ProgramAlly: Creating Custom Visual Access Programs via Multi-Modal End-User Programming

Jaylin Herskovitz
University of Michigan
Ann Arbor, MI, USA
jayhersk@umich.edu

Andi Xu
University of Michigan
Ann Arbor, MI, USA
andixu@umich.edu

Rahaf Alharbi
University of Michigan
Ann Arbor, MI, USA
rmalharb@umich.edu

Anhong Guo
University of Michigan
Ann Arbor, MI, USA
anhong@umich.edu

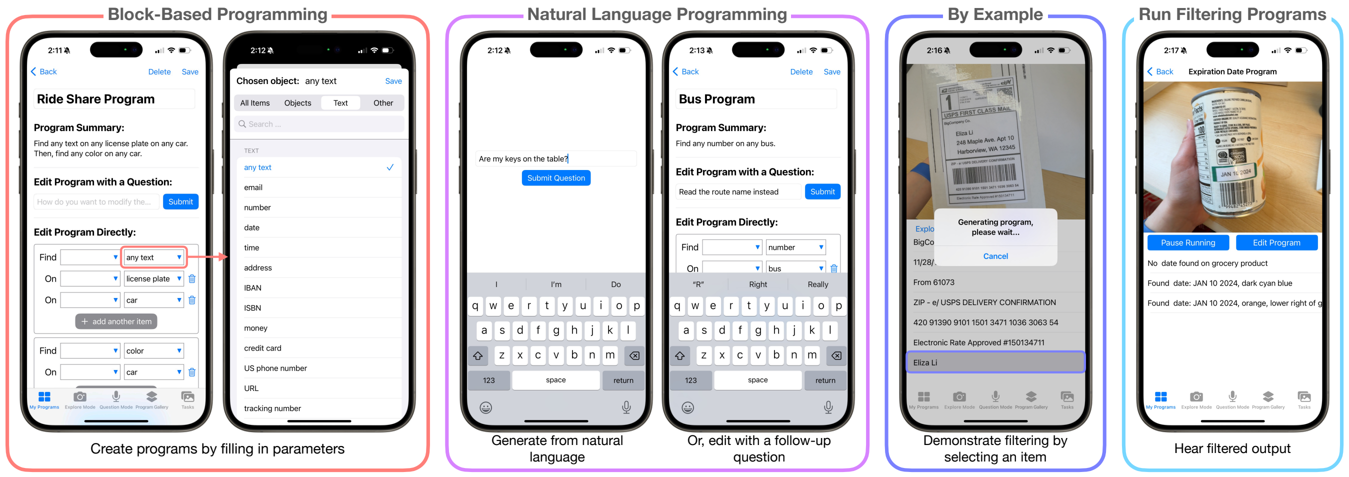
Block-Based Programming
Ride Share Program
Program Summary:
Find any text on any license plate on any car. Then, find any color on any car.
Edit Program with a Question:
Submit
Edit Program Directly:
Find any text
On license plate
On car
add another item
Find color
On car
Chosen object: any text
All Items
Objects
Text
Other
any text
email
number
date
time
address
IBAN
ISBN
money
credit card
US phone number
URL
tracking number
Create programs by filling in parameters
Natural Language Programming
Are my keys on the table?
Submit Question
Generate from natural language
Bus Program
Program Summary:
Find any number on any bus.
Edit Program with a Question:
Read the route name instead
Edit Program Directly:
Find number
On bus
Or, edit with a follow-up question
By Example
Generating program, please wait...
Cancel
From 61073
ZIP - e/ USPS DELIVERY CONFIRMATION
420 91390 9101 1501 3471 1036 3063 54
Electronic Rate Approved #150134711
Eliza Li
Demonstrate filtering by selecting an item
Run Filtering Programs
Expiration Date Program
Pause Running
Edit Program
No date found on grocery product
Found date: JAN 10 2024, dark cyan blue
Found date: JAN 10 2024, orange, lower right of g
Hear filtered output

Figure 1: ProgramAlly is an end-user programming tool for creating visual information filtering programs. ProgramAlly provides a multi-modal interface, with block-based, natural language, and programming by example approaches.

ABSTRACT

Existing visual assistive technologies are built for simple and common use cases, and have few avenues for blind people to customize their functionalities. Drawing from prior work on DIY assistive technology, this paper investigates end-user programming as a means for users to create and customize visual access programs to meet their unique needs. We introduce ProgramAlly, a system for creating custom filters for visual information, e.g., ‘`find NUMBER on BUS`’, leveraging three end-user programming approaches: block programming, natural language, and programming by example. To implement ProgramAlly, we designed a representation of visual filtering tasks based on scenarios encountered by blind people, and integrated a set of on-device and cloud models for generating and running these programs. In user studies with 12 blind adults, we found that participants preferred different programming modalities depending on the task, and envisioned using visual access programs to address unique accessibility challenges that are otherwise difficult with existing applications. Through ProgramAlly, we present an exploration of how blind end-users can create visual access programs to customize and control their experiences.

CCS CONCEPTS

• **Human-centered computing → Accessibility systems and tools**; **Interactive systems and tools**.

KEYWORDS

Accessibility, Assistive technology, Do-it-yourself, End-user programming, Blind, Visual impairment, Design

ACM Reference Format:
Jaylin Herskovitz, Andi Xu, Rahaf Alharbi, and Anhong Guo. 2024. ProgramAlly: Creating Custom Visual Access Programs via Multi-Modal End-User Programming. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), October 13–16, 2024, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3654777.3676391

1 INTRODUCTION

Artificial intelligence (AI)-based assistive technologies can help blind people gain visual access in a variety of common scenarios, such as reading printed text and identifying objects. These applications tend to be designed for simple and common use cases to maximize their broad usability, and prior work has demonstrated that there is still a long-tail of diverse scenarios that automated assistive technologies cannot account for [26]. This leads to users having to shoulder additional cognitive load and adjust how they use the technology to get usable results. Depending on the application, users may need to sift through irrelevant or repetitive information, ask follow up questions, or re-take photos to find specific pieces of information they are looking for. While these methods can be acceptable in some situations, they can be especially difficult in situations where people want a specific piece of information quickly. For example, situations that are repetitive (like sorting mail), time-sensitive (like catching a bus), or require scanning an entire object or room with the camera (like finding an expiration date) can all become burdensome with general purpose assistive technology.

Do-It-Yourself (DIY) assistive technology research has sought to address the related issue of a lack of customizability in assistive devices [29]. To this end, a variety of approaches have been developed aiming to make it easier for non-experts to create adaptive 3D models for themselves or a family member [12, 28]. The same concept has yet to be fully applied to the space of assistive software. From prior work, we know that blind people already put a significant degree of effort into customizing, hacking, or simply envisioning new assistive technologies [26]. Yet, there is a gap between the technologies and customizations that people desire to create, and the systems that can support them in doing so with various degrees of technical expertise.

End-user programming is a potential method for supporting users in customizing and DIY-ing AI assistive software. Ko et al. define end-user programming as a form of programming done by non-professionals, 'to support some goal in their own domains of expertise', further, 'to achieve the result of a program primarily for personal, rather [than] public use' [34]. This definition is aligned with assistive technology needs: blind people are domain experts in designing and using assistive technology [4], and there is a long-tail of unique scenarios that require personalization to meet individual needs. End-user programming approaches, while powerful for enabling users to do more complex tasks, have not yet been applied to the domain of visual accessibility. Doing so presents new research challenges, namely in making tools that are approachable, accessible, and expressive. In this work, we demonstrate the potential of end-user programming approaches for assistive technology creation and customization.

We introduce ProgramAlly, an end-user programming tool for creating and customizing reusable visual information filters (see Figure 1). ProgramAlly is a mobile application that provides a multi-modal interface for creating and iteratively editing short block-based programs (e.g., '`find NUMBER on BUS`'). It is built on a generalizable program representation of similar filtering tasks, derived from a dataset of real-world scenarios from blind people's everyday experiences. ProgramAlly provides a set of methods for implementing programs, using multiple interaction modes: direct input, speech, and camera input. These modes implement common end-user programming approaches: block-based programming, natural language programming, and programming by example. ProgramAlly integrates a set of on-device and cloud models for generating and running programs, and can easily be extended to support new, specialized models.

In a study of ProgramAlly with 12 blind participants, we assessed its three program creation modes, comparing ProgramAlly to existing AI-powered assistive applications, and gathering participants' thoughts on programming and DIY-ing assistive technology more broadly. Four of these participants were consulted as ProgramAlly was being developed, providing design feedback and suggestions for new features, as well as evaluating the programming interfaces and concept. The remaining participants performed a final evaluation of ProgramAlly in both in-person and remote settings.

We found that participants were receptive to the idea of customizing and programming their assistive technology, even if they had no programming experience. Participants envisioned using different programming interfaces depending on the program they wanted to write, the setting, and their experiences with technology. We observed that each interface requires different cognitive and technical skills, and outline specific challenges faced by blind end-user programmers when creating visual programs.

Overall, ProgramAlly is an investigation of how end-user programming techniques can be used to create and customize AI-based assistive technology. ProgramAlly aims to inform how AI models may be directly used as building blocks by blind people in order to support new, complex tasks. This work aims to promote the democratization of AI technology creation and support blind people in having greater control over the AI-based technologies in their lives. This paper makes the following contributions:

(1) A generalized representation of visual information filtering tasks, informed by real-world scenarios from blind people's everyday experiences, that can be easily extended to support new object classes.
(2) ProgramAlly, a system instantiating this representation and providing a set of multi-modal interaction methods for creating visual information filtering programs: block-based programming, natural language programming, and programming by example.
(3) A study of ProgramAlly with blind users, assessing the application of end-user programming approaches to the DIY assistive technology space and highlighting new challenges faced by blind end-user programmers.

2 RELATED WORK

ProgramAlly builds upon a body of prior research on accessibility and programming tools. We first review the need to express specific intents in assistive technology. Then, we review various approaches to technology personalization: personalization in assistive technologies, DIY assistive technology, and end-user programming.

2.1 Information Seeking in Assistive Technology

Searching visual scenes for specific pieces of information has always been an important aspect of assistive technology design. In early remote human assistance approaches like VizWiz, users would submit a question along with an image, and assistants would use their human intelligence to determine a relevant answer [5]. This need for specific information is present across a variety of accessibility contexts: Find My Things and Kacorri use teachable object recognizers to help blind users locate specific possessions [32, 50], VizLens helps blind users search for specific buttons on physical interfaces [23], and CueSee highlights products of interest for people with low vision [72]. Even outside of accessibility contexts, the Ctrl-F shortcut for 'find' is ubiquitous. General-purpose and specific assistive technologies each have their uses; compare the ambient audio cues in Microsoft's Soundscape [47] to navigation directions from Google Maps, neither is a direct replacement for the other.

Yet, current automated assistive applications present challenges to getting specific information quickly. Whether they run on a live camera feed (e.g, Seeing AI [46]) or on a static image (e.g, the GPT-4 powered 'Be My AI' [18]), commercial applications have taken a general approach to describing visual information, conveying all results from the underlying OCR or object detection models, or generating as rich of a description about the visual content as possible. While this is sometimes desirable, it risks slowing down and increasing the cognitive burden on users who are looking for something specific [20]. In this work, we aim to target this need for specificity. ProgramAlly is a live assistive technology that can provide continuous feedback, but it also aims to capture a user's explicit intent through the creation of filtering programs.

2.2 Methods for Personalizing Assistive Technology

In accessibility research, personalization of technology to meet user needs is used to reduce the burden of accessibility on users [21, 61, 62]. This typically leaves the function of the technology unchanged, but aims to automatically map the input and output mechanisms to new systems or modalities [19, 69]. For example, Yamagami et al. recently considered how people with motor impairments would create personalized gesture sets that map to common input mechanisms [71].

Work customizing the functionality of assistive technology is more limited in comparison. In AI assistive technology, teachable object recognizers have been used to allow users to personalize recognition models themselves [33]. Users capture their own image or video data of unique objects that can be stored and later recognized [50, 64]. These approaches can be more useful than off the shelf object recognition models as they are customized to user's specific needs [9, 32]. However, for commercial applications, users have limited avenues for customization. While screen readers can be personalized through a variety of settings, shortcuts, and add-ons [49], AI powered assistive applications are typically part of closed software ecosystems. While they may have some settings within the application for things like language and output speed, this is typically the extent of the customization. Through this work, we hope to demonstrate new methods for personalizing assistive technology functionality to meet unique user needs.

2.3 DIY Assistive Technology

DIY communities have adopted an approach to making centered around personalization, democratization, and collaboration [37, 63]. For assistive technology, DIY approaches can help to address assistive technology adoption due to unique or changing needs [29]. To this end, prior research on DIY assistive technology has sought to make the process of prototyping and making more accessible to participants with a range of technical skills [44, 56].

Most of this research focuses on making physical tools for accessibility (e.g., making 3D-printed devices like an ironing guide, right angle spoon, or tactile graphics [10], prototyping custom prosthetics [27]), rather than software tools. Some tools are being developed to support blind people in DIY-ing more high-tech hardware sensing systems, such as A11yBits [25] or the Blind Arduino Project [30]. While these raise the ceiling of high-tech DIY creation, little research has focused on DIY-ing new software systems for existing devices users already own. In their original case studies of DIY assistive technology, Hurst and Tobias highlighted one instance of 'high-tech custom-built assistive technology', wherein a team of professional programmers worked with an artist with ALS to create software that used eye-tracking input for drawing [29]. Lowering the barrier to entry for creating technically complex assistive software is an important next step in enabling people to DIY personally meaningful assistive technology.

2.4 End-User Programming

Decades of end-user programming research has sought to understand and support programming work done by people who are not trained as programmers [51]. While initially focusing on end-user programming in professional contexts (e.g., using spreadsheets or other domain-specific tools [59]), a variety of approaches have been developed to support programming for personal utility as well. For example, Marmite is an end-user programming tool that allows users to create new applications by combining data and services from multiple existing websites [70]. Here, we describe previous end-user programming approaches that work towards the goal of making programming more approachable for novices. In this work, we aim to apply these existing end-user programming approaches to the domain of visual assistive technology, enabling blind people to have a new level of control over assistive software.

Block-Based Programming. Visual, block-based programming approaches allow users to create programs by graphically organizing elements. These approaches often aim to support novices by providing pre-structured statements to reduce or eliminate syntax errors [48], for example, as in Scratch [52]. While these approaches are commonly used in educational settings [68], they are also used in commercial mobile automation tools to provide sets of components that users can arrange as they wish to create time-saving automations, as in Shortcuts on iOS [1] and Google Assistant [42].

Natural Language Programming. Further work has aimed to synthesize programs from natural language alone. These approaches commonly require a set of training data consisting of queries and desired automations [17, 39]. Large language models have also been used for program synthesis, with mixed results [2, 67].

Programming By Example. Programming by example approaches alternatively allow users to create programs by providing demonstrations of desired functionality, without the need for any code [16, 40]. Programming by example has been implemented in a range of domains, for instance, Rousillon automates web scraping with a demonstration from users on how to collect the first row of a data table [11], and Sugilite automates actions on mobile interface using a demonstration and natural language request [38].

End-User Programming and Accessibility. The accessibility of programming tools is a nascent area [53–55] that has largely focused on developers rather than end users. End-user programming approaches have occasionally been applied to accessibility contexts for the purposes of sharing accessibility bugs and teaching blind children. For example, for web accessibility, demonstration has been used as a method for end-users to convey accessibility issues to developers [3, 6]. Story blocks is a tangible block-based tool for teaching blind students programming concepts [36]. We aim to continue and extend this line of research by supporting blind end-user programmers in creating new visual assistive software.

3 PROGRAMALLY

ProgramAlly is a mobile application that implements end-user programming techniques to allow users to create block-based visual information filtering programs (e.g., `‘find NUMBER on BUS’`). ProgramAlly is implemented as a native iOS application, and consists of the following components, as shown in Figure 2:

(1) **A program representation**, as the framework for implementing and running programs with on-device models.
(2) **Program creation interfaces** provide a multi-modal set of tools for users to create and iterate on programs.
(3) **A program generation server** provides the components for automatically generating programs based on images or natural language text.

3.1 Design Goals

Overall, we designed ProgramAlly based on three primary goals: Expressiveness, Approachability, and Accessibility.

D1: Expressiveness. ProgramAlly's goal is to be an interface where users can customize off-the-shelf models for their own uses. Programs should be able to support a wide variety of real-world use cases through a flexible structure and range of models.

D2: Approachability. ProgramAlly needs to be approachable for non-experts. To this end, it includes a set of methods to create and iterate on programs through multiple modalities, and users can choose what fits their needs. ProgramAlly should aim to have as little technical jargon as possible and explain program parameters in natural terms.

D3: Accessibility. ProgramAlly needs to be VoiceOver and Braille display accessible for users, both while creating and running programs. ProgramAlly's VoiceOver implementation groups related parameters together to provide context for each statement. Additionally, ProgramAlly provides visual context while running programs to help the user aim and know what is in frame.

3.2 Visual Filtering Programs in ProgramAlly

ProgramAlly is built on a generalizable representation of visual filtering tasks. Here, we describe how that represntation was designed, how it is implemented, and how it is used to run visual filtering programs.

3.2.1 Designing a Representation of Filtering Tasks. ProgramAlly's scope of programming visual filtering tasks was determined based on prior work indicating it to be a possible domain for assistive technology customization [26]. We aimed to understand features of filtering tasks in order to build a program representation that could capture a variety of user needs (D1: Expressiveness). Herskovitz et al. captured a dataset of scenarios where blind participants described cases of wanting to create or customize assistive technology [26]. From this dataset, we labeled specific instances as filtering tasks: cases where the participant was searching for a certain type of information. We considered a task to require filtering if using a general scene description tool like Be My AI [18] or OCR like Seeing AI's Document Mode [46] would produce extraneous or distracting information beyond the intended task, but could produce useful results with additional processing.

Out of the original set of 201 scenarios, we identified 29 as filtering. These were fairly evenly spread across all 12 participants from the dataset, with each participant describing at least one. Scenarios fell roughly into two types: finding specific types of text, or specific items. For searching for text, this could be finding specific strings (i.e., a name on a package, a room number in a hotel), finding certain types of text (i.e., a number of miles on a treadmill, the number of calories on a package), or finding text in a specific location (i.e., on a thermostat display, on a license plate). For searching for objects, this could be finding a specific type of item (i.e., a trash can in a mall, a stairway), or items in a specific location (i.e., a person in a chair, an obstacle on a sidewalk).

From this analysis, we determined two key aspects to include in our filtering program representation: (1) the ability to filter by type of object or text, and (2) the ability to filter by an item's location. Our representation includes two types of statements to address this: a ‘find’ statement, and an ‘on’ statement to convey objects overlapping. We confirmed this representation by analyzing a random sample of questions from the VizWiz Question Answering dataset, a dataset of images and questions asked by blind people [24]. We found that the two statements in our representation could represent a significant portion (approximately half) of the 100 queries we analyzed, without the need for additional operators that would increase program complexity.

3.2.2 Program Representation. Programs in ProgramAlly are generally in the form `‘find ITEM on ITEM’`, with any number of ‘find’ or ‘on’ statements. For example, a program can range from `‘find CAR’` to `‘find COLOR on CAR’` to `‘find TEXT on LICENSE PLATE on CAR’`. Adding multiple ‘find’ statements runs each ‘find’ statement in parallel and produces a similar effect to an `OR` operator. For example, the program `‘find COLOR on CAR, find TEXT on LICENSE PLATE on CAR’` for locating a ride share would announce both the color and license plate number of a car if visible.

Additionally, each item in the statements can consist of both a target item (e.g., an object, a type of text), and an optional adjective to describe that target. ProgramAlly supports adjectives denoting color, size, or location. For example, `‘find NUMBER on RED BUS’`, `‘find LARGEST TEXT on SIGN’`, or `‘find ADDRESS on CENTER ENVELOPE’` are programs where the output would be further restricted to match specific conditions. Programs in ProgramAlly are stored as lists of these items (adjective and target pairs).

3.2.3 Running Programs. ProgramAlly uses this representation to run programs to generate live output. ProgramAlly does this by iterating over the list of items in a program backwards, cropping or filtering the source image at each step. For example, in the `‘find NUMBER on BUS’` program shown in Figure 2, ProgramAlly first runs an object detection model that has the class ‘bus’. The model will output a series of bounding boxes that have that class label. Then, the next item in the program is processed. In this case, for each bus bounding box, the frame is cropped and passed into a text detection model. The resulting text is then filtered for strings

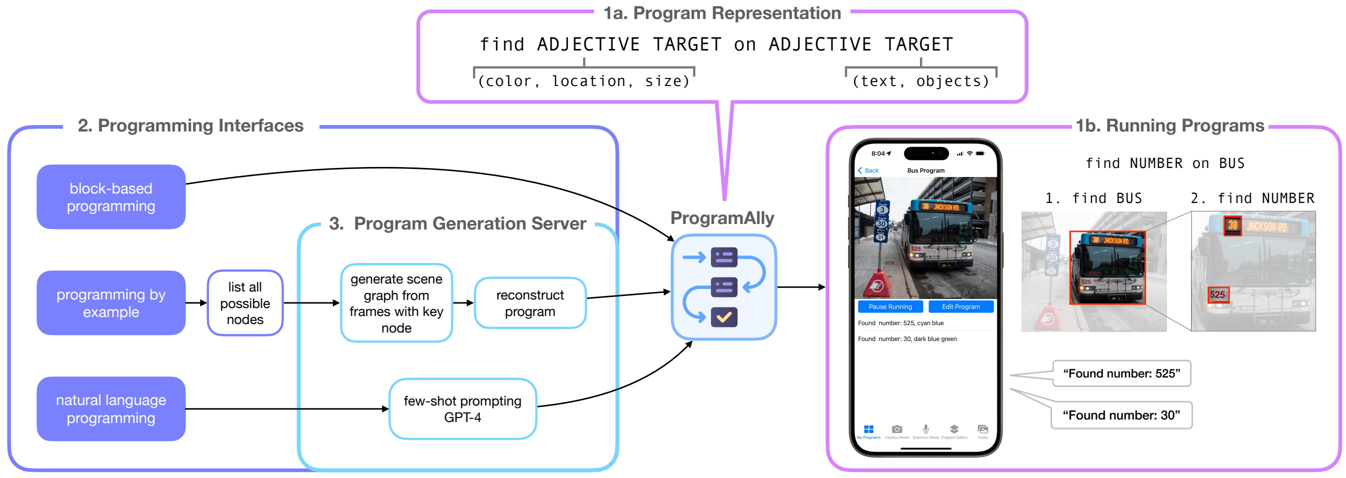

Figure 2: ProgramAlly's main components: (1a) An underlying program representation, the framework for running visual filtering programs (1b). (2) A set of three, multi-modal programming interfaces to support programmers with different levels of expertise. (3) A program generation server which synthesizes filtering programs from images or natural language.

that only consist of numbers. ProgramAlly keeps track of points where filtering fails (say, if no buses are found), and later uses that information to generate descriptive program output.

ProgramAlly currently leverages a set of models and functions for processing each piece of a program. Each item that can be included in a program is stored in a dictionary associating it with the relevant model. The two primary types of targets, objects and text, are both recognized with a set of on-device models, though this could be extended in the future to support cloud models as well. For object detection, ProgramAlly uses a set of YOLO models as they have low latency on a range of iPhones. This includes the default set of 80 object classes detected by YOLOv8 [57, 66]. Additionally, we included a set of modified YOLO-World models, a version of YOLOv8 that can be extended with new detection classes without any fine-tuning through a vision-language modeling approach [13, 65]. We added four additional models with classes that we chose to be relevant to accessibility tasks (D1: Expressiveness) [8]: an outdoor navigation model ('sign', 'license plate'), an indoor navigation model ('door', 'stairs', 'hallway', 'exit sign', 'trash can'), a reading model ('envelope', 'package', 'document", 'poster'), and a product identification model ('package', 'can', 'bottle', 'box', 'product', 'jar'). For this last set of classes, we made them available under one super-class called 'grocery item' for flexibility. New models can easily be added to ProgramAlly as it searches the dictionary for the appropriate model when running a program.

In addition to recognizing objects, ProgramAlly also detects text with iOS's native text recognition. Programs can include the item 'any text', but can also include various more specific types of text such as 'address', 'email', 'phone number', 'date', etc. These types can all be used within programs, for example, `'find ADDRESS on PACKAGE'`. These text types are detected by a combination of Google's Entity Extraction API [22] and regex functions.

Finally, adjectives in ProgramAlly are then used to further filter object or text results. Adjectives include color (red, blue, etc.), size (largest, smallest), and location (center, upper left, etc.). These can be used alongside any item, for instance in the program `'find LARGEST TEXT on BLUE SIGN'`. Adjectives in ProgramAlly were implemented natively: color is detected by matching the most common pixel colors within an object's bounding box to a set of strings; size is determined by comparing an object's bounding box to others of its type and then filtering for the lower or upper quartile; and item location is determined based on a quadrant system, breaking down the parent object (either the image frame or a bounding box) into sections to label the location of a child item (i.e., "text on upper left"). While these implementations are naive, they are meant to demonstrate that a variety of sources of classification can be used in ProgramAlly, and could eventually be replaced by more robust models or algorithms.

3.2.4 Program Output. While running programs, ProgramAlly keeps track of where target items were found in order to give context for each piece of information. For example, if two buses are found in the frame, the output could be: "Found number 73 on bus, left of frame, found number 21 on bus, right of frame." This system also tracks where the program failed if the target was not found. If the first item in the program is not found, ProgramAlly will attempt to provide output for the second item, and so on. For example, if a bus is found with no number on it, the output would be, "Found bus, no number." In this case, because 'number' results are filtered from the more general text detection model, ProgramAlly would also read strings that are not numbers as a backup, for example, the route name. Unique messages are generated for each failure point, for example, if no buses are found ("No bus found"), or if a bus is found but the adjective does not match ("Found white bus, no red bus visible"). This information is used to provided helpful backup information for understanding the scene and aiming the camera.

3.3 Block-Based Programming Mode

ProgramAlly's first method for creating new filtering programs is a block-based programming interface, shown in Figure 1. This block mode can be used to create a program from scratch, or to edit a program that was generated automatically by one of the other two methods. When first creating a new program, the authoring interface will display a program with two empty items in order to provide a default structure for users to fill in. There are two sections of this interface. First, a heading called 'Program Summary', which includes a natural language summary of the implemented program for users to refer back to as they edit. For the default program, this will initially read, "Find any object on any object", and will update as users fill in the program with their desired items.

Next, under a heading called 'Edit Program Directly', users can read through each statement in the program, and edit them with actions in VoiceOver. For example, when VoiceOver focus is on the first 'find' statement, it will announce: "Find any object, actions available: Edit adjective, Edit object, Delete this item." If parameters for the item have already been selected, the VoiceOver description changes to reflect what has been chosen. For example, for the statement `'find RED BUS'`, the description would be: "Find red bus, actions available: Edit adjective 'red', Edit object 'bus', Delete this item." Grouping these together as one single element with multiple actions, rather than having 'edit adjective' and 'edit object' as separate VoiceOver elements, is meant to clarify that the different parameters in the 'find' statement are functionally related, without relying on the visual aspect of them each being on one line (D3: Accessibility). This design was also based on the VoiceOver experience of Apple's Shortcuts app [1], where each block is read as a separate element and editing parameters can be similarly accessed through actions. When either of the edit actions are activated, a new page will appear listing the possible adjectives or objects to fill in the program (shown in Figure 1). The menu includes buttons that can be used to filter the items by type, or a search bar for finding a specific item.

3.4 Natural Language: Question Mode

Inspired by natural language programming approaches, ProgramAlly includes 'Question Mode', which generates a program from a question or statement (D2: Approachability). Users can type or dictate a question, and the resulting program will appear in the block-based interface for them to review and refine further. For instance, the query, 'What does this bottle say?' would result in the generated program: `'find ANY TEXT on BOTTLE'`. This result could then be modified with a follow up question: 'Actually, just read the biggest text' changes the program to `'find LARGEST TEXT on BOTTLE'`.

To prototype this interaction, we use a few-shot prompting approach with GPT-4. We provide a custom system prompt describing how to extract items, and listing the possible item classes. Then, we provide a set of approximately 20 queries and their correct JSON program representation that we wrote based on examples from accessibility datasets [24, 26]. Without developing a custom entity extraction workflow, we found that this approach works well. However, GPT-4 will sometimes produce errors. The most common issue is the model hallucinating new object classes. In this case, the block interface will alert the user that there is an unsupported field and open the editing menu for users to select an alternative. The model very rarely produces programs with an incorrect structure. If the model fails to extract entities, which can happen if the question is vague (e.g., "What is this?") it will occasionally respond

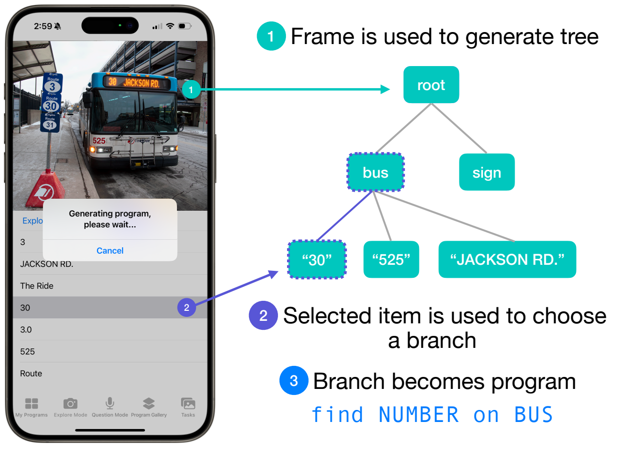

Figure 3: In explore mode, ProgramAlly provides a list of all items detected in the camera feed. Users then demonstrate filtering by choosing a specific item. That item is then used to fetch a specific branch from a scene hierarchy, which becomes the program.

with natural language rather than a program (e.g., "I'm sorry, I don't know what you mean, can you clarify?"). Future work could use additional fine-tuning to create a conversational approach to clarifying ambiguous language.

ProgramAlly also includes a method for users to edit programs with a follow-up question, rather than by manually editing a generated program using the block-mode. When editing a generated or pre-existing program, there is a text box where users can type or dictate a follow-up question (Figure 1). Various follow up questions are included in our system prompt to GPT. Based on feedback in our formative studies, we also included an option to access this feature while a program is running. If the program output is not as expected, users can directly access the option to edit the program with natural language, for rapid iteration. For example, when running the program `'find NUMBER on BUS'`, the user could provide the statement, "Read the route name instead", and the program would be modified to be `'find TEXT on BUS'`.

3.5 Programming-By-Example: Explore Mode

ProgramAlly's Explore Mode allows users to automatically generate a program by selecting a target feature detected in the camera feed (D2: Approachability). Explore mode lists all object and text features in the image, and users select an item to filter for, effectively providing a demonstration of the filtering behavior. Explore mode was included to address the challenge of unknown-unknowns [7]: without knowing what visual features or information is present, blind users would not necessarily have all of the information available to write a working filtering program.

In this mode, ProgramAlly runs all object and text detection models at once, with the goal of outputting everything that a user might want to create a program to find. Users then select the information they are looking for to demonstrate filtering behavior, and a program is generated which aims to filter for that type of information in the future. For example, as shown in Figure 3, the camera is pointing at a bus stop. If the user selects '30', which is a route number, the resulting generated program will be `'find NUMBER on BUS'`, because that is where the text '30' was found in the frame. Once a program is generated, the app again displays the result in the block-based interface, with the program summary and the option to edit the generated program further either with a question or with blocks.

3.5.1 Generating Programs from Demonstrations. Programs are generated using ProgramAlly's server, which represents images as a tree of items. First, on the device, ProgramAlly maintains a list of each frame where an item was detected. When the user selects an item, a frame is then chosen that contains that item, which is then sent to the generation server. ProgramAlly's server uses a set of models to then generate the tree structure. These are slightly different than the models used on device, and includes Mask R-CNN under Detectron2 [45] and Google's Cloud Vision API [15] for object detection and Google Cloud's OCR model [14] for text. Additionally, to label adjectives and other properties associated with each item, the server runs DenseCap [31], a model that creates rich language descriptions of image regions. Because DenseCap produces natural language descriptions associated with bounding boxes, we use a few-shot prompting approach to GPT-4 [58] to extract and label the relevant objects and their associated adjectives.

Next, each item is stored as a node in a scene graph hierarchy based on bounding box overlap. The parent node is the entire image, and child nodes can either be text or objects, stored with their associated adjectives. Finally, from this scene graph, the originally selected node is then used to generate a program. The selected node is located in the scene graph, and all of its ancestor nodes (not including the root image) are then selected, representing a branch of the graph (see Figure 3). Traversing up this branch, each node then becomes an item in the program. Each node in this set is converted into an adjective and object pair, and ordered based on their parent-child relationship in the source graph. This generated program is then sent to the device as JSON. While ProgramAlly currently uses a strict tree structure to avoid any ambiguity (ensuring that a single branch can always be chosen), this does limit this generation technique to supporting only the current 'find' and 'on' operators. To support more complex programs, new synthesis techniques would need to be developed.

Because the server includes the addition of DenseCap for describing objects, there may in rare cases be a class in the generated program that is not present in the app, although we aim to filter these classes out when possible. In this case, the block interface will again alert the user that the field is unsupported and surface the menu for selecting a replacement.

4 USER STUDY PROTOCOL

To understand how ProgramAlly can be used as a tool for creating and customizing assistive technology, we conducted a study with 12 blind participants. **Our goals were to (1) assess the accessibility and approachability of ProgramAlly, and (2) understand unique challenges faced by blind end-user developers creating visual technology.** This study was approved by our Institutional Review Board (IRB). Participants were compensated $25 per hour for their time and expertise. This ranged from 1.5 to 3 hours in total, with an average time of 2 hours.

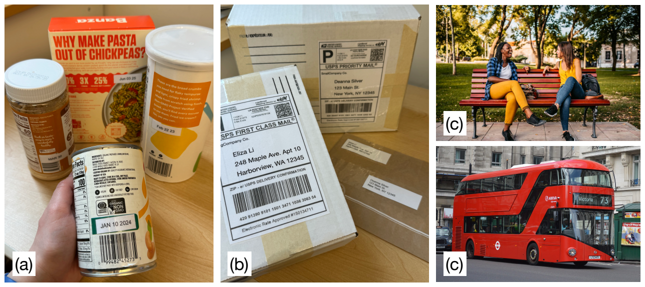

Figure 4: Samples of props used in our study: (a) Grocery props for in-person testing of `'find DATE on GROCERY ITEM'`, (b) Mail props for in-person testing of `'find ADDRESS on PACKAGE'`, (c) Images used by remote participants, for testing `'find NUMBER on BUS'` and `'find PERSON on BENCH'`.

We aimed to involve participants in ProgramAlly's design, so the first four participants were consulted as it was being developed and informed many of its final features. Because these participants also completed a similar study protocol as the remaining participants, we include their results here as well. Overall, this study was completed with three groups of participants:

1. **Pilot Participants:** Four remote participants who tested ProgramAlly as it was being developed. The first two participants only used the block-based programming mode, and the second two participants used all three modes.
2. **Remote Participants:** Four remote participants who completed a full evaluation of ProgramAlly, running filtering programs on sample images.
3. **Face-To-Face Participants:** Four in-person participants who completed a full evaluation of ProgramAlly, running filtering programs on provided props and comparing filters to existing assistive apps.

4.1 Participants

Participants were recruited using email lists for local accessibility organizations, prior contacts, and snowball sampling. Participants were required to be over 18 years old, have some level of visual impairment, and regularly use a screen reader to access their devices. Participants were also required to have an iPhone so that they could download ProgramAlly via TestFlight.

Demographic information for participants is shown in Table 1. Of the 12 participants, two had some prior programming experience for their coursework or career. However, participants had a range of experiences with technology and VoiceOver, and not all were experts. For example, R2 and R4 were assistive technology professionals, while F2 was new to using VoiceOver and had not previously used any mobile assistive applications. We recognize that recruiting remote participants can create a bias for people who are technically savvy, as they need to have a desktop and be familiar with Zoom. To try diversifying our sample, we also recruited in-person participants.

4.2 Procedure

After a brief introductory interview, participants were introduced to ProgramAlly by reading through a pre-written program to familiarize themselves with the concept and interface. Participants

ID	Gender	Age	Occupation	Vision and Hearing Level
P1	Man	22	Computer science university student	Blind with no light perception, from age 5
P2	Woman	54	Not employed	Blind with some light perception, from age 49
P3	Man	29	Graduate student	Blind with some light perception, from birth
P4	Woman	32	Program Director	Blind with some light pereption, from birth
R1	Woman	41	Translator	Blind, from birth
R2	Man	37	Accessibility consultant	Blind, from birth progressive up to age 15
R3	Woman	68	Retired	Blind with some light perception, progressive since birth
R4	Man	41	Assistive technology research and training specialist	Blind with no light perception from birth. Deaf/ hard of hearing.
F1	Woman	53	Not employed	Blind with no light perception, from age 51
F2	Woman	60	Retired teacher	Blind/ low vision, 20/500, from age 27
F3	Man	40	N/A	Blind with some light perception, from birth. Slightly hard of hearing.
F4	Woman	72	Retired social worker	Blind with some light perception, from age 23

Table 1: Participant demographics for our study with 12 visually impaired people. Participants self-described their level of vision. All participants used a screen reader to access their devices and read text. Participants with a 'P' were part of the pilot testing, participants with an 'R' completed the full study remotely, and participants with an 'F' completed the full study face-to-face.

were then asked to modify the example program slightly by adding an adjective. We tried to keep verbal instructions minimal to let participants reason for themselves about the program. The only pointer that we gave to participants was that they could swipe up or down on the program elements to hear the different editing actions available, because depending on the verbosity settings of their device, VoiceOver may not have spoken this information.

After this introduction, participants then used ProgramAlly to create and run three programs. Participants used each of the three programming interfaces (block mode, explore mode, question mode), and were assigned tasks to create a program for (i.e., 'create a program that will find addresses on mail'). While creating each program, participants were prompted to think aloud to explain their thoughts or concerns, or to ask questions. Participants then ran the programs they created, with remote participants using sample images and in-person participants using props. These props included books, grocery items, and packages and mail, and examples are shown in Figure 4. In-person participants were also given the option to compare their program to either Be My AI or Seeing AI, depending on what they would normally use for the same task. After creating each program, participants were asked to rate the ease of creating the program, and how accurate they felt the program was.

Finally, participants were asked to rank the three creation modes on different factors. The study concluded with an open-ended interview about the prototype app and the experience of using end-user programming methods for DIY assistive technology overall. Participants were asked to imagine how such an app could fit into their existing assistive technology workflows, and the pros and cons of creating programs to customize assistive technology.

4.3 Data Collection and Analysis

Remote participants joined a Zoom call that was recorded, and when testing the app they were asked to share their phone's screen in the call. Similarly, the face-to-face participants interviews were audio recorded, and their devices were screen recorded. The audio was later transcribed and used for analysis. We then created written descriptions of participant's strategies for completing each tasks from the video data.

Since participants were encouraged to use a think-aloud method and take their time to fully explore the functions, ask usability questions, and give feedback, performing an analysis of task completion time does not provide much insight about how ProgramAlly works in practice. Instead, we primarily report qualitative data on participants' strategies and workflows while using ProgramAlly, and their general feedback.

5 USER STUDY RESULTS

Here we present results from our user study. First, we discuss participants' experiences creating and running programs in general. Next, we discuss how participants used and compared each of the three program creation modes. Finally, we discuss participants impressions of end-user programming as a customization tool.

5.1 Using Filters in ProgramAlly

Participants generally felt positively about using filtering programs in ProgramAlly. As P1 described: *"There is an abundance of visual information. And sometimes a blind person is short on time, and you really just want the particular piece of information you're curious about"* (P1). Not all of the example filtering programs were equally useful to participants. For example, R1 appreciated the `'find PERSON on BENCH'` program and envisioned using it in a local park with many benches, while R3 had a seeing eye dog already trained to do this task. All participants saw practical use in at least one of the filters they tried, and all were able to come up with ideas for filters they could create in the future.

5.1.1 Benefits of Filtering. Almost all participants saw filtering as an important niche not filled by existing assistive technology. As R3 described: *"I definitely see a use for this app. Earlier I mentioned that I usually think, 'I already have 3 apps that do the same thing. Why do I need another one?' But this one, because of the ability to filter, you know, as much as you want to, you can get very specific, I don't think anything like that exists. Or maybe it exists in a 5 step*

ID	Program Idea
P1, F2	Identify products by brand or flavor when sorting, `find BRAND NAME* on GROCERY ITEM`
P3	Find cooking instructions, `find TEXT AFTER "COOKING INSTRUCTIONS"* on GROCERY ITEM`
P3	Sort credit cards, `find NAME* on CREDIT CARD*`
R3, F3	Organize books and CDs, `find "AUTHOR NAME"* on BOOK`
R4	Find room number in hotel, `find NUMBER on SIGN`
F3	Identify car models, `find MAKE/MODEL* of CAR`
F4	Differentiate between two cats, `find ORANGE CAT`

Table 2: Participants envisioned creating new filtering programs outside of those they used in the study, sometimes involving new classification models (marked with a *). A sample of these ideas are shown here.

process" (R3). Similarly, R4 mentioned how filtering could speed up some tasks: *"I think it's the next natural direction to go in, in some ways, sometimes you get a bit too much information and you can speed up the process by not having it automatically generate an image description that you might not even have any use for"* (R4).

Participants expressed that filtering seemed particularly useful for tasks that were repeated as part of their routines. F2 said, *"I probably would save some [programs] to the library. Because some stuff I would probably use over and over again"* (F2). In routine tasks, efficiency can be more crucial. F3 said, *"Well, it depends on your routine, you know. If you have something that you do on an ongoing basis, and you really need this life hack, say, to make it real simple for you, then absolutely"* (F3). Participants were also able to envision creating new filters outside of the ones prompted in the study. A list of examples is shown in Table 2.

5.1.2 Comparing ProgramAlly to Other Assistive Technology. In-person participants were able to directly compare filtering programs they made to other automated assistive apps of their choice. In general, participants preferred using the filters they wrote over Seeing AI for the same task. For example, F2 reflected on the '`find ADDRESS on PACKAGE`' program: *"The filter, I think, did center just on the address. It read it one time, and I knew where the start and finish was. In Seeing AI, it seemed like it would keep reading and reading and reading... I would have to listen to it twice to make sure I got the full address, So that's why I slowed it down, because I really had to pay attention where the start of the address was and what it was telling me"* (F2). Here, when F2 used Seeing AI to read a shipping label, it read additional extraneous information aside from the address that they then had to manually sort through, to the point where they reduced VoiceOver's speaking rate to listen carefully for the information. Similarly, F1 and F3 preferred the '`find DATE on GROCERY ITEM`' program over Seeing AI, as Seeing AI never read an expiration date, it just read other information on the product package, despite the date being in frame.

Participants preferred Be My AI for some tasks, but not all of them. For example, F4 compared the '`find DATE on GROCERY ITEM`' to Be My AI. After trying multiple times, they were unable to take a photo for Be My AI with the expiration date in frame, and gave up. On the other hand, participants tried a filter '`find LARGEST TEXT on POSTER`', which was created imagining a scenario where someone would want to skim over fliers on a bulletin board. Participants though Be My AI was better suited to this task, as it was easy to feel a flyer to center it in the frame, read the first line of the output description, and disregard the rest. This confirms our hypothesis that ProgramAlly is better suited to tasks that are continuous or repetitive, where taking a single photo is difficult.

5.2 Programming Process and Challenges

Most participants felt like with practice, they could become more familiar with the system and would be able to quickly create new programs. However, as blind end-user developers they also faced unique programming challenges.

5.2.1 Programming with Unknown Unknowns. One such challenge is not knowing what the 'ideal' program would be due to not knowing the parameters or targets included in the system. While this is a challenge for many end-user programming tools [35] and can be alleviated with more familiarity, R3 pointed out that this is also tied to prior visual ability: *"I'm gonna point out that, depending on your onset of blindness, you might not know what questions to ask. So it's going to depend on the user and their life experiences... Because their experiences, haven't you know, given them the ideas of how to word a question"* (R3).

To potentially address this, R3 imagined follow up information in the example and question modes that would help them understand the possible programs better. They said, *"If there were things about this object that I wanted to know, but I didn't know that they're there, then there are questions that I'm not coming up with that would help me. So you know, in that respect, if the app gave me suggestions like, 'Why don't you ask it this?' It might help you get to your final question"* (R3). Explore mode in particular could be improved by listing possible programs, rather than only listing possible features of interest.

5.2.2 Understanding Object Classes. Even when participants were aware of a possible object class, they often wondered about the extent of what it would detect. For example, P1 questioned the program '`find DATE on BOTTLE`': *"I don't know if it's super restricted to a specific definition of a bottle, or pretty much any box or container. Could it also work with a box of cereal, which also has an expiration date on it? A cereal box can hardly be, in colloquial terms described as a bottle. But from the AI's perspective, I wouldn't be too surprised"* (P1). The ability to test classes in isolation, outside of a filtering program, could be helpful in determining the applicability or reliability of a class.

5.2.3 Balancing Specificity and Reusability. Participants thought carefully about how to produce filters that were specific enough to be useful, but generalizable enough to be re-usable. For example, when writing the '`find DATE on GROCERY ITEM`' program, R2 noted that there may be both an expiration date and sell-by date on an item. They said, *"I guess with any of these filters is like, do you go broad, or do you go narrow? And I think 'any date' works, with the knowledge that there are likely to be multiple dates available. That's not to say that I won't find the right information, but it just might find additional information that I didn't want"* (R2). This is also somewhat dependent on prior visual ability, as someone with prior sight may be able to recall specific visual features that could cause conflicts in their filters.

Using only a single object detection class in a program could also limit its re-usability. For example, when reading the program `'find ADDRESS on PACKAGE'`, F4 noted they would want it to run on not just packages, but also on envelopes, mailers, or similar items. Creating new super-classes by grouping relevant items could improve the robustness of possible filters.

R4 noted that balancing program specificity could become more difficult when they were outside of their normal routine: *"For example, in a situation where maybe I am trying to find a specific bus, I want to be able to quickly do that. So I would pre-write the program. But like, I could show up in a city next week, and not be there again for 3 years, so there would be no point in me generating different programs based on that. So it's faster to just ask the question. But you know, in other situations, when I might have a little more time to put it together, or things like that, it seems like the other methods are a bit more accurate"* (R4). Here, they described how if they needed a filter outside of one that they would typically use, they could quickly create one by asking a question, rather than spending time to carefully set up something they may only use a handful of times.

5.3 Creating Programs with Blocks

ProgramAlly's block-based programming interface has the advantage of offering more fine-grained control over a program, but it also has the highest learning cost. However, participants expressed that this was something they felt confident in being able to learn over time as they created new filters.

5.3.1 Advantages: Control and Accuracy. Participants found that this mode produced programs that most accurately matched their intents because they could quickly specify exactly what information they wanted: *"I think this produces the most consistent accuracy, knowing if you can go in there and just select things"* (F1). They also observed that this mode reduced some of the ambiguity present when a program was automatically generated for them: *"I would say it's the least difficult way, just because you know kind of what you're getting, and you know what's available"* (R2). Here, R2 is describing that when they already had a program in mind and knew the format, it was easier to program it directly than it was to think about how to phrase it in natural language.

5.3.2 Learning to Think Programatically. The block-programming mode required participants to adopt a programming mindset and break down an idea into multiple components. For example, participants sometimes added items in the wrong order. P3 wrote the program `'find BUS on NUMBER'`, and described their thought process as *"I was thinking, okay, first I need to look for a bus. That's the most important thing. So putting that first... I was going through a more like, linear process, you know, look for the bus then look for the number. But that's just like how the brain works. But if I had paid more attention to what it was saying..."* (P3). R2 also noted that it took some extra thought to consider how the scene would visually appear: *"For me, I was just thinking about it in terms of like, if you're looking for two objects which is going to be visually larger and easier to discover"* (R2). They then realized that they had put the elements in the wrong order when they went back to read the program summary: *"This particular field helps me to determine which order I should put things. So I see that I said, 'find any bench on any object'. So I actually want to find 'any person' first"* (R2).

Additionally, when parameterizing their request participants were sometimes unsure if an item they wanted to add would be considered an adjective or an object. R3 wanted to add the adjective 'red' to an existing statement that said 'find any book', but they selected 'edit object' because they wanted to edit how the book was detected: *"Cause I was like, 'adjective'? I mean, I know what an adjective is, but I wasn't like relating it to 'red book'"* (R3). Similarly, F3 described, *"It's like I have to fill in these categories. And I have to think about which category is which. I have to think why adjectives, and why objects and things, and it's interesting"* (F3).

5.3.3 Challenge: Interface Complexity. The block-based programming mode has more interface elements, which participants recognized as a learning curve. Additionally, even once familiar with the interface, there is still a time cost to creating longer or more complex programs where many components need to be edited. R2 described: *"So the disadvantage, of course, is like having to go through the whole process, which can be as long or short as necessary. It's the one that has the most, I would say, interaction cost in terms of just having to touch your device and manipulate the interface"* (R2). Creating a long program like `'find TEXT on LICENSE PLATE on CAR, find COLOR on CAR'` would take more time than creating the program `'find PHONE'`, simply because there are more parameters.

5.4 Generating Programs from Questions

Participants considered question mode to be the fastest way to get started making programs. However, the generated programs sometimes did not capture the correct parameters.

5.4.1 Advantages: Fast and Approachable. Many participants preferred the natural language mode because it was fast, and required the least cognitive effort. As R4 described: *"It's faster, overall like, you don't have to break it down and add an adjective, or add this or that. You just have to have it categorize things correctly, which it seems like it mostly does"* (R4). Even if the generated program was not exactly as intended, participants felt that they could use it as a starting point for editing (either with blocks or follow-up questions). For example, F3 said, *"I feel like I'm more in control of it. I can be just type in what I want, and then customize it from there. It just seems more straightforward to me to interact that way"* (F3).

5.4.2 Challenge: Language Can Be Vague. Participants felt that sometimes the generated programs did not capture their full intent. R3 described how this could be due to ambiguities in natural language, comparing how they would phrase a program for finding dates versus one for finding bus routes: *"I mean a date is a date. There's no leniency. But if you're asking for a bus route, do you mean the number of a route or the label of a route?"* (R3). Because of this, some participants spent time thinking about exactly how to phrase a question to get it to detect what they wanted. R1 noted that they would rather edit the program directly than think about how to phrase questions: *"[In block mode], I can be more specific and I can choose exactly what I want... Here, I don't know, it's still too difficult to phrase. I need too much brain power"* (R1). Participants also sometimes forwent natural language, and simply dictated a statement in

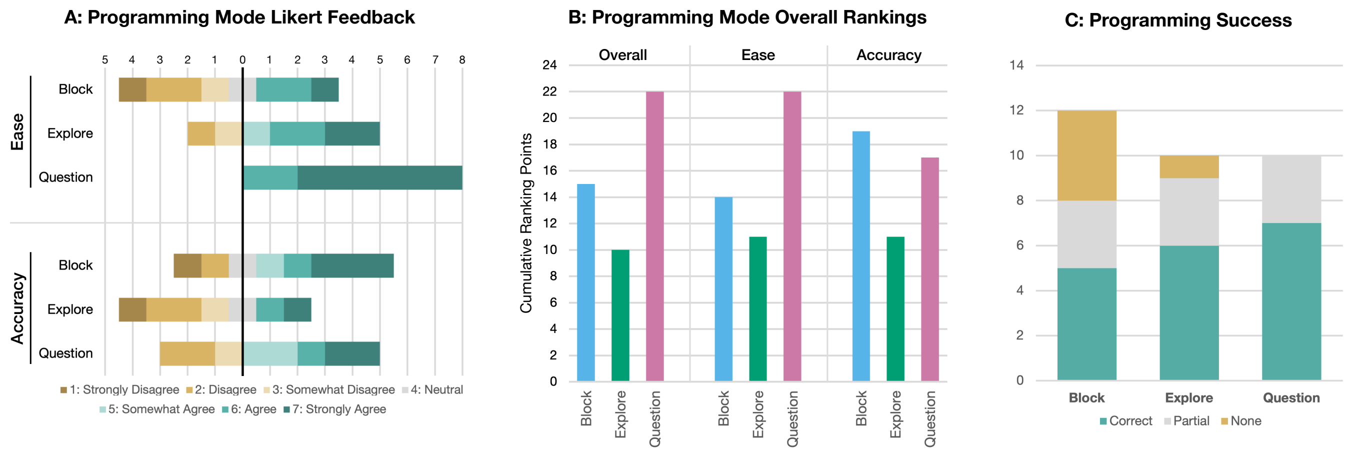

Figure 5: Participants rated each of ProgramAlly's three creation modes on a set of factors. Charts (A) and (B) demonstrate the trade offs between block and question mode: question mode was found to be easiest, but block mode was perceived to be slightly more accurate. Chart (C) demonstrates that block mode had the highest learning curve, though participants were able to create correct programs with all three modes. Each mode may be suited to different users or scenarios.

the program structure ("find blank on blank") to try to overcome this.

5.5 Generating Programs from Examples

Participant appreciated the potential of explore mode to give them a new awareness of visual features. But, in its current state, it was difficult for participants to know what features to select, and the generated programs sometimes contained unexpected conditions.

5.5.1 Advantages: Finding Unknown Features. Participants saw the greatest strength of this mode as its potential to make them aware of new features and program possibilities. *"The summary gave information that I wasn't even aware of. So in that respect it was good, it was like 'oh, buses are red, gotcha'"* (R3). *"For like the odd situation, you could use the explore option. And once you realize all of the details that are out there, you could say, well, this is worth categorizing, and I wanna be able to, you know, have a filter just based on stuff in the environment that I didn't know about. And I could actually see a real need for the the explore mode over the [question mode], because sometimes we don't know what we're working with in the field"* (F3).

Additionally, P3 pointed out that it was a useful way of testing the object detection would work on a specific item before manually writing a program for it. *"For example, if there is some item, let's say medicine, or a bottle of milk or something that you consume every day. You don't know if the app would recognize that particular milk or not. So rather than try to create a program and try my luck, I can test it out directly. And if it detects, then you're cool, like, just create a filter for the next time, you know, and keep it saved, and then you can run it every time"* (P3).

5.5.2 Challenge: Extrapolating Intent from a Single Feature. In explore mode, participants needed to pick a feature of interest, and ProgramAlly generated a program based on finding that feature in the future. Participants noted that it was hard to pick a target without knowing what the resulting filter would be, and without knowing the context of some visual features. For example, when trying to generate a program that would read the route number on a bus, some participants debated between choosing the object 'bus' or the number '73', both of which appeared as possible targets.

The program generation method we developed attempted to create filters to match specific visual content, though participant feedback revealed that one generation method is not suited to all tasks, and ProgramAlly may be including too many visual features. For example, R3 reasoned about why a generated filter included the adjective 'red': *"Maybe the bus in the image was red, but no it doesn't seem relevant to the route number. It probably did the right thing, it probably filtered it. Maybe all 73 buses are red. So it might have done the job that it was supposed to do, and not the job that I wanted it to do"* (R3). F4 similarly noted that different features are relevant for different tasks. They generated a program by selecting 'book' as the target, and the result was 'find blue book on table': *"You know you may be looking for. The colors, you're like, okay... When it comes to a book, that's not what I really need, because, like, when we look for our books, we look by title and all, of course"* (F4). Considering these types of semantics when generating programs could improve the results.

5.6 Comparing Creation Modes

Participants generally appreciated all three programming modes were available in the app. An overview of how participants compared the modes is shown in Figure 5. Each mode required participants to think about their goal in a new way, and required different types of effort to turn it into an operable program. Parameterizing a request in block mode, selecting a visual feature in explore mode, and phrasing requests in question mode each presented their own strengths and challenges. Because of this, participants could see benefits of using different programming interfaces in different scenarios. As R4 put it: *"It's all contextual, I think. So it depends on what you want to do. Like, we did 3 different examples. But I would* *use different methods based on what I knew about the environment. It just depends on what you're doing, you know, if you already have an image you're working with. You might go with that particular program, you know, you explore the image, and then that's what you use. It just really depends on the situation"* (R4).

Participants also noted that the different modes could be helpful to people with different levels of technical expertise. P3 said, *"I think, also, like, to the user, they'll be scared, like, I have to do programming to use the app... But I mean, anyone pretty much can do it, and there are multiple ways, even if one method is gone. So there were two other methods to create"* (P3).

5.6.1 Structured vs. Unstructured Input. Despite question mode seeming more approachable, block mode's structure provided participants a framework to work within. As P1 described: *"I think one positive is that it gives you more of a predetermined list to choose from. So kinds of objects that you might be interested in, kinds of things on those objects that you might be looking for. One of the challenges with personalization is you can give someone too much choice to the point that they are overwhelmed and unsure of where to even start"* (P1). This framework gave people an understanding of the limits of what they could create.

To balance these approaches, multiple participants imagined a hybrid block and natural language approach, where the block structure would still be present, but they could type in or dictate each program item instead of scrolling through menus. For example, F3 described, *"Maybe if the display were slightly different, like 'find any blank on any blank', and I could input the text there, that would make sense. But I'm trying to choose objects and adjectives and things like that. And it just seems a little cluttered"* (F3).

5.6.2 Editing Generated Programs. Participants generally appreciated that the two program generation modes (explore mode and question mode) displayed their results in the block interface, even participants who did not prefer that interface to start. For instance, F2 and F3 both preferred the generation interfaces, but agreed that it was easier to edit an existing program than to create one from scratch. Participants also expressed that they would use this interface to refine the generated programs, as P4 said, *"It's always good to have a backup there"* (P4).

5.7 Benefits and Drawbacks of DIY-ing Assistive Technology

Participants appreciated the deeper level of customization available when programming filters as compared to current assistive technology. As R2 described, it puts power into the hands of the user to decide what they needed: *"I think it all comes down to providing choice. Ultimately, what I like is that you're putting the information available, in the person's hands to choose... You know, just being able to empower people to be independent... What you've all created here is really neat because it's creating modularity to access the information. And I love that. I love that. And I wish more and more assistive technology companies thought about, how can we take these pieces of information and put it in the hands of the people that need it in a way that they can then modify it and change it and make it their own"* (R2).

On the other hand, R1 noted that having to put in the effort to create filters themselves could be considered a burden: *"People are not developers. Another developer, I suppose, knows how to program… I'm not a builder. There are tasks that are difficult for us to do, but what, I have to spend 5 hours to tinker with a program for what?"* (R1). R1 expressed that they felt like ProgramAlly was a good option for developers wanting to quickly create things, but that it may not be ideally directed towards end-users. Other participants who did not mind the idea of programming still mentioned that re-framing the functionality might make ProgramAlly feel more approachable. Eventually, ProgramAlly could be considered as a platform for people to share programs that they have created, enabling a level of collaboration among end-users with different levels of expertise.

6 DISCUSSION AND FUTURE WORK

We found that ProgramAlly addresses unmet needs, empowering blind people to customize their experiences with AI. Here, we outline opportunities for building on ProgramAlly to further improve its utility.

6.1 Raising the Ceiling of Creation Possibilities

Throughout our user studies and analysis of existing data, we encountered many scenarios that could be addressed by ProgramAlly with the addition of new program operators. While ProgramAlly was implemented with 'find' and 'on' statements for simplicity and approachability, the addition of new operators could make ProgramAlly more expressive for DIY enthusiasts and power users. For example, the addition of traditional logical operators such as `AND`, `OR`, and `NOT` would allow for a greater degree of specificity in programs. However, `AND` and `OR` are easily confused among end-users as their natural language counterparts can be ambiguous, so introducing these would need to be done with care.

New operators could also define additional ways for objects to interact with each other. Currently, 'on' denotes objects whose bounding boxes are primarily overlapping. An operator like 'nearby' could specify items that do not overlap, but are in proximity. Similarly, 'following' could specifically find text content after a phrase, as in `find TEXT following "EXP:"` for more specifically finding an expiration date. ProgramAlly as a system could be extended with these, but it would require new block interface designs, a challenge for approachability and accessibility.

Additionally, ProgramAlly could benefit from the inclusion of additional, specialized models for certain tasks. For instance, a model that detects the make and model of a vehicle for locating a ride share, or text classification models that can filter out 'brand names' or 'flavors' for shopping scenarios. One notable object class missing in ProgramAlly is 'digital display', for reading screens on thermostats, microwaves, or buses. We attempted to use YOLO-World to detect this class, but found that it was not accurate enough to be usable. Although we currently use customized YOLO-World models on-device so the classes are pre-selected, YOLO-World was built for the ability to add new classes in real-time. In the future, when question mode extracts parameters from a request, the server could automatically generate new YOLO-World models to fill in gaps in the program as needed.

In the future, we imagine ProgramAlly being paired with other personalization approaches to create a deeper level of customization. For instance, if combined with the capabilities of teachable object recognizers, users would not only be able to locate personal items, but to integrate them into programs as a basis for further filtering or automation. Overall, we also see ProgramAlly as going beyond programming for visual information tasks. We believe our study reveals important findings about how blind end-users program, ideally leading to supporting more complex tools for people to DIY a range of assistive technologies outside of filtering programs.

Furthermore, when the creation ceiling in ProgramAlly is raised with things like additional operators and models, or if end-user programming approaches are eventually used to create different types of assistive technologies, this comes with a trade-off. Balancing this complexity with ease of use is a critical concern for future accessibility work in this direction. For instance, although ProgramAlly could eventually include a large library of models and classes to detect, these could be activated selectively or as-needed for different users or situations, making the system less overwhelming. Eventually, ProgramAlly could also make suggestions for how to create or improve programs based on how a person uses the system over time.

6.2 Automating Running Programs

Because of the long-tail problem, some of our participants saw managing a library of programs as unwieldy. For example, R1 said, *"Would I have to program actions for all the objects in the world?"* (R1). P1 expressed a similar sentiment: *"When it comes to just the variety of information that anyone might be looking for, at any given point of time... I don't know. It just feels like there are so many permutations and combinations here. So many ways in which humans may want to query information that trying to build an even remotely comprehensive list of the more common categories of information seems like an endeavor that's really hard"* (P1).

Being able to automate when programs are run could remove some of this burden. For example, a `'find ADDRESS on PACKAGE'` program could be automatically started whenever a package enters the frame, on the assumption that the user is sorting mail. Or, like other mobile automations, programs could be tied to a location. For example, when a user arrives at a bus stop, the `'find NUMBER on BUS'` program could start.

6.3 Programming in the Age of VLMs

Although large vision language models (VLMs) are becoming more powerful, they may not be a panacea, and making them truly beneficial requires deep integration with the needs of blind people. Despite the advantages of providing fully automated, subjective descriptions, they also add new challenges for blind users to acquire information. While this is still an area of active research, Massiceti et al. found the CLIP-based models were up to 15% less accurate on images taken by blind people [43]. Additionally, as hallucinations seem to happen more often when describing complex scenes [41] or when being asked a follow-up question (the model appears to second guess itself), they could potentially arise more in accessibility contexts.

Generally, this also calls back to the long lived direct manipulation vs interface agents debate [60]; although there is an effort cost to creating personalizations, there is also a cost when an intelligent system assumes someone's needs and gets them wrong. Although they may appear at odds, we envision end-user programming as a supplement, not a replacement for large VLMs. We envision the two approaches complementing each other in the following ways:

Reducing hallucinations by breaking down problems. Programs can serve as a way to break down visual problems into smaller pieces, avoiding complex questions that might cause models to fail. Just as ProgramAlly crops each image frame to relevant object bounding boxes when running programs, a cropped version of an image could be passed to the VLM to query in a more constrained way. For example, programs could contain subjective adjectives like 'clean' or 'matching'. A VLM could be queried for these items in a constrained way, the answer could be parsed and fed back into the program. VLMs still are not good at reading large chunks of printed text or reasoning about complex scenes, but if the input and output were constrained then they may produce better results.

Balancing ambiguity in language and improving explainability. As discussed in our study findings, language is ambiguous, and programs can help articulate specific intents. Additionally, programs can serve as explicit step-by-step instructions of what a system is doing to come up with a given answer. This could help users better understand the limits of different tools, to better understand and predict why they fail.

Making large VLMs 'live' and creating reusable queries. Current large VLMs take in a static image as input. Yet, as models become faster, running a query on a live camera feed will not be as simple as repeating the question on each frame. Because programs specify what users want to hear and when, they could be used to convert natural language responses into real-time feedback.

7 CONCLUSION

We have presented ProgramAlly, an end-user programming tool for creating custom visual filtering programs. ProgramAlly implements a set of programming interfaces: block-based, natural language, and programming by example. Through a user study of ProgramAlly conducted with 12 blind participants, we demonstrate the promise of end-user programming approaches for creating and customizing AI-based assistive technologies. We observed that users prefer different approaches depending on their experiences and the task, and also note areas where blind end-user programmers may face unique challenges while creating highly visual, camera-based technologies. Overall, ProgramAlly is a step towards supporting blind people in creating personally meaningful assistive technologies.

ACKNOWLEDGMENTS

We sincerely thank our participants for their time, and for sharing their expertise and experiences. We also thank our reviewers for their time and feedback. This research was supported in part by a Google Research Scholar Award. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1841052. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Apple. 2022. Shortcuts User Guide. https://support.apple.com/guide/shortcuts/welcome/ios

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Jeffrey P Bigham, Jeremy T Brudvik, and Bernie Zhang. 2010. Accessibility by demonstration: enabling end users to guide developers to web accessibility solutions. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility.* 35–42.

[4] Jeffrey P Bigham and Patrick Carrington. 2018. Learning from the front: People with disabilities as early adopters of AI. *Proceedings of the 2018 HCIC Human-Computer Interaction Consortium* (2018).

[5] Jeffrey P Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, et al. 2010. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology.* 333–342.

[6] Jeffrey P Bigham and Richard E Ladner. 2007. Accessmonkey: a collaborative scripting framework for web users and developers. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A).* 25–34.

[7] Jeffrey P Bigham, Irene Lin, and Saiph Savage. 2017. The Effects of" Not Knowing What You Don't Know" on Web Accessibility for Blind Web Users. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility.* 101–109.

[8] Erin Brady, Meredith Ringel Morris, Yu Zhong, Samuel White, and Jeffrey P Bigham. 2013. Visual challenges in the everyday lives of blind people. In *Proceedings of the SIGCHI conference on human factors in computing systems.* 2117–2126.

[9] Danielle Bragg, Nicholas Huynh, and Richard E Ladner. 2016. A personalizable mobile sound detector app design for deaf and hard-of-hearing users. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility.* 3–13.

[10] Erin Buehler, Stacy Branham, Abdullah Ali, Jeremy J Chang, Megan Kelly Hofmann, Amy Hurst, and Shaun K Kane. 2015. Sharing is caring: Assistive technology designs on thingiverse. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems.* 525–534.

[11] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping distributed hierarchical web data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology.* 963–975.

[12] Xiang'Anthony' Chen, Jeeeun Kim, Jennifer Mankoff, Tovi Grossman, Stelian Coros, and Scott E Hudson. 2016. Reprise: A design tool for specifying, generating, and customizing 3D printable adaptations on everyday objects. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology.* 29–39.

[13] Tianheng Cheng, Lin Song, Yixiao Ge, Wenyu Liu, Xinggang Wang, and Ying Shan. 2024. YOLO-World: Real-Time Open-Vocabulary Object Detection. *arXiv preprint arXiv:2401.17270* (2024).

[14] Google Cloud. 2022. Optical Character Recognition (OCR) Vision API. https://cloud.google.com/vision/docs/ocr

[15] Google Cloud. 2024. Detect Multiple Objects. https://cloud.google.com/vision/docs/object-localizer

[16] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration.* MIT press.

[17] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering.* 345–356.

[18] Be My Eyes. 2024. Introducing: Be My AI. https://www.bemyeyes.com/blog/introducing-be-my-ai

[19] Krzysztof Z Gajos, Jacob O Wobbrock, and Daniel S Weld. 2007. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *Proceedings of the 20th annual ACM symposium on User interface software and technology.* 231–240.

[20] Bhanuka Gamage, Thanh-Toan Do, Nicholas Seow Chiang Price, Arthur Lowery, and Kim Marriott. 2023. What do Blind and Low-Vision People Really Want from Assistive Smart Devices? Comparison of the Literature with a Focus Study. In *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility.* 1–21.

[21] Alejandra Garrido, Sergio Firmenich, Gustavo Rossi, Julian Grigera, Nuria Medina-Medina, and Ivana Harari. 2012. Personalized web accessibility using client-side refactoring. *IEEE Internet Computing* 17, 4 (2012), 58–66.

[22] Google. 2024. ML Kit Entity Extraction API. https://developers.google.com/ml-kit/language/entity-extraction

[23] Anhong Guo, Xiang 'Anthony' Chen, Haoran Qi, Samuel White, Suman Ghosh, Chieko Asakawa, and Jeffrey P. Bigham. 2016. VizLens: A Robust and Interactive Screen Reader for Interfaces in the Real World. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16).* Association for Computing Machinery, New York, NY, USA, 651–664. https://doi.org/10.1145/2984511.2984518

[24] Danna Gurari, Qing Li, Abigale J Stangl, Anhong Guo, Chi Lin, Kristen Grauman, Jiebo Luo, and Jeffrey P Bigham. 2018. Vizwiz grand challenge: Answering visual questions from blind people. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 3608–3617.

[25] Liwen He, Yifan Li, Mingming Fan, Liang He, and Yuhang Zhao. 2023. A Multi-modal Toolkit to Support DIY Assistive Technology Creation for Blind and Low Vision People. In *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology.* 1–3.

[26] Jaylin Herskovitz, Andi Xu, Rahaf Alharbi, and Anhong Guo. 2023. Hacking, switching, combining: understanding and supporting DIY assistive technology design by blind people. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* 1–17.

[27] Megan Hofmann, Jeffrey Harris, Scott E Hudson, and Jennifer Mankoff. 2016. Helping hands: Requirements for a prototyping methodology for upper-limb prosthetics users. In *Proceedings of the 2016 CHI conference on human factors in computing systems.* 1769–1780.

[28] Amy Hurst and Shaun Kane. 2013. Making" making" accessible. In *Proceedings of the 12th international conference on interaction design and children.* 635–638.

[29] Amy Hurst and Jasmine Tobias. 2011. Empowering individuals with do-it-yourself assistive technology. In *The proceedings of the 13th international ACM SIGACCESS conference on Computers and accessibility.* 11–18.

[30] The Smith-Kettlewell Eye Research Institute. 2024. The Blind Arduino Project. https://www.ski.org/projects/blind-arduino-project

[31] Justin Johnson, Andrej Karpathy, and Li Fei-Fei. 2016. DenseCap: Fully Convolutional Localization Networks for Dense Captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.*

[32] Hernisa Kacorri. 2017. Teachable machines for accessibility. *ACM SIGACCESS Accessibility and Computing* 119 (2017), 10–18.

[33] Hernisa Kacorri, Kris M Kitani, Jeffrey P Bigham, and Chieko Asakawa. 2017. People with visual impairment training personal object recognizers: Feasibility and challenges. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems.* 5839–5849.

[34] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 1–44.

[35] 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.

[36] Varsha Koushik, Darren Guinness, and Shaun K Kane. 2019. Storyblocks: A tangible programming game to create accessible audio stories. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems.* 1–12.

[37] Stacey Kuznetsov and Eric Paulos. 2010. Rise of the expert amateur: DIY projects, communities, and cultures. In *Proceedings of the 6th Nordic conference on human-computer interaction: extending boundaries.* 295–304.

[38] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems.* 6038–6049.

[39] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. *arXiv preprint arXiv:2005.03776* (2020).

[40] Henry Lieberman. 2001. *Your wish is my command: Programming by example.* Morgan Kaufmann.

[41] Hanchao Liu, Wenyuan Xue, Yifei Chen, Dapeng Chen, Xiutian Zhao, Ke Wang, Liping Hou, Rongjun Li, and Wei Peng. 2024. A survey on hallucination in large vision-language models. *arXiv preprint arXiv:2402.00253* (2024).

[42] Google LLC. 2022. Google Assistant. https://play.google.com/store/apps/details?id=com.google.android.apps.googleassistant&hl=en_US&gl=US&pli=1

[43] Daniela Massiceti, Camilla Longden, Agnieszka Slowik, Samuel Wills, Martin Grayson, and Cecily Morrison. 2023. Explaining CLIP's performance disparities on data from blind/low vision users. *arXiv preprint arXiv:2311.17315* (2023).

[44] Janis Lena Meissner, John Vines, Janice McLaughlin, Thomas Nappey, Jekaterina Maksimova, and Peter Wright. 2017. Do-it-yourself empowerment as experienced by novice makers with disabilities. In *Proceedings of the 2017 conference on designing interactive systems.* 1053–1065.

[45] Meta. 2024. Detectron2. https://cloud.google.com/vision/docs/ocr

[46] Microsoft. 2021. Seeing AI. https://www.microsoft.com/en-us/ai/seeing-ai

[47] Microsoft Research. 2021. Microsoft Soundscape – A map delivered in 3D sound. https://www.microsoft.com/en-us/research/product/soundscape/.

[48] Siti Nor Hafizah Mohamad, Ahmed Patel, Rodziah Latih, Qais Qassim, Liu Na, and Yiqi Tew. 2011. Block-based programming approach: challenges and benefits. In *Proceedings of the 2011 international conference on electrical engineering and informatics.* IEEE, 1–5.

[49] Farhani Momotaz, Md Touhidul Islam, Md Ehtesham-Ul-Haque, and Syed Masum Billah. 2021. Understanding screen readers' plugins. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility.* 1–10.

[50] Cecily Morrison, Martin Grayson, Rita Faia Marques, Daniela Massiceti, Camilla Longden, Linda Wen, and Edward Cutrell. 2023. Understanding Personalized Accessibility through Teachable AI: Designing and Evaluating Find My Things for People who are Blind or Low Vision. In *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–12.

[51] Brad A Myers, Amy J Ko, and Margaret M Burnett. 2006. Invited research overview: end-user programming. In *CHI'06 extended abstracts on Human factors in computing systems*. 75–80.

[52] Massachusetts Institute of Technology. 2024. Scratch. https://scratch.mit.edu/

[53] Maulishree Pandey, Sharvari Bondre, Sile O'Modhrain, and Steve Oney. 2022. Accessibility of UI Frameworks and Libraries for Programmers with Visual Impairments. (2022), 10.

[54] 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.

[55] 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.

[56] Ravihansa Rajapakse, Margot Brereton, Paul Roe, and Laurianne Sitbon. 2014. Designing with people with disabilities: Adapting best practices of DIY and organizational approaches. In *Proceedings of the 26th Australian Computer-Human Interaction Conference on Designing Futures: the Future of Design*. 519–522.

[57] Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. 2023. Real-Time Flying Object Detection with YOLOv8. arXiv:2305.09972 [cs.CV]

[58] GPT-4 Technical Report. 2023. OpenAI. https://openai.com/research/gpt-4

[59] Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 207–214.

[60] Ben Shneiderman and Pattie Maes. 1997. Direct manipulation vs. interface agents. *Interactions* 4, 6 (nov 1997), 42–61. https://doi.org/10.1145/267505.267514

[61] David Sloan, Matthew Tylee Atkinson, Colin Machin, and Yunqiu Li. 2010. The potential of adaptive interfaces as an accessibility aid for older web users. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*. 1–10.

[62] Abigale Stangl, Nitin Verma, Kenneth R Fleischmann, Meredith Ringel Morris, and Danna Gurari. 2021. Going beyond one-size-fits-all image descriptions to satisfy the information wants of people who are blind or have low vision. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 1–15.

[63] Theresa Jean Tanenbaum, Amanda M Williams, Audrey Desjardins, and Karen Tanenbaum. 2013. Democratizing technology: pleasure, utility and expressiveness in DIY and maker practice. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 2603–2612.

[64] Lida Theodorou, Daniela Massiceti, Luisa Zintgraf, Simone Stumpf, Cecily Morrison, Edward Cutrell, Matthew Tobias Harris, and Katja Hofmann. 2021. Disability-first dataset creation: Lessons from constructing a dataset for teachable object recognition with blind and low vision data collectors. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 1–12.

[65] Ultralytics. 2024. YOLO-World: Real-Time Open Vocabulary Object Detection. https://docs.ultralytics.com/models/yolo-world/

[66] Ultralytics. 2024. YOLOv8. https://docs.ultralytics.com/models/yolov8/

[67] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[68] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25.

[69] 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.

[70] Jeffrey Wong and Jason I Hong. 2007. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1435–1444.

[71] Momona Yamagami, Alexandra A Portnova-Fahreeva, Junhan Kong, Jacob O Wobbrock, and Jennifer Mankoff. 2023. How Do People with Limited Movement Personalize Upper-Body Gestures? Considerations for the Design of Personalized and Accessible Gesture Interfaces. In *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–15.

[72] Yuhang Zhao, Sarit Szpiro, Jonathan Knighten, and Shiri Azenkot. 2016. CueSee: exploring visual cues for people with low vision to facilitate a visual search task. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 73–84.