessibility changes*."

The group's ideas are consistent with leading ideas in accessible computing about ability-based design [79], which argues that interfaces should adapt to people's abilities, rather than having people adapt to interfaces. However, this surfaces hard engineering challenges. For example, some of the customization they mention can be done strictly in the platform, but others might need to be set in the operating system, or in access technology software. Other aspects might be constrained by web standards, which only provide limited configurability, usually to ensure security. There is also the challenge of identity: being able to adjust configurations based on which learner is using a platform might work fine when learners create alone. But, when they work in groups and with share devices, tensions between mixed abilities and language fluencies may arise, requiring new forms of fluid customization.

***Critique**: EPL should be inclusive of neurodivergent learners.* For example, MST1 highlighted that *Wordplay*'s focus on perceptual and motor access appeared to neglect the sensory and communication needs of neurodivergent learners:

> *MST1*: I mean, I think, talking about accepting any form of input or any form of output. When I read that, I think it really applies to physical or perceptual disabilities, but maybe doesn't get at neurodivergence as much. That's less about input and output and more about, you know, settings while you're using the program or supports.

MST2 appeared to affirm and elaborate on the suggested barrier, noting that an EPL's complexity may be overwhelming for learners:

> *MST2*: So, some people just look at something, and they take it all in at once, and other people process it in a sequence, and for programming, if you take it all in at once, you get overwhelmed, you know? And so, not putting too much stuff out there all at once, because it overwhelms you, it's just, you know, it's like a piece of artwork. If you look at artwork, you just take it all in. And I mean, when was the last time you understood a piece of artwork?

During analysis, the facilitators disagreed on whether MST2 affirmed MST1's suggestion or merely commented on how programming may be cognitively challenging for neurotypical learners.

These critiques surface another tension: creating multiple ways for learners to engage, including multiple forms of input and output, multiple language supports, and multiple forms of learning increases complexity, which can be detrimental for students depending on their information processing needs and preferences. For example, several studies on programming show wide variation in preferences for and against learning and problem solving strategies, like tinkering or tutorial-based learning, and that programming environments have embedded bias towards or against these different strategies [11]. There is some evidence that programming environments can be designed to be more universal [12], but such designs have not also tried to be accessible and multilingual.

4.4.3 Session #3: Democratic + Enduring. This session focused on two design goals. The first, *democratic*, was that the development and maintenance of an EPL be governed by and accountable to its stakeholders, such as learners and teachers, and especially to marginalized communities. The second, *enduring*, was that an EPL be available for as long as a community needs it but no longer, recognizing the community and Earth's capacity to sustain it. Participants affirmed both requirements but questioned *Wordplay*'s approach to each.

***Critique**: An EPL should be governed by a diverse community centering on teachers and students.* Participants affirmed the centrality of marginalized stakeholder input. For example, when discussing prior experience of giving feedback to software developers, MSS2, MST1, and MST2 highlighted and questioned individuals' role in filtering and amplifying feedback. MST1 began the dialogue by suggesting open-source communities are demographically hegemonic, suggesting this inhibits democracy. MSS2 then agreed:

> *MSS2*: I think to be democratic you should be, you know what [MST1] said, it should be a diverse amount of people that are looking at email, because a lot of

> people may discount, let's say, what a kid said, or especially what a girl said or stuff like that... And I think that to be democratic, you need to really look at everything, because the company is not all-knowing.
>
> *MST1*: And I think that — sometimes I see when there is a place to make feature requests and stuff, and then you can upvote the ones that you want, and then sometimes they're focused on building the things that kind of rise to the top as the most popularly requested things, but that doesn't always mean that they're the most important things. So if there's only one person, say, I don't know, one person with a very specific perceptual disability… But that doesn't make it less important.
>
> *MST2*: I'm wondering about the use of the word "democratic" and really what that means. Because in a democracy, it's mass force rule by the majority, which is exactly what you're saying, [MST1]. And so by definition, democracy would ignore that one person, because there's only one of them. So, is that what's intended to be used by the word "democratic" here?

MST1 and MST2 also highlighted how the infrastructure and composition of *Wordplay*'s community supports fail to center teachers and students from backgrounds other than computer science:

> *MST1*: So, if [GitHub] is where the design decisions and priorities are being debated and educators or people trained to be educators aren't intentionally brought in, or the community isn't set up in a way that is inviting and welcoming, and makes people want to come in, then I think that that is a problem… So, that could be a way that *Wordplay* could maybe become more democratic in the future.
>
> *MST2*: I would go even farther than that and say that the way this is designed, I think you have a very heavy bias… Just look at this room, look at the personal connections and whatnot of how this feedback session was populated.

This critique was affirmed in participants' brainstorming. Participants wrote that an EPL should "*make sure that communities have a diverse amount of opinions regarding feedback*," "*listens and looks at all feedback regardless of the popularity of who made the feedback*", "*takes into consideration underrepresented voices more*," and ensure that "*power is shared with/centered on teachers + students.*" Others suggested strategies, such as having "*a diverse, well-represented group of people to merge pull requests/master pushes*" or "*a process in place so that diverse feedback & input can be heard and the conversation is not dominated by one or a small group of voices.*"

Most of these choices, of course, were explicit goals of the project; the gaps the group was noticing stemmed from the difficulty of achieving it. For example, MTS1 encouraged the project to engage more educators in the design process. But doing so is far from trivial: in fact, one partner teacher who served a far more language diverse public school wanted to participate and have his students participate, but he did not have capacity, even with funding that we offered to free his time in summer. This mirrors prior work during the COVID-19 pandemic, which revealed how inequities in school funding translate into reduced participation by teachers and students in professional learning, student enrichment, and more [35].

***Critique**: EPL should minimize barriers to contributions.* Participants viewed *Wordplay*'s decision to host its community on GitHub as inhibiting students' and teachers' participation. For example, MST1 judged that GitHub may present accessibility and usability barriers:

> *MST1*: But, I do wonder if also, by all of this taking place on GitHub, if there are perspectives that also get lost there, whereas [MST2] mentioned, maybe there's a place to have more discussion or debate around design decisions and priorities that could be on a different platform that could be less intimidating for teachers, or, say, middle-school students or high-school students or non-technical people.

Participants also identified GitHub's design as a tool specialized for developers as a barrier. UG1, for instance, noted GitHub may not facilitate the kind of contributions stakeholders want to make:

> *UG1*: … [F]or a lot of people, they don't want to contribute by writing code. They want to contribute by just saying what they want it [the EPL] to do. I think having a potential disconnect with GitHub, although GitHub is an extremely useful tool, it's a useful tool for programmers and designers, not necessarily all users.

Others suggested strategies for mitigating barriers to contribution. For instance, MSS2 and UG1 suggested *Wordplay* provide space for feedback on its website. Alternatively, MST2 suggested that *Wordplay* host separate communities for learners and educators and extract actionable input for developers:

> *MST2*: I'm wondering if maybe setting up multiple spaces that are focused for different user groups rather than mixing an educator into GitHub, I mean, just using GitHub is overwhelming… Then come up with an information flow that would pull [feedback] out so it's obvious that the user community needs this set of features, and then take that and move it over to the tech space...
>
> *MST1*: Exactly, yeah. You don't need the technical expertise to necessarily make all the design decisions or decide what the priorities are.

These critiques surface key challenges to implementing democratic community and governance with existing infrastructure for online open source communities. GitHub, of course, is somewhat inescapable from an implementation perspective; addressing it would mean creating an entirely new infrastructure for collaborative software development, centered on teacher and youth participation. The only effort we know of is GitHub Education, which is primarily designed for teachers using GitHub in higher education CS classrooms, not for engaging communities of teachers and learners in open source contributions. Additionally, though *Wordplay* does have a Discord server, teachers and students have not joined,

perhaps for two key reasons: federal laws that deter children's participation without parental approval and teachers' limited capacity to participate in communities, even in professional communities focused on teaching [73].

***Critique**: An EPL should be enduring.* Participants affirmed the *enduring* goal by presenting strategies to fulfill it. For example, MSS2 suggested that an EPL advertise itself and establish a centralized point of funding, similar to the Scratch Foundation:

> *MSS2*: But then also to endure, you need human resources and money. So when the company first starts, if it was my company, it was my community, I would try to get the word out that it exists, and the features that are in it, and what it is... And then, what Scratch did, probably create a foundation, not for-profit...

Similarly, MST1 insisted that an enduring EPL should refuse funding requiring compromise on democratic values, drawing from instances in which low-resource schools collaborate with technology companies to secure financing:

> *MST1*: [Educators] have to discount all their knowledge of their students, their community, their school, their training as an educator to some tech CEO with no training in education in exchange to get the supplies they need, and that model is so widespread and so problematic in so many ways... I would love to see robust, sustainable, enduring funding in place that does not attach, does not compromise those democratic values, and does not take that power of decision-making away from the students, and the teachers, and the underrepresented communities where it should be.

The facilitators presented how *Wordplay* has made technical and engineering decisions to be robust against failing dependencies or changing design requirements. However, when asked to evaluate the decision, participants gave plain, short affirmations or began commenting on subjects ancillary to the decision, such as bugs. This may represent a general difficulty in communicating architectural design choices to stakeholders, as evaluating this choice required expertise most participants lacked.

4.4.4 Session #4: Transparent. This design goal aims for EPL to enable learners to develop knowledge and self-efficacy about how programs execute, possibly by providing clear documentation and debugging tools. Participants affirmed this requirement but expanded its scope and questioned *Wordplay*'s approach.

***Critique**: An EPL should help learners solve it on their own.* Participants judged that a transparent EPL should provide debugging tools, clear error messages, or other material to enable learners to develop self-efficacy and solve errors independently, allowing instructors to help other students. For instance, when discussing the features a transparent EPL should have, UG1 stressed that the platform's debugging tools should not resolve errors for learners:

> *UG1*: The error messages should be descriptive instead of obtusely technical, and they should also preferably give you some kind of course of action or recommend some kind of course of action for you to fix the error. I know IDEs like Visual Studio Code can do that sometimes, but it's a tiny, tiny button under the error message that says "resolve automatically", and it does it for you.

For MSS2 and MST1, it was essential that descriptive errors or debugging facilities enabled a learner to fix errors independently. When discussing debugging tools early in the focus group, MSS2 described how they might free teachers to help other students:

> *MSS2*: Usually I need a teacher sometimes, which is why, personally, I think a tool would be really good just to help you figure it out, because I think also a lot of people I also know freak out when they don't understand something. And I think a tool would also be maybe easier for teachers, because then they could focus on problems that are a lot bigger. If it's a big classroom, like little things and the student just can't find it or something like that because there's a lot of code.

MST1 affirmed MSS2's critique, elaborating that teachers are additionally burdened with scaffolding students' understanding of errors:

> *MST1*: And also it's not like I'm just running around the class being like, "Oh! This is your problem, do this. This is your problem. Do this." If I'm sitting with [MSS3], even if I know what [MSS3]'s bug is, there's going to be a whole process that I'm going to go through with [MSS3] to help him find it himself and understand why.

Additionally, the group suggested *Wordplay*'s debugging tools may be insufficient for fostering self-efficacy. For example, MST1 judged that EPLs should authentically embed debugging tools into curricula to prepare students and instructors:

> *MST1*: And I'm not talking about just, "Here's a 1-page PDF for the teacher on all the debugging tools," and I'm going to stand up there and be very boring, walk through them all with the students... That is the kind of thing that needs to just be authentically integrated into the curriculum so that teachers don't find out a year later that there are debugging tools that they did not know were there.

Participants agreed in their brainstorms, writing that EPL should provide "*curriculum support for teaching debugging processes and available tools*," "*debugging tools that give more than vague error messages but less than automatically fixing it*," "*descriptive error messages that give a course of action to resolve the error*," and "*a tool that can help students find their problems and guide them to the path to fixing and understanding [it].*"

While the group viewed the many innovations in Wordplay's debugging support as important and necessary, they did not view them as good enough for transparency goals. They viewed curricular integrations with these innovations and an explicit focus on debugging skills as central to achieving transparency. This mirrors a longstanding gap in computing education, where debugging is viewed as an essential skill, but rarely taught [43].

***Critique**: An EPL should let learners "see under the hood".* Participants believed that a transparent EPL should enable learners to discover program meaning at their own pace and understand platform semantics by allowing learners to examine how platform-provided tools or functions work internally. For example, when asked to consider moments in which EPL confused them, MSS2 suggested how simple interfaces, though useful for initial learning, may inhibit or complicate students' learning:

> *MSS2*: I wish, I mean, the way that the [EPL platform] worked was fine, I guess, because I guess it was simple... But I wish that there was some way that we could figure out why [EPL platform] worked, if we wanted to. If we had that option, that would be amazing, because then I think a lot of problems wouldn't have happened — that it wouldn't have happened that way. If you're going to make something simpler, at least have a way for students to find out why and how it actually works, how that is being simplified, and stuff like that.

Later in the session, MST2 judged that "seeing under the hood" requires that learners not have to examine the source code of the EPL, appearing to suggest this information instead be provided in the platform:

> *MST2*: So, if [MSS2] is doing programming here, after [her] artificial intelligence experience, she looks at that time thing. Can she open the hood and look inside and see how it works?

SF1 then explained that a user can access *Wordplay*'s documentation within the platform but must examine its source code to understand how the `Time` stream works. MST2 judged that this is insufficient:

> *MST2*: I guess the reason I brought that up is because a lot of people were talking about being able to open the hood and look underneath as a design choice, and maybe that falls a little short of that.

Participants agreed, writing that an ELP should be designed such that "*students can look 'under the hood' as far as they want to*," can "*zoom into specific parts so students can see what is happening at inner levels*," and have "*the option if some parts of the program made simple to see how those simple look.*"

These critiques surfaced an interesting tension: the group understood that *Wordplay*'s source was available and how it's debugging tools could be used to learn how a program worked, but they also felt that *Wordplay*'s source, written at a lower level of abstraction than the interface and level of complexity likely intractable for learners, could not readily answer questions about the platform's behavior, reducing transparency. This suggests a need for interfaces like those in prior work, that answer *why* and *why not* questions in user interfaces but with explanations that are at the level of abstraction of the interface, not lower [51].

***Critique**: An EPL should describe "why" a program behaves as it does.* MST2 expressed this critique and stated that object-oriented PL helped describe the reasons for program statements. When SF1 asked MST2 to elaborate, MST2 explained object-oriented programs provided context:

> *MST2*: Because it gives contextual understanding to what you're doing. Because computers do things to relate to the real world, to do things, you know? Even if it's an abstract thing like math, you can take and put a graph object in there and understand the different representations of, not even physical things, but just conceptual areas of application for software.

By adopting a functional paradigm, MST2 later suggests, *Wordplay* neglects learners who may struggle to relate statements to programs' goals:

> *MST2*: My comment last week about the functional programming nature: I guess functional programming to me is — I'm not a real "math" kind of person. So, I don't see the world through that viewpoint very well. And, so, it's hard for me to understand the context of what this line is building up and how it fits into everything else.

While other participants affirmed MST2's critique, it was not revisited in the remaining discussion, and only 1 sticky note affirmed it. Further, for UG1, an object-oriented paradigm would not always provide contextual understanding, suggesting that the critique may need to be accompanied by other paradigms or features:

> *UG1*: [Object-orientation] makes sense from a conceptual level when you're using it. But, I think from the implementation, like the implementation details, you still have the same problem where it's like, "But how? How is the object implemented?" Because, at some point, you're going to have to reach that bottom level of abstraction and use… I don't want to say math, but kind of base features, base language features.

These critiques were fascinating in that functional styles are intended to be more explainable than imperative and object-oriented styles because everything that the program computes is contained in one place. Some in the group viewed the functional paradigm as lacking other forms of explainability, such as the conceptual alignment between objects in the world and objects in object-oriented programming. This tension also recalls work from linguistic anthropology, which link language and identity [10], suggesting that Wordplay's reliance on symbolic keywords signaled a connection to mathematics, which some in the group felt would necessarily be less "explainable."

4.4.5 Session #5: Cultural + Liberatory. The final session focused on two design goals: *cultural*, ensuring EPLs respond to, sustain, and center learners' culture, values, and identities in how they are designed, explained, and framed, and resist cultural and linguistic hegemony; and *liberatory*, ensuring EPLs empower learners to reshape themselves and the world. Participants affirmed both requirements but judged that *Wordplay*'s decisions conflict with *cultural*.

***Critique**: An EPL should be multicultural.* Participants agreed that EPL should be multicultural, insisting that EPLs embed multicultural or culturally responsive content in its tutorials, documentation, or curricular support. This was rooted in participants'

multicultural and instructional experiences. For MSS2, cultural inclusion signaled acceptance and helped broaden participation in computing:

> *MSS2*: With a lot of languages, some people may even be more drawn towards the website just because they're trying to include that person, no matter how small the minority is that speaks that language. Kind of adding on, I know that the Khan Academy also, when they have questions, they use names from different cultures all around the world, which seems really amazing.

For MST1, however, multiculturalism needed to be accompanied by curricular support for computing teachers, as many lack training in cultural responsiveness or may struggle to take advantage of cultural EPLs:

> *MST1*: My background is more strong in education than it is in computer science, and I still felt like, "Okay, I think *Wordplay* is really cool. I want to use it with my students, but I don't really know how to start." ... You just need something as a teacher to go on. So my platform would include that kind of stuff for teachers to have a starting place.

Participants agreed in brainstorms, writing that EPLs should "*provide multi-cultural [sic] examples using diverse speakers, environments, and application[s]*", include "*curriculum + training resources for teachers (in both culturally responsive teaching + using the CS tool)*", "*center program on cultural ideas and languages based on scenarios*", and be "*representing different cultures.*"

These critiques mirrored critiques of Wordplay's debugging support: it was taken as a given that support for multiple languages at once, in code and output, was inherently good, but such support was also viewed as insufficient without curricular efforts to reap the benefits of these innovations. As with debugging, the group pointed to the need for robust curriculum and pedagogy support to make use of those multilingual features, reinforcing prior work on multilingual translanguaging in computing classrooms [75].

***Critique**: An EPL should be multilingual.* Participants agreed that EPL should be multilingual, enabling learners to write and translate programs in many languages while preserving culture- or language-specific ideas. For example, when discussing the importance of language inclusion in classrooms, MSS1 judged that a multilingual EPL would enable non-English learners to focus on writing rather than translating code:

> *MSS1*: ... [T]hey already have a passion for technology, and then we mainly use text-based coding, but they can't understand exactly what they're coding. So, if their language was included, it would be, yes, very convenient, so they wouldn't have to struggle so much when it came to just translating instead of just writing the code.

While participants supported *Wordplay*'s overall commitment to multilingual code, they also identified how *Wordplay*'s design choices conflicted with this critique. When discussing *Wordplay*'s choice not to use natural language keywords, for example, MSS2 highlighted *Wordplay*'s use of another language's symbols in its syntax and how it may confuse learners:

> *MSS2*: It's a crazy old language, but "T" and the upside down "T" are actually letters, like vowels, so that might be slightly confusing.

Similarly, when discussing *Wordplay*'s choice to not use keywords and enable translation, MST2, and MST1 suggested *Wordplay*'s symbol selection and dependence on multilingualism continue to privilege English:

> *MST2*: Are those [symbols] universal in all languages? Probably not. Do those translate as well? The function [f] and the theta [θ], and whatever the other ones are? ... I was thinking this is used for your function symbol, and that's a well-known Western symbol. So, doesn't that provide a little bit of privilege? The choice of that?

Shortly after, MST1 reaffirmed that *Wordplay* may continue to privilege English:

> *MST1*: I think there are ways that it still privileges students who know English as well, because to be able to share the full expression of their project with a teacher or peers who do not speak their native language, they have to be able to provide the translations.

Participants concurred in brainstorms, writing that an EPL should "*allow the student to work in their own language/cultural perspective [and] not just translation of words*", include "*supports for teachers with classes of students of many linguistic backgrounds*", provide "*features for collaboration in multiple languages [and] not coding in isolation*", "*use as many languages as needed*", "*be able to explain Western ideas*", include "*multiple languages[,] even ones that aren't spoken by very many people*", and include "*language-specific documentation (as in translatable documentation).*"

The group's debates about English colonization mirror the acknowledged complexities of translanguaging as decolonization [77]: embracing and affirming multiple languages is necessary but not sufficient for any project of decolonization; English is still privileged in the world, and Wordplay cannot escape that. Additionally, trying to overcome that privilege comes with costs: just as multilingual classrooms can involve more effort, confusion, and tensions between language learning and subject learning [71], multilingual EPLs may impose similar costs.

***Critique**: An EPL should center learners as decision-makers.* Participants agreed that a liberatory EPL should center students as current and future decision-makers in design, social, and political contexts. For example, when asked to consider the significance of computing, MST1 highlighted the role learners may play as citizens:

> *MST1*: So, even the students who, let's say, I can see into the future, and I know that these students are specific students are never going to do computer science after my class, that's fine. ... But, they're still going to be citizens, they're still going to be voting. They're still going to be participating in society. They're still going to be impacted by technology and so I think part of it, too, is exploring why these things are, why this change is happening, why these things are being

> created, questioning "Should these things be created at all?"

In the same discussion, MSS2 highlighted computing's ubiquity and the necessity for learners to be informed decision-makers:

> *MSS2*: I think that computer programming is going into everything that we do. I think, actually, it's already in a lot of what we do, a lot of the jobs that are already there, like doctors, lawyers. It's gonna be implemented almost everywhere... So I feel like everyone needs to know what it is and know, like [MST2] said and like [MST1] said, so that they can make informed decisions, and also that they know how it works.

When discussing what features a liberatory EPL should have, MST1 insisted that the accompanying platform or curriculum center learners' role as decision-makers in program design:

> *MST1*: I think for me, that's a lot of platforms or curriculums are like, "Everyone is going to make this exact same program, and whoever can replicate it the most exactly as the computer gets the A," versus centering the student. I'm like, the way it's set up often communicates to the student that the computer is in charge, the computer already knows the right way to do it, and you've got to prove that you can do it as well as the computer, and that's what computer science means, versus tools that really communicate that, "All of this whole language, is totally useless without you, the human who is making the decisions."

In brainstorms, participants wrote that EPL should help learners in "*selecting the tools, approaches, [and] project based on purpose (the 'why' comes before the doing)*," center "*the students as the driver + decision maker[,] not the tool*", "*teach all implications of coding in many jobs*," and acknowledge "*the social impact part of code and making sure it exists and is explained to kids in the program*." Others wrote that *Wordplay* should help learners imagine novel programs, such as by "*includ[ing] a way for learners to incorporate their imagination into the process of envisioning and creating their product*" or including "*imaginable example projects that illustrate the power of programming*."

The group's final critiques were more a global critique of EPLs as they are used in teaching: they envisioned a future in which learners have agency over their learning, agency over what computer science means, and agency over what they express and how they are assessed on it. These were less critiques of *Wordplay* itself, but more imagined futures in which it and other EPLs might be used in liberatory ways.

5 Discussion

We return to our two questions:

RQ1 *How might EPLs be both multilingual and accessible?* Our attempt to design such an EPL revealed that designing an accessible, multilingual EPL requires not just tool support, but coordinated choices across syntax, semantics, and output that designing for either in isolation does not. *Wordplay* offers one novel approach, using symbolic keywords; multilingual names, text, output, documentation; multimodal and localized editors and tools; precise, reversible control over program evaluation; and a functional, reactive style that not only enables these features, but also potentially enables better descriptions of code and output. In our design process, we also found many alternatives, especially around syntax, language paradigm, and user interface design, each with tradeoffs.

RQ2 *What challenges arise in designing at this intersection?* The choices above have numerous tradeoffs. Youth and teachers affirmed many of them — particularly being web-based, screen-readable, partly usable offline, and uniquely multilingual — but also noted several gaps as they imagined ideal futures. Summarized earlier in Table 2, their critiques ranged widely, including gaps in offline access, device support, neurodiversity needs for information organization and presentation, funding, content, collaboration and teaching supports, and most importantly, youth's ability to contribute to the project in ways that accounted for their positionality and prior knowledge. All of the critiques surfaced — sometimes explicitly in the focus group, sometimes implicitly as the authors examined the group's critiques — were less about tensions between accessibility and language and more about tensions about complexity, infrastructure, governance, and funding that achieving both might require. For example:

- Youth and teachers wondered if an EPL be sufficiently customizable for language *and* access needs without being overwhelming teachers and students with settings, modes, and other interface complexities that differentiate its use.
- They wondered if an EPL could be governed by diverse youth and teacher voices, but in a way that could reconcile the many unknown conflicts between languages, access needs, and their intersections.
- They wondered about the tradeoffs of imperative and functional styles, and their comprehensibility, both from a language and accessibility perspective.
- They wondered whether aspiring for a universal design might make space for everyone, but in the process, might make Wordplay harder for *everyone* to use.

These many tensions emerging from our design and evaluation challenge the feasibility of universal design [32]. Future work might find through further inquiry that they are reconcilable — but our findings show that even *perceived* tradeoffs can be a barrier to adoption, especially relative to the status quo.

Our findings also reinforce the importance of accessibility and language inclusion through EPL *ecosystems*. A teacher might, for example, have some youth use *Wordplay*, and others use *Quorum* or *Scratch*, depending on their particular goals and language and access needs. This already happens within the limits of teachers' capacity to support multiple platforms, but future work might explore how to support teachers in teaching across EPLs to meet diverse student needs, or even how to make EPL more malleable, and therefore responsive to youth and teacher needs.

The significant cost of creating *Wordplay* suggests the importance of focusing the limited capacity of the research community on what parts of the accessible, multilingual EPL design space to explore. Our case study suggests some points worth prioritizing, including imperative EPLs that embrace natural language grammar

variations, like *Hedy* [29], but are also rigorously accessible, and imperative EPLs that are accessible, like *Quorum* [69], but rigorously multilingual. Such explorations might clarify the tradeoffs of imperative styles accessibility, language inclusion, and transparency.

These findings suggest that not only is jointly designing accessible, multilingual EPL rife with design tensions but also that these tensions are inseparable from other considerations. Designing EPL for these qualities means accounting for the digital divide, for culture, for the ways that teaching and curricula must be integrated with tools, and for youth desire for agency in their learning. EPLs are not necessarily central to all of these concerns, but they are part of them, and ignoring them in EPL design only contributes to the friction found in ability diverse [4] and language diverse classrooms [75].

While much work is ahead, our findings also suggest that progress is possible, even if incremental. And our results suggest many ways forward to disentangle these tensions between design, resources, and diversity. Having many EPLs striving for different dimensions may help. Imagining open source communities, infrastructure, and governance models designed explicitly for teachers and learners may help reduce barriers to representation in governance. Identifying public resources to sustain EPLs that are overseen by representative communities of youth and teachers may relieve tensions around platform needs and endurance. And further exploring the design space of EPLs that strive for different notions of justice may reveal interaction paradigms that are more universal.

Whatever these future works teach us, this present work shows that teachers and youth broadly concur with a justice-centered visions of EPLs and can play a vital role in shaping innovations that help realize them. We hope this case study and its findings are one step in helping imagine those more equitable platforms, playing just one part in addressing the inequities in efforts to broaden participation in computing globally.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 2318257, 2137834, 2137312, 2100296, 2122950, and 2031265, as well as unrestricted gifts from Microsoft, Google, Adobe, and the citizens of Washington state in the United States of America. We thank all of the focus group participants for their time and thoughtfulness, the hundreds of contributors to the Wordplay design, implementation, and localizations, and the University of Washington for it's support of the first author's paid sabbatical leave, which supported this work.

References

[1] Arfa Afzal, Saima Khan, Sana Daud, Zahoor Ahmad, and Ayesha Butt. 2023. Addressing the digital divide: Access and use of technology in education. *Journal of Social Sciences Review* 3, 2 (2023), 883–895.

[2] Suad Alaofi and Seán Russell. 2022. A validated computer terminology test for predicting non-native english-speaking CS1 students' academic performance. In *Proceedings of the 24th Australasian Computing Education Conference*. 133–142.

[3] Hend Alrasheed, Amjad Alnashwan, and Ruwayda Alshowiman. 2021. Impact of English proficiency on academic performance of software engineering students. In *Proceedings of the 2021 4th International Conference on Data Storage and Data Engineering*. 107–111.

[4] Catherine M Baker, Cynthia L Bennett, and Richard E Ladner. 2019. Educational experiences of blind programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 759–765.

[5] Catherine M Baker, Lauren R Milne, and Richard E Ladner. 2015. Structjumper: A tool to help blind programmers navigate and understand the structure of code. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 3043–3052.

[6] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education* (2019), 177–210.

[7] Andrew Begel and Susan L Graham. 2006. An assessment of a speech-based programming environment. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 116–120.

[8] Alan Blackwell and Thomas Green. 2003. Notational systems–the cognitive dimensions of notations framework. *HCI models, theories, and frameworks: toward an interdisciplinary science. Morgan Kaufmann* 234 (2003).

[9] Leanne Bowler, Karen Wang, Irene Lopatovska, and Mark Rosin. 2021. The meaning of "Participation" in co-design with children and youth: relationships, roles, and interactions. *Proceedings of the Association for Information Science and Technology* 58, 1 (2021), 13–24.

[10] Mary Bucholtz and Kira Hall. 2004. Language and identity. *A companion to linguistic anthropology* 1 (2004), 369–394.

[11] Margaret Burnett, Scott D Fleming, Shamsi Iqbal, Gina Venolia, Vidya Rajaram, Umer Farooq, Valentina Grigoreanu, and Mary Czerwinski. 2010. Gender differences and programming environments: across programming populations. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*. 1–10.

[12] Margaret M Burnett, Laura Beckwith, Susan Wiedenbeck, Scott D Fleming, Jill Cao, Thomas H Park, Valentina Grigoreanu, and Kyle Rector. 2011. Gender pluralism in problem-solving software. *Interacting with computers* 23, 5 (2011), 450–460.

[13] John M Carroll and Mary Beth Rosson. 1987. Paradox of the active user. In *Interfacing thought: Cognitive aspects of human-computer interaction*. 80–111.

[14] Steven Clarke. 2001. Evaluating a new programming language.. In *PPIG*, Vol. 13. 275–289.

[15] 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.

[16] Sasha Costanza-Chock. 2020. *Design justice: Community-led practices to build the worlds we need*. The MIT Press.

[17] Richard Coyne. 2005. Wicked problems revisited. *Design studies* 26, 1 (2005), 5–17.

[18] Nigel Cross. 1982. Designerly ways of knowinhammer. *Design studies* 3, 4 (1982), 221–227.

[19] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices* 48, 6 (2013), 411–422.

[20] Sayamindu Dasgupta and Benjamin Mako Hill. 2017. Learning to code in localized programming languages. In *ACM Learning@Scale*. 33–39.

[21] Md Ehtesham-Ul-Haque, Syed Mostofa Monsur, and Syed Masum Billah. 2022. Grid-coding: An accessible, efficient, and structured coding paradigm for blind and low-vision programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–21.

[22] Bishnu Goswami and Sarmila Pal. 2022. Introduction of two new programming tools in Bengali and measurement of their reception among high-school students in Purba Bardhaman, India with the prototypic inclusion of a vector-biology module. *Education and Information Technologies* (2022), 1–23.

[23] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

[24] Carmen Nayeli Guzman, Anne Xu, and Adalbert Gerald Soosai Raj. 2021. Experiences of Non-Native English Speakers Learning Computer Science in a US University. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 633–639.

[25] Alex Hadwen-Bennett, Sue Sentance, and Cecily Morrison. 2018. Making programming accessible to learners with visual impairments: a literature review. *International Journal of Computer Science Education in Schools* 2, 2 (2018), 3–13.

[26] David Hammer and Leema K Berland. 2014. Confusing claims for data: A critique of common practices for presenting qualitative research on learning. *Journal of the Learning Sciences* 23, 1 (2014), 37–46.

[27] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* 19, 5 (2014), 1335–1382.

[28] Christina N Harrington, Aashaka Desai, Aaleyah Lewis, Sanika Moharana, Anne Spencer Ross, and Jennifer Mankoff. 2023. Working at the intersection of race, disability and accessibility. In *proceedings of the 25th International ACM*

SIGACCESS Conference on Computers and Accessibility. 1–18.
[29] Felienne Hermans. 2020. Hedy: a gradual language for programming education. In *Proceedings of the 2020 ACM conference on international computing education research*. 259–270.
[30] Earl W Huff Jr, Kwajo Boateng, Makayla Moster, Paige Rodeghero, and Julian Brinkley. 2021. Exploring the perspectives of teachers of the visually impaired regarding accessible K-12 computing education. In *Proceedings of the 52nd acm technical symposium on computer science education*. 156–162.
[31] Véronique Irwin, Ke Wang, Julie Jung, Tabitha Tezil, Sara Alhassani, Alison Filbey, Rita Dilig, and Farrah Bullock Mann. 2024. Report on the Condition of Education 2024. NCES 2024-144. *National Center for Education Statistics* (2024).
[32] Maya Israel, Latoya Chandler, Alexis Cobo, and Lauren Weisberg. 2023. Increasing access, participation and inclusion within k–12 cs education through universal design for learning and high leverage practices. *Computer Science Education: Perspectives on Teaching and Learning in School* 115 (2023).
[33] Maya Israel, Brittany Kester, Jessica J Williams, and Meg J Ray. 2022. Equity and inclusion through UDL in K-6 computer science education: Perspectives of teachers and instructional coaches. *ACM Transactions on Computing Education* 22, 3 (2022), 1–22.
[34] Sharin Rawhiya Jacob, Jonathan Montoya, Ha Nguyen, Debra Richardson, and Mark Warschauer. 2022. Examining the what, why, and how of multilingual student identity development in computer science. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–33.
[35] Ute Kaden. 2020. COVID-19 school closure-related changes to the professional life of a K–12 teacher. *Education sciences* 10, 6 (2020), 165.
[36] Yasmin B Kafai and Quinn Burke. 2014. *Connected code: Why children need to learn programming*. MIT press.
[37] Shaun K Kane, Varsha Koushik, and Annika Muehlbradt. 2018. Bonk: accessible programming for accessible audio games. In *Proceedings of the 17th ACM conference on interaction design and children*. 132–142.
[38] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM computing surveys (CSUR)* 37, 2 (2005), 83–137.
[39] Taj Muhammad Khan and Syed Waqar Nabi. 2021. English versus native language for higher education in computer science: A pilot study. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*. 1–5.
[40] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.
[41] Michael J Lee and Amy J Ko. 2011. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*. 109–116.
[42] Yinchen Lei and Meghan Allen. 2022. English language learners in computer science education: A scoping review. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*. 57–63.
[43] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*. 79–86.
[44] Yuhan Lin and David Weintrop. 2021. The landscape of Block-based programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages* 67 (2021), 101075.
[45] Kelly Mack, Emma McDonnell, Dhruv Jain, Lucy Lu Wang, Jon E. Froehlich, and Leah Findlater. 2021. What do we mean by "accessibility research"? A literature survey of accessibility papers in CHI and ASSETS from 1994 to 2019. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–18.
[46] Raina Mason and Carolyn Seton. 2021. Leveling the playing field for international students in IT courses. In *Proceedings of the 23rd Australasian Computing Education Conference*. 138–146.
[47] Linda McIver and Damian Conway. 1996. Seven deadly sins of introductory programming language design. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*. IEEE, 309–316.
[48] Lauren R Milne. 2017. Blocks4All: making block programming languages accessible for blind children. *ACM SIGACCESS Accessibility and Computing* 117 (2017), 26–29.
[49] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Addressing accessibility barriers in programming for people with visual impairments: A literature review. *ACM Transactions on Accessible Computing (TACCESS)* 15, 1 (2022), 1–26.
[50] Salikoko S Mufwene. 2010. Globalization, global English, and world English (es): Myths and facts. *The handbook of language and globalization* (2010), 29–55.
[51] Brad A Myers, David A Weitzman, Amy J Ko, and Duen H Chau. 2006. Answering why and why not questions in user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 397–406.
[52] Oluwakemi Ola. 2023. Using Near-Peer Interviews to Support English Language Learners. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 952–958.
[53] Carla Paciotto and Gloria A Delany-Barmann. 2024. Multilingual Educators in Superdiverse Rural Schools: Placing Administrators and Teachers' Cultural and Linguistic Wealth at the Center of Rural Education. *The Rural Educator* 45, 4 (2024), 62–76.
[54] John F Pane, Brad A Myers, et al. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies* 54, 2 (2001), 237–264.
[55] William Christopher Payne, Eric Xu, Izabella Rodrigues, Matthew Kaney, Madeline Mau, and Amy Hurst. 2024. " Different and Boundary-Pushing:" How Blind and Low Vision Youth Live Code Together. In *Proceedings of the 16th Conference on Creativity & Cognition*. 627–637.
[56] Piumi Perera and Supunmali Ahangama. 2021. SimplyTrans: A Simplified Approach to Sinhala-Based Coding and Introductory Programming Language Localization. In *2021 IEEE 16th International Conference on Industrial and Information Systems (ICIIS)*. IEEE, 318–323.
[57] Chris Piech and Sami Abu-El-Haija. 2020. Human languages in source code: Auto-translation for localized instruction. In *Proceedings of the Seventh ACM Conference on Learning@ Scale*. 167–174.
[58] Ana Cristina Pires, Filipa Rocha, Antonio José de Barros Neto, Hugo Simão, Hugo Nicolau, and Tiago Guerreiro. 2020. Exploring accessible programming with educators and visually impaired children. In *Proceedings of the Interaction Design and Children Conference*. 148–160.
[59] Vesna Popovic. 2004. Expertise development in product design—strategic and domain-specific knowledge connections. *Design Studies* 25, 5 (2004), 527–545.
[60] Venkatesh Potluri, John Thompson, James Devine, Bongshin Lee, Nora Morsi, Peli De Halleux, Steve Hodges, and Jennifer Mankoff. 2022. Psst: Enabling blind or visually impaired developers to author sonifications of streaming sensor data. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–13.
[61] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. Codetalk: Improving programming environment accessibility for visually impaired developers. In *Proceedings of the 2018 chi conference on human factors in computing systems*. 1–11.
[62] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
[63] Rafi Santo, Leigh Ann DeLyser, June Ahn, Anthony Pellicone, Julia Aguiar, and Stephanie Wortel-London. 2019. Equity in the who, how and what of computer science education: K12 school district conceptualizations of equity in "cs for all" initiatives. In *2019 research on equity and sustained participation in engineering, computing, and technology (RESPECT)*. IEEE, 1–8.
[64] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2019. Accessible AST-based programming for visually-impaired programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 773–779.
[65] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. 2015. Transferring skills at solving word problems from computing to algebra through Bootstrap. In *Proceedings of the 46th ACM Technical symposium on computer science education*. 616–621.
[66] Geovana Silva, Giovanni Santos, Edna Dias Canedo, Vandor Rissoli, Bruno Praciano, and Guilherme Andrade. 2020. Impact of calango language in an introductory computer programming course. In *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.
[67] Andreas Stefik, Willliam Allee, Gabriel Contreras, Timothy Kluthe, Alex Hoffman, Brianna Blaser, and Richard Ladner. 2024. Accessible to whom? Bringing accessibility to blocks. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. 1286–1292.
[68] Andreas Stefik and Stefan Hanenberg. 2014. The programming language wars: Questions and responsibilities for the programming language community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 283–299.
[69] Andreas Stefik and Richard Ladner. 2017. The quorum programming language. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 641–641.
[70] Alaaeddin Swidan and Felienne Hermans. 2023. A Framework for the Localization of Programming Languages. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*. 13–25.
[71] Anouk Ticheloven, Elma Blom, Paul Leseman, and Sarah McMonagle. 2021. Translanguaging challenges in multilingual classrooms: scholar, teacher and student perspectives. *International Journal of Multilingualism* 18, 3 (2021), 491–514.
[72] Stefan Trausan-Matu and James D Slotta. 2021. Artifact analysis. *International Handbook of Computer-Supported Collaborative Learning* (2021), 551–567.
[73] Katrien Vangrieken, Chloé Meredith, Tlalit Packer, and Eva Kyndt. 2017. Teacher communities as a context for professional development: A systematic review. *Teaching and teacher education* 61 (2017), 47–59.
[74] Sara Vogel. 2021. "Los Programadores Debieron Pensarse Como Dos Veces": Exploring the intersections of language, power, and technology with bi/multilingual

students. *ACM Transactions on Computing Education (TOCE)* 21, 4 (2021), 1–25.

[75] Sara Vogel, Christopher Hoadley, Ana Rebeca Castillo, and Laura Ascenzi-Moreno. 2020. Languages, literacies and literate programming: can we use the latest theories on how bilingual people learn to help us teach computational literacies? *Computer Science Education* 30, 4 (2020), 420–443.

[76] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 242–252.

[77] Li Wei and Ofelia García. 2022. Not a first language but one repertoire: Translanguaging as a decolonizing project. *RELC journal* 53, 2 (2022), 313–324.

[78] David Weintrop and Uri Wilensky. 2017. Between a block and a typeface: Designing and evaluating hybrid programming environments. In *Proceedings of the 2017 conference on interaction design and children*. 183–192.

[79] Jacob O Wobbrock, Shaun K Kane, Krzysztof Z Gajos, Susumu Harada, and Jon Froehlich. 2011. Ability-based design: Concept, principles and examples. *ACM Transactions on Accessible Computing (TACCESS)* 3, 3 (2011), 1–27.