PUMICE: A Multi-Modal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations

Toby Jia-Jun Li[1], **Marissa Radensky**[2], **Justin Jia**[1], **Kirielle Singarajah**[1], **Tom M. Mitchell**[1], **Brad A. Myers**[1]
[1]Carnegie Mellon University, [2]Amherst College
{tobyli, tom.mitchell, bam}@cs.cmu.edu, {justinj1, ksingara}@andrew.cmu.edu, mradensky19@amherst.edu

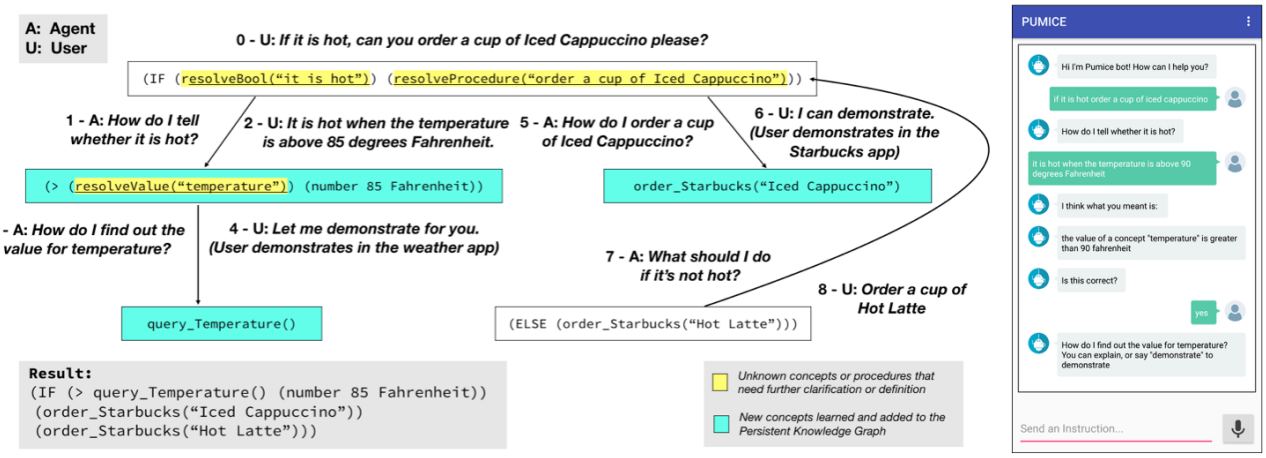

Figure 1. Example structure of how PUMICE learns the concepts and procedures in the command "If it's hot, order a cup of Iced Cappuccino." The numbers indicate the order of utterances. The screenshot on the right shows the conversational interface of PUMICE. In this interactive parsing process, the agent learns how to query the current temperature, how to order any kind of drink from Starbucks, and the generalized concept of "hot" as "a temperature (of something) is greater than another temperature".

ABSTRACT

Natural language programming is a promising approach to enable end users to instruct new tasks for intelligent agents. However, our formative study found that end users would often use unclear, ambiguous or vague concepts when naturally instructing tasks in natural language, especially when specifying conditionals. Existing systems have limited support for letting the user teach agents new concepts or explaining unclear concepts. In this paper, we describe a new multi-modal domain-independent approach that combines natural language programming and programming-by-demonstration to allow users to first naturally describe tasks and associated conditions at a high level, and then collaborate with the agent to recursively resolve any ambiguities or vagueness through conversations and demonstrations. Users can also define new procedures and concepts by demonstrating and referring to contents within GUIs of existing mobile apps. We demonstrate this approach in PUMICE, an end-user programmable agent that implements this approach. A lab study with 10 users showed its usability.

UIST '19, October 20-23, 2019, New Orleans, LA, USA.

ACM ISBN 978-1-4503-6816-2/19/10 ...$15.00.
http://dx.doi.org/10.1145/3332165.3347899

CCS Concepts

•**Human-centered computing** → **Natural language interfaces;**

Author Keywords

Programming by Demonstration; Natural Language Programming; End User Development; Multi-modal Interaction.

INTRODUCTION

The goal of end user development (EUD) is to empower users with little or no programming expertise to program [43]. Among many EUD applications, a particularly useful one would be task automation, through which users program intelligent agents to perform tasks on their behalf [31]. To support such EUD activities, a major challenge is to help non-programmers to specify conditional structures in programs. Many common tasks involve conditional structures, yet they are difficult for non-programmers to correctly specify using existing EUD techniques due to the great distance between how end users think about the conditional structures, and how they are represented in programming languages [41].

According to Green and Petre's cognitive dimensions of notations [13], the closer the programming world is to the problem world, the easier the problem-solving ought to be. This *closeness of mapping* is usually low in conventional and EUD programming languages, as they require users to think about their tasks very differently from how they would think

about them in familiar contexts [41], making programming particularly difficult for end users who are not familiar with programming languages and "computational thinking" [49]. To address this issue, the concept of *natural programming* [37, 38] has been proposed to create techniques and tools that match more closely the ways users think.

Natural language programming is a promising technique for bridging the gap between user mental models of tasks and programming languages [34]. It should have a low learning barrier for end users, under the assumption that the majority of end users can already communicate procedures and structures for familiar tasks through natural language conversations [25, 41]. Speech is also a natural input modality for users to describe desired program behaviors [40]. However, existing natural language programming systems are not adequate for supporting end user task automation in domain-general tasks. Some prior systems (e.g., [44]) directly translate user instructions in natural language into conventional programming languages like Java. This approach requires users to use unambiguous language with fixed structures similar to those in conventional programming languages. Therefore, it does not match the user's existing mental model of tasks, imposing significant learning barriers and high cognitive demands on end users.

Other natural language programming approaches (e.g., [3, 11, 17, 46]) restricted the problem space to specific task domains, so that they could constrain the space and the complexity of target program statements in order to enable the understanding of flexible user utterances. Such restrictions are due to the limited capabilities of existing natural language understanding techniques – they do not yet support robust understanding of utterances across diverse domains without extensive training data and structured prior knowledge within each domain.

Another difficult problem in natural language programming is supporting the instruction of *concepts*. In our study (details below in the Formative Study section), we found that end users often refer to ambiguous or vague concepts (e.g., **cold** weather, **heavy** traffic) when naturally instructing a task. Moreover, even if a concept may seem clear to a human, an agent may still not understand it due to the limitations in its natural language understanding techniques and pre-defined ontology.

In this paper, we address the research challenge of enabling end users to augment domain-independent task automation scripts with conditional structures and new concepts through a combination of natural language programming and programming by demonstration (PBD). To support programming for tasks in diverse domains, we leverage the graphical user interfaces (GUIs) of existing third-party mobile apps as a medium, where procedural actions are represented as sequences of GUI operations, and declarative concepts can be represented through references to GUI contents. This approach supports EUD for a wide range of tasks, provided that these tasks can be performed with one or more existing third-party mobile apps.

We took a *user-centered design* approach, first studying how end users naturally describe tasks with conditionals in natural language in the context of mobile apps, and what types of tasks they are interested in automating. Based on insights from this study, we designed and implemented an end-user-programmable conversational agent named PUMICE[1] that allows end users to program tasks with flexible conditional structures and new concepts across diverse domains through spoken natural language instructions and demonstrations.

PUMICE extends our previous SUGILITE [20] system. A key novel aspect of PUMICE's design is that it allows users to first describe the desired program behaviors and conditional structures naturally in natural language at a high level, and then collaborate with an intelligent agent through multi-turn conversations to explain and to define any ambiguities, concepts and procedures in the initial description as needed in a top-down fashion. Users can explain new concepts by referring to either previously defined concepts, or to the contents of the GUIs of third-party mobile apps. Users can also define new procedures by demonstrating using third-party apps [20]. Such an approach facilitates effective program reuse in automation authoring, and provides support for a wide range of application domains, which are two major challenges in prior EUD systems. The results from the motivating study suggest that this paradigm is not only feasible, but also natural for end users, which was supported by our summative lab usability study.

We build upon recent advances in natural language processing (NLP) to allow PUMICE's semantic parser to learn from users' flexible verbal expressions when describing desired program behaviors. Through PUMICE's mixed-initiative conversations with users, an underlying persistent knowledge graph is dynamically updated with new procedural (i.e., actions) and declarative (i.e., concepts and facts) knowledge introduced by users, allowing the semantic parser to improve its understanding of user utterances over time. This structure also allows for effective reuse of user-defined procedures and concepts at a fine granularity, reducing user effort in EUD.

PUMICE presents a multi-modal interface, through which users interact with the system using a combination of demonstrations, pointing, and spoken commands. Users may use any modality that they choose, so they can leverage their prior experience to minimize necessary training [30]. This interface also provides users with guidance through a mix of visual aids and verbal directions through various stages in the process to help users overcome common challenges and pitfalls identified in the formative study, such as the omission of else statements, the difficulty in finding correct GUI objects for defining new concepts, and the confusion in specifying proper data descriptions for target GUI objects. A summative lab usability study with 10 participants showed that users with little or no prior programming expertise could use PUMICE to program automation scripts for 4 tasks derived from real-world scenarios. Participants also found PUMICE easy and natural to use.

This paper presents the following three primary contributions:

1. A formative study showing the characteristics of end users' natural language instructions for tasks with conditional structures in the context of mobile apps.

[1] PUMICE is a type of volcanic rock. It is also an acronym for **P**rogramming in a **U**ser-friendly **M**ultimodal **I**nterface through **C**onversations and **E**xamples

2. A multi-modal conversational approach for the EUD of task automation motivated by the aforementioned formative study, with the following major advantages:
   (a) The top-down conversational structure allows users to **naturally** start with describing the task and its conditionals at a high-level, and then recursively clarify ambiguities, explain unknown concepts and define new procedures through conversations.
   (b) The agent learns new procedural and declarative knowledge through explicit instructions from users, and stores them in a persistent knowledge graph, facilitating effective **reusability and generalizability** of learned knowledge.
   (c) The agent learns concepts and procedures in **various task domains** while having a **low learning barrier** through its **multi-modal approach** that supports references and demonstrations using the contents of third-party apps' GUIs.
3. The PUMICE system: an implementation of this approach, along with a user study evaluating its usability.

BACKGROUND AND RELATED WORK

This research builds upon prior work from many different sub-disciplines across human-computer interaction (HCI), software engineering (SE), and natural language processing (NLP). In this section, we focus on related work on three topics: (1) natural language programming; (2) programming by demonstration; and (3) the multi-modal approach that combines natural language inputs with demonstrations.

Natural Language Programming

PUMICE uses natural language as the primary modality for users to program task automation. The idea of using natural language inputs for programming has been explored for decades [4, 6]. In the NLP and AI communities, this idea is also known as learning by instruction [3, 27].

The foremost challenge in supporting natural language programming is dealing with the inherent ambiguities and vagueness in natural language [48]. To address this challenge, one prior approach was to constrain the structures and expressions in the user's language to similar formulations of conventional programming languages (e.g., [4, 44]), so that user inputs can be directly translated into programming statements. This approach is not adequate for EUD, as it has a high learning barrier for users without programming expertise.

Another approach for handling ambiguities and vagueness in natural language inputs is to seek user clarification through conversations. For example, Iris [11] asks follow-up questions and presents possible options through conversations when the initial user input is incomplete or unclear. This approach lowers the learning barrier for end users, as it does not require them to clearly define everything up front. It also allows users to form complex commands by combining multiple natural language instructions in conversational turns under the guidance of the system. PUMICE also adopts the use of multi-turn conversations as a key strategy in handling ambiguities and vagueness in user inputs. However, a key difference between PUMICE and other conversational instructable agents is that PUMICE is domain-independent. All conversational instructable agents need to map the user's inputs onto existing concepts, procedures and system functionalities supported by the agent, and to have natural language understanding mechanisms and training data in each task domain. Because of this constraint, existing agents limit their supported tasks to one or a few pre-defined domains, such as data science [11], email processing [3, 46], or database queries [17].

PUMICE supports learning concepts and procedures from existing third-party mobile apps regardless of the task domains. End users can create new concepts with PUMICE by referencing relevant information shown in app GUIs, and define new procedures by demonstrating with existing apps. This approach allows PUMICE to support a wide range of tasks from diverse domains as long as the corresponding mobile apps are available. This approach also has a low learning barrier because end users are already familiar with the capabilities of mobile apps and how to use them. In comparison, with prior instructable agents, it is often unclear what concepts, procedures and functionalities already exist to be used as "building blocks" for developing new ones.

Programming by Demonstration

PUMICE uses the programming by demonstration (PBD) technique to enable end users to define concepts by referring to the contents of GUIs of third-party mobile apps, and teach new procedures through demonstrations with those apps. PBD is a natural way of supporting EUD with a low learning barrier [10, 24]. Many domain-specific PBD tools have been developed in the past in various domains, such as text editing (e.g., [18]), photo editing (e.g., [12]), web scraping (e.g., [8]), smart home control (e.g., [22]) and robot control (e.g., [2]).

PUMICE supports domain-independent PBD by using the GUIs of third-party apps for task automation and data extraction. Similar approaches have also been used in prior systems. For example, SUGILITE [20], KITE [23] and APPINITE [21] use mobile app GUIs, CoScripter [19], d.mix [15], Vegemite [28] and PLOW [1] use web interfaces, and HILC [16] and Sikuli [50] use desktop GUIs. Compared to those, PUMICE is the only one that can learn concepts as generalized knowledge, and the only one that supports creating conditionals from natural language instructions. Sikuli [50] allows users to create conditionals in a scripting language, which is not suitable for end users without programming expertise.

The Multi-Modal Approach

A central challenge in PBD is generalization. A PBD agent should go beyond literal record-and-replay macros, and be able to perform similar tasks in new contexts [10, 24]. This challenge is also a part of the program synthesis problem. An effective way of addressing this challenge is through multi-modal interaction [40]. Demonstrations can clearly communicate *what* the user does, but not *why* the user does this and *how* the user wants to do this in different contexts. On the other hand, natural language instructions can often reflect the user's underlying intent (*why*) and preferences (*how*), but they are

usually ambiguous or unclear. This is where grounding natural language instructions with concrete GUI demonstrations can help.

This *mutual disambiguation* approach [39] in multi-modal interaction has been proposed and used in many previous systems. This approach leverages *repetition* in a different modality for mediation [32]. Particularly for PBD generalization, SUGILITE [20] and PLOW [1] use natural language inputs to identify parameterization in demonstrations, and APPINITE [21] uses natural language explanations of intents to resolve the "data description" [10] for demonstrated actions.

PUMICE builds upon this prior work, and extends the multi-modal approach to support learning concepts involved in demonstrated tasks. The learned concepts can also be generalized to new task domains, as described in later sections. The prior multi-modal PBD systems also use demonstration as the main modality. In comparison, PUMICE uses natural language conversation as the main modality, and uses demonstration for grounding unknown concepts, values, and procedures after they have been broken down and explained in conversations.

FORMATIVE STUDY

We took a *user-centered* approach [36] for designing a natural end-user development system [37]. We first studied how end users naturally communicate tasks with declarative concepts and control structures in natural language for various tasks in the mobile app context through a formative study on Amazon Mechanical Turk with 58 participants (41 of which are non-programmers; 38 men, 19 women, 1 non-binary person).

Each participant was presented with a graphical description of an everyday task for a conversational agent to complete in the context of mobile apps. All tasks had distinct conditions for a given task so that each task should be performed differently under different conditions, such as playing different genres of music based on the time of the day. Each participant was assigned to one of 9 tasks. To avoid biasing the language used in the responses, we used the Natural Programming Elicitation method [36] by showing graphical representations of the tasks with limited text in the prompts. Participants were asked how they would verbally instruct the agent to perform the tasks, so that the system could understand the differences among the conditions and what to do in each condition. Each participant was first trained using an example scenario and the corresponding example verbal instructions.

To study whether having mobile app GUIs can affect users' verbal instructions, we randomly assigned participants into one of two groups. For the experimental group, participants instructed agents to perform the tasks while looking at relevant app GUIs. Each participant was presented with a mobile app screenshot with arrows pointing to the screen component that contained the information pertinent to the task condition. Participants in the control group were not shown app GUIs. At the end of each study session, we also asked the participants to come up with another task scenario of their own where an agent should perform differently in different conditions.

The participants' responses were analyzed by two independent coders using open coding [47]. The inter-rater agreement [9] was $\kappa = 0.87$, suggesting good agreement. 19% of responses were excluded from the analysis for quality control due to the lack of efforts in the responses, question misunderstandings, and blank responses.

We report the most relevant findings which motivated the design of PUMICE next.

App GUI Grounding Reduces Unclear Concept Usage

We analyzed whether each user's verbal instruction for the task provided a clear definition of the conditions in the task. In the control group (instructing without seeing app screenshots), 33% of the participants used ambiguous, unclear or vague concepts in the instructions, such as "*If it is daytime, play upbeat music...*" which is ambiguous as to when the user considers it to be "daytime." This is despite the fact that the example instructions they saw had clearly defined conditions.

Interestingly, for the experimental group, where each participant was provided an app screenshot displaying specific information relevant to the task's condition, fewer participants (9%) used ambiguous or vague concepts (this difference is statistically significant with $p < 0.05$), while the rest clearly defined the condition (e.g., "*If the current time is before 7 pm...*"). These results suggest that end users naturally use ambiguous and vague concepts when verbally instructing task logic, but showing users relevant mobile app GUIs with concrete instances of the values can help them ground the concepts, leading to fewer ambiguities and vagueness in their descriptions. The implication is that a potentially effective approach to avoiding unclear utterances for agents is to guide users to explain them in the context of app GUIs.

Unmet User Expectation of Common Sense Reasoning

We observed that participants often expected and assumed the agent to have the capability of understanding and reasoning with common sense knowledge when instructing tasks. For example, one user said, "*if the day is a weekend*". The agent would therefore need to understand the concept of "weekend" (i.e., how to know today's day of the week, and what days count as "weekend") to resolve this condition. Similarly when a user talked about "sunset time", he expected the agent to know what it meant, and how to find out its value.

However, the capability for common sense knowledge and reasoning is very limited in current agents, especially across many diverse domains, due to the spotty coverage and unreliable inference of existing common sense knowledge systems. Managing user expectation and communicating the agent's capability is also a long-standing unsolved challenge in building interactive intelligent systems [26]. A feasible workaround is to enable the agent to ask users to ground new concepts to existing contents in apps when they come up, and to build up knowledge of concepts over time through its interaction with users.

Frequent Omission of Else Statements

In the study, despite all provided example responses containing else statements, 18% of the 39 descriptions from users omitted an else statement when it was expected. "*Play upbeat music until 8pm every day,*" for instance, may imply that the user

desires an alternative genre of music to be played at other times. Furthermore, 33% omitted an else statement when a person would be expected to infer an else statement, such as: "*If a public transportation access point is more than half a mile away, then request an Uber*," which implies using public transportation otherwise. This might be a result of the user's expectation of common sense reasoning capabilities. The user omits what they expect the agent can infer to avoid prolixity, similar to patterns in human-human conversations [14].

These findings suggest that end users will often omit appropriate else statements in their natural language instructions for conditionals. Therefore, the agent should proactively ask users about alternative situations in conditionals when appropriate.

PUMICE

Motivated by the formative study results, we designed the PUMICE agent that supports understanding ambiguous natural language instructions for task automation by allowing users to recursively define any new, ambiguous or vague concepts in a multi-level top-down process.

Example Scenario

This section shows an example scenario to illustrate how PUMICE works. Suppose a user starts teaching the agent a new task automation rule by saying, "*If it's hot, order a cup of Iced Cappuccino.*" We also assume that the agent has no prior knowledge about the relevant task domains (weather and coffee ordering). Due to the lack of domain knowledge, the agent does not understand "it's hot" and "order a cup of Iced Cappuccino". However, the agent can recognize the conditional structure in the utterance (the parse for Utterance 0 in Figure 1) and can identify that "it's hot" should represent a Boolean expression while "order a cup of Iced Cappuccino" represents the action to perform if the condition is true.

PUMICE's semantic parser can mark unknown parts in user utterances using typed `resolve...()` functions, as marked in the yellow highlights in the parse for Utterance 0 in Figure 1. The PUMICE agent then proceeds to ask the user to further explain these concepts. It asks, "*How do I tell whether it's hot?*" since it has already figured out that "it's hot" should be a function that returns a Boolean value. The user answers "*It is hot when the temperature is above 85 degrees Fahrenheit.*", as shown in Utterance 2 in Figure 1. PUMICE understands the comparison (as shown in the parse for Utterance 2 in Figure 1), but does not know the concept of "temperature", only knowing that it should be a numeric value comparable to 85 degrees Fahrenheit. Hence it asks, "*How do I find out the value for temperature?*", to which the user responds, "*Let me demonstrate for you.*"

Here the user can demonstrate the procedure of finding the current temperature by opening the weather app on the phone, and pointing at the current reading of the weather. To assist the user, PUMICE uses a visualization overlay to highlight any GUI objects on the screen that fit into the comparison (i.e., those that display a value comparable to 85 degrees Fahrenheit). The user can choose from these highlighted objects (see Figure 2 for an example). Through this demonstration, PUMICE learns a reusable procedure `query_Temperature()` for getting the current value for the new concept *temperature*, and stores it in a persistent knowledge graph so that it can be used in other tasks. PUMICE confirms with the user every time it learns a new concept or a new rule, so that the user is aware of the current state of the system, and can correct any errors (see the Error Recovery and Backtracking section for details).

For the next phase, PUMICE has already determined that "order a cup of Iced Cappuccino" should be an action triggered when the condition "it's hot" is true, but does not know how to perform this action (also known as intent fulfillment in chatbots [23]). To learn how to perform this action, it asks, "*How do I order a cup of Iced Cappuccino?*", to which the user responds, "*I can demonstrate.*" The user then proceeds to demonstrate the procedure of ordering a cup of Iced Cappuccino using the existing app for Starbucks (a coffee chain). From the user demonstration, PUMICE can figure out that "Iced Cappuccino" is a task parameter, and can learn the generalized procedure `order_Starbucks()` for ordering any item in the Starbucks app, as well as a list of available items to order in the Starbucks app by looking through the Starbucks app's menus, using the underlying SUGILITE framework [9] for processing the task recording.

Finally, PUMICE asks the user about the else condition by saying, "*What should I do if it's not hot?*" Suppose the user says "*Order a cup of Hot Latte*," then the user will not need to demonstrate again because PUMICE can recognize "hot latte" as an available parameter for the known `order_Starbucks()` procedure.

Design Features

In this section, we discuss several of PUMICE's key design features in its user interactions, and how they were motivated by results of the formative study.

Support for Concept Learning

In the formative study, we identified two main challenges in regards to concept learning. First, user often naturally use intrinsically unclear or ambiguous concepts when instructing intelligent agents (e.g., "*register for easy courses*", where the definition of "easy" depends on the context and the user preference). Second, users expect agents to understand common-sense concepts that the agents may not know. To address these challenges, we designed and implemented the support for concept learning in PUMICE. PUMICE can detect and learn three kinds of unknown components in user utterances: *procedures*, *Boolean concepts*, and *value concepts*. Because PUMICE's support for procedure learning is unchanged from the underlying SUGILITE mechanisms [21, 20], in this section, we focus on discussing how PUMICE learns Boolean concepts and value concepts.

When encountering an unknown or unclear concept in the utterance parsing result, PUMICE first determines the concept type based on the context. If the concept is used as a condition (e.g., "if **it is hot**"), then it should be of Boolean type. Similarly, if a concept is used where a value is expected (e.g., "if the **current temperature** is above 70°F" or "set the AC to the **current temperature**"), then it will be marked as a value concept. Both kinds of concepts are represented as typed `resolve()`

functions in the parsing result (shown in Figure 1), indicating that they need to be further resolved down the line. This process is flexible. For example, if the user clearly defines a condition without introducing unknown or unclear concepts, then PUMICE will not need to ask follow-up questions for concept resolution.

PUMICE recursively executes each `resolve()` function in the parsing result in a depth-first fashion. After a concept is fully resolved (i.e., all concepts used for defining it have been resolved), it is added to a persistent knowledge graph (details in the System Implementation section), and a link to it replaces the `resolve()` function. From the user's perspective, when a `resolve()` function is executed, the agent asks a question to prompt the user to further explain the concept. When resolving a Boolean concept, PUMICE asks, "*How do I know whether [concept_name]?*" For resolving a value concept, PUMICE asks, "*How do I find out the value of [concept_name]?*"

To explain a new Boolean concept, the user may verbally refer to another Boolean concept (e.g., "traffic is heavy" means "commute takes a long time") or may describe a Boolean expression (e.g., "the commute time is longer than 30 minutes"). When describing the Boolean expression, the user can use flexible words (e.g., colder, further, more expensive) to describe the relation (i.e., greater than, less than, and equal to). As explained previously, if any new Boolean or value concepts are used in the explanation, PUMICE will recursively resolve them. The user can also use more than one unknown value concepts, such as "if the price of a Uber is greater than the price of a Lyft" (Uber and Lyft are both popular ridesharing apps).

Similar to Boolean concepts, the user can refer to another value concept when explaining a value concept. When a value concept is concrete and available in a mobile app, the user can also demonstrate how to query the value through app GUIs. The formative study has suggested that this multi-modal approach is effective and feasible for end users. After users indicate that they want to demonstrate, PUMICE switches to the home screen of the phone, and prompts the user to demonstrate how to find out the value of the concept.

To help the user with value concept demonstrations, PUMICE highlights possible items on the current app GUI if the type of the target concept can be inferred from the type of the constant value, or using the type of value concept to which it is being compared (see Figure 2). For example, in the aforementioned "commute time" example, PUMICE knows that "commute time" should be a duration, because it is comparable to the constant value "30 minutes". Once the user finds the target value in an app, they can long press on the target value to select it and indicate it as the target value. PUMICE uses an interaction proxy overlay [53] for recording, so that it can record *all* values visible on the screen, not limited to the selectable or clickable ones. PUMICE can extract these values from the GUI using the screen scraping mechanism in the underlying SUGILITE framework [20]. Once the target value is selected, PUMICE stores the procedure of navigating to the screen where the target value is displayed and finding the target value on the screen into its persistent knowledge graph as a value query, so that this query can be used whenever the underlying value

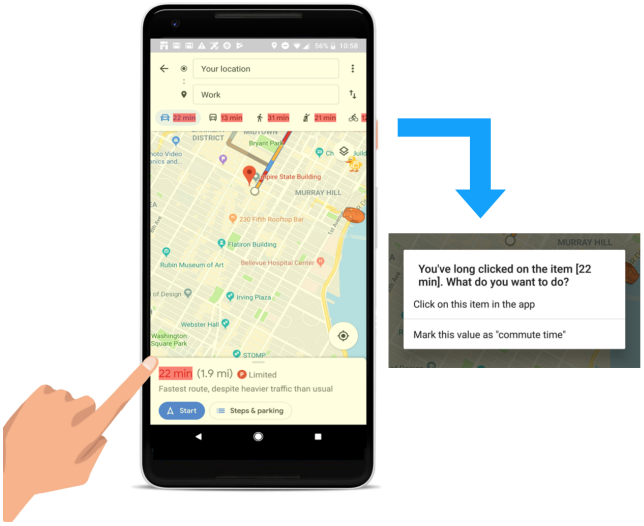

Figure 2. The user teaches the value concept "commute time" by demonstrating querying the value in Google Maps. The red overlays highlight all durations it was able to identify on the Google Maps GUI.

is needed. After the value concept demonstration, PUMICE switches back to the conversational interface and continues to resolve the next concept if needed.

Concept Generalization and Reuse

Once concepts are learned, another major challenge is to generalize them so that they can be reused correctly in different contexts and task domains. This is a key design goal of PUMICE. It should be able to learn concepts at a fine granularity, and reuse parts of existing concepts as much as possible to avoid asking users to make redundant demonstrations. In our previous works on generalization for PBD, we focused on generalizing procedures, specifically learning parameters [20] and intents for underlying operations [21]. We have already deployed these existing generalization mechanisms in PUMICE, but in addition, we also explored the generalization of Boolean concepts and value concepts.

When generalizing Boolean concepts, PUMICE assumes that the Boolean operation stays the same, but the arguments may differ. For example, for the concept "hot" in Figure 1, it should still mean that a temperature (of something) is greater than another temperature. But the two in comparison can be different constants, or from different value queries. For example, suppose after the interactions in Figure 1, the user instructs a new rule "*if the oven is hot, start the cook timer.*" PUMICE can recognize that "hot" is a concept that has been instructed before in a different context, so it asks "*I already know how to tell whether it is hot when determining whether to order a cup of Iced Cappuccino. Is it the same here when determining whether to start the cook timer?*" After responding "No", the user can instruct how to find out the temperature of the oven, and the new threshold value for "hot" either by instructing a new value concept, or using a constant value.

The generalization mechanism for value concepts works similarly. PUMICE supports value concepts that share the same name to have different query implementations for different

task contexts. For example, following the "if the oven is hot, start the cook timer" example, suppose the user defines "hot" for this new context as "*The temperature is above 400 degrees.*" PUMICE realizes that there is already a value concept named "temperature", so it will ask "*I already know how to find out the value for temperature using the Weather app. Should I use that for determining whether the oven is hot?*", to which the user can say "No" and then demonstrate querying the temperature of the oven using the corresponding app (assuming the user has a smart oven with an in-app display of its temperature).

This mechanism allows concepts like "hot" to be reused at three different levels: (1) exactly the same (e.g., the temperature of the weather is greater than 85°F); (2) different threshold (e.g., the temperature of the weather is greater than x); and (3) different value query (e.g., the temperature of *something else* is greater than x).

Error Recovery and Backtracking

Like all other interactive EUD systems, it is crucial for PUMICE to properly handle errors, and to backtrack from errors in speech recognition, semantic parsing, generalizations, and inferences of intent. We iteratively tested early prototypes of PUMICE with users through early usability testing, and developed the following mechanisms to support error recovery and backtracking in PUMICE.

To mitigate semantic parsing errors, we implemented a mixed-initiative mechanism where PUMICE can ask users about *components* within the parsed expression if the parsing result is considered incorrect by the user. Because parsing result candidates are all typed expressions in PUMICE's internal functional domain-specific language (DSL) as a conditional, Boolean, value, or procedure, PUMICE can identify components in a parsing result that it is less confident about by comparing the top candidate with the alternatives and confidence scores, and ask the user about them.

For example, suppose the user defines a Boolean concept "good restaurant" with the utterance "the rating is better than 2". The parser is uncertain about the comparison operator in this Boolean expression, since "better" can mean either "greater than" or "less than" depending on the context. It will ask the user "*I understand you are trying to compare the value concept 'rating' and the value '2', should 'rating' be greater than, or less than '2'?*" The same technique can also be used to disambiguate other parts of the parsing results, such as the argument of `resolve()` functions (e.g., determining whether the unknown procedure should be "order a cup of Iced Cappuccino" or "order a cup" for Utterance 0 in Figure 1).

PUMICE also provides an "undo" function to allow the user to backtrack to a previous conversational state in case of incorrect speech recognition, incorrect generalization, or when the user wants to modify a previous input. Users can either say that they want to go back to the previous state, or click on an "undo" option in PUMICE's menu (activated from the option icon on the top right corner on the screen shown in Figure 1).

System Implementation

We implemented the PUMICE agent as an Android app. The app was developed and tested on a Google Pixel 2 XL phone running Android 8.0. PUMICE does *not* require the root access to the phone, and should run on any phone running Android 6.0 or higher. PUMICE is open-sourced on GitHub[2].

Semantic Parsing

We built the semantic parser for PUMICE using the SEMPRE framework [5]. The parser runs on a remote Linux server, and communicates with the PUMICE client through an HTTP RESTful interface. It uses the Floating Parser architecture, which is a grammar-based approach that provides more flexibility without requiring hand-engineering of lexicalized rules like synchronous CFG or CCG based semantic parsers [42]. This approach also provides more interpretable results and requires less training data than neural network approaches (e.g., [51, 52]). The parser parses user utterances into expressions in a simple functional DSL we created for PUMICE.

A key feature we added to PUMICE's parser is allowing typed `resolve()` functions in the parsing results to indicate unknown or unclear concepts and procedures. This feature adds interactivity to the traditional semantic parsing process. When this `resolve()` function is called at runtime, the front-end PUMICE agent asks the user to verbally explain, or to demonstrate how to fulfill this `resolve()` function. If an unknown concept or procedure is resolved through verbal explanation, the parser can parse the new explanation into an expression of its original type in the target DSL (e.g., an explanation for a Boolean concept is parsed into a Boolean expression), and replace the original `resolve()` function with the new expression. The parser also adds relevant utterances for existing concepts and procedures, and visible text labels from demonstrations on third-party app GUIs to its set of lexicons, so that it can understand user references to those existing knowledge and in-app contents. PUMICE's parser was trained on rich features that associate lexical and syntactic patterns (e.g., unigrams, bigrams, skipgrams, part-of-speech tags, named entity tags) of user utterances with semantics and structures of the target DSL over a small number of training data ($n = 905$) that were mostly collected and enriched from the formative study.

Demonstration Recording and Replaying

PUMICE uses our open-sourced SUGILITE [20] framework to support its demonstration recording and replaying. SUGILITE provides action recording and replaying capabilities on third-party Android apps using the Android Accessibility API. SUGILITE also provides the support for parameterization of sequences of actions (e.g., identifying "Iced Cappuccino" as a parameter and "Hot Latte" as an alternative value in the example in Figure 1), and the support for handling minor GUI changes in apps. Through SUGILITE, PUMICE operates well on most native Android apps, but may have problems working with web apps and apps with special graphic engines (e.g., games). It currently does not support recording gestural and sensory inputs (e.g., rotating the phone) either.

Knowledge Representations

PUMICE maintains two kinds of knowledge representations: a continuously refreshing UI snapshot graph representing third-

[2]https://github.com/tobyli/Sugilite_development

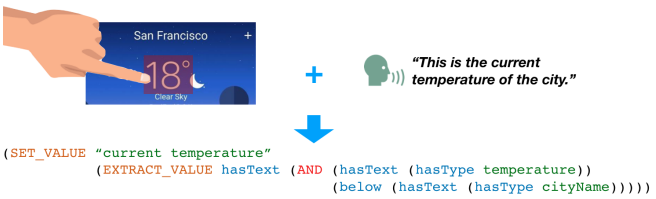

Figure 3. An example showing how PUMICE parses the user's demonstrated action and verbal reference to an app's GUI content into a SET_VALUE statement with a query over the UI snapshot graph when resolving a new value concept "current temperature"

party app contexts for demonstration, and a persistent knowledge base for storing learned procedures and concepts.

The purpose of the UI snapshot graph is to support understanding the user's references to app GUI contents in their verbal instructions. The UI snapshot graph mechanism used in PUMICE was extended from APPINITE [21]. For every state of an app's GUI, a UI snapshot graph is constructed to represent *all* visible and invisible GUI objects on the screen, including their types, positions, accessibility labels, text labels, various properties, and spatial relations among them. We used a lightweight semantic parser from the Stanford CoreNLP [33] to extract types of structured data (e.g., temperature, time, date, phone number) and named entities (e.g., city names, people's names). When handling the user's references to app GUI contents, PUMICE parses the original utterances into queries over the current UI snapshot graph (example in Figure 3). This approach allows PUMICE to generate flexible queries for value concepts and procedures that accurately reflect user intents, and which can be reliably executed in future contexts.

The persistent knowledge base stores all procedures, concepts, and facts PUMICE has learned from the user. Procedures are stored as SUGILITE [20] scripts, with the corresponding trigger utterances, parameters, and possible alternatives for each parameter. Each Boolean concept is represented as a set of trigger utterances, Boolean expressions with references to the value concepts or constants involved, and contexts (i.e., the apps and actions used) for each Boolean expression. Similarly, the structure for each stored value concept includes its triggering utterances, demonstrated value queries for extracting target values from app GUIs, and contexts for each value query.

USER STUDY

We conducted a lab study to evaluate the usability of PUMICE. In each session, a user completed 4 tasks. For each task, the user instructed PUMICE to create a new task automation, with the required conditionals and new concepts. We used a task-based method to specifically test the usability of PUMICE's design, since the motivation for the design derives from the formative study results. We did not use a control condition, as we could not find other tools that can feasibly support users with little programming expertise to complete the target tasks.

Participants

We recruited 10 participants (5 women, 5 men, ages 19 to 35) for our study. Each study session lasted 40 to 60 minutes. We compensated each participant $15 for their time. 6 participants were students in two local universities, and the other 4 worked different technical, administrative or managerial jobs. All participants were experienced smartphone users who had been using smartphones for at least 3 years. 8 out of 10 participants had some prior experience of interacting with conversational agents like Siri, Alexa and Google Assistant.

We asked the participants to report their programming experience on a five-point scale from "never programmed" to "experienced programmer". Among our participants, there were 1 who had never programmed, 5 who had only used end-user programming tools (e.g., Excel functions, Office macros), 1 novice programmer with experience equivalent to 1-2 college level programming classes, 1 programmer with 1-2 years of experience, and 2 programmers with more than 3 years of experience. In our analysis, we will label the first two groups "non-programmers" and the last three groups "programmers".

Procedure

At the beginning of each session, the participant received a 5-minute tutorial on how to use PUMICE. In the tutorial, the experimenter demonstrated an example of teaching PUMICE to check the bus schedule when "it is late", and "late" was defined as "current time is after 8pm" through a conversation with PUMICE. The experimenter then showed how to demonstrate to PUMICE finding out the current time using the Clock app.

Following the tutorial, the participant was provided a Google Pixel 2 phone with PUMICE and relevant third-party apps installed. The experimenter showed the participant the available apps, and made sure that the participant understood the functionality of each third-party app. We did this because the underlying assumption of the study (and the design of PUMICE) is that users are familiar with the third-party apps, so we are testing whether they can successfully use PUMICE, not the apps. Then, the participant received 4 tasks in random order. We asked participants to keep trying until they were able to correctly execute the automation, and that they were happy with the resulting actions of the agent. We also checked the scripts at the end of each study session to evaluate their correctness.

After completing the tasks, the participant filled out a post-survey to report the perceived usefulness, ease of use and naturalness of interactions with PUMICE. We ended each session with a short informal interview with the participant on their experiences with PUMICE.

Tasks

We assigned 4 tasks to each participant. These tasks were designed by combining common themes observed in users' proposed scenarios from the formative study. We ensured that these tasks (1) covered key PUMICE features (i.e., concept learning, value query demonstration, procedure demonstration, concept generalization, procedure generalization and "else" condition handling); (2) involved only app interfaces that most users are familiar with; and (3) used conditions that we can control so we can test the correctness of the scripts (we controlled the temperature, the traffic condition, and the room price by manipulating the GPS location of the phone).

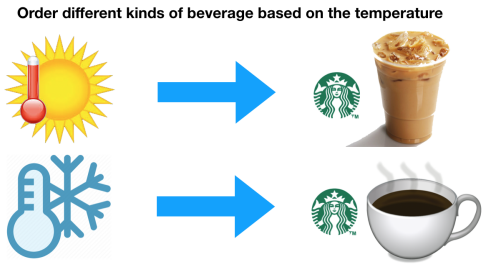

Figure 4. The graphical prompt used for Task 1 – A possible user command can be "*Order Iced coffee when it's hot outside, otherwise order hot coffee when the weather is cold.*"

In order to minimize biasing users' utterances, we used the Natural Programming Elicitation method [36]. Task descriptions were provided in the form of graphics, with minimal text descriptions that could not be directly used in user instructions (see Figure 4 for an example).

Task 1

In this task, the user instructs PUMICE to order iced coffee when the weather is hot, and order hot coffee otherwise (Figure 4). We pre-taught PUMICE the concept of "hot" in the task domain of turning on the air conditioner. So the user needs to utilize the concept generalization feature to generalize the existing concept "hot" to the new domain of coffee ordering. The user also needs to demonstrate ordering iced coffee using the Starbucks app, and to provide "order hot coffee" as the alternative for the "else" operation. The user does not need to demonstrate again for ordering hot coffee, as it can be automatically generalized from the previous demonstration of ordering iced coffee.

Task 2

In this task, the user instructs PUMICE to set an alarm for 7:00am if the traffic is heavy on their commuting route. We pre-stored "home" and "work" locations in Google Maps. The user needs to define "heavy traffic" as prompted by PUMICE by demonstrating how to find out the estimated commute time, and explaining that "heavy traffic" means that the commute takes more than 30 minutes. The user then needs to demonstrate setting a 7:00am alarm using the built-in Clock app.

Task 3

In this task, the user instructs PUMICE to choose between making a hotel reservation and requesting a Uber to go home depending on whether the hotel price is cheap. The user should verbally define "cheap" as "room price is below $100", and demonstrate how to find out the hotel price using the Marriott (a hotel chain) app. The user also needs to demonstrate making the hotel reservation using the Marriott app, specify "request an Uber" as the action for the "else" condition, and demonstrate how to request an Uber using the Uber app.

Task 4

In this task, the user instructs PUMICE to order a pepperoni pizza if there is enough money left in the food budget. The user needs to define the concept of "enough budget", demonstrate finding out the balance of the budget using the Spending Tracker app, and demonstrate ordering a pepperoni pizza using the Papa Johns (a pizza chain) app.

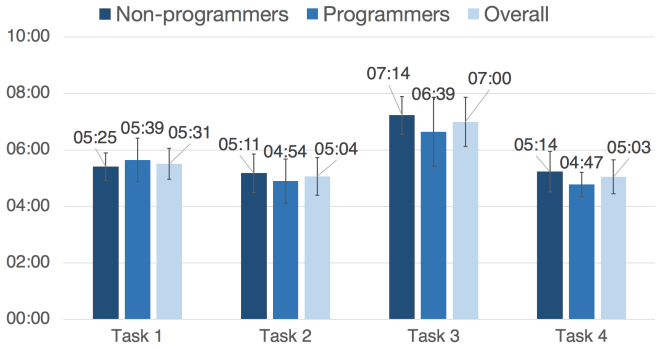

Figure 5. The average task completion times for each task. The error bars show one standard deviation in each direction.

Results

All participants were able to complete all 4 tasks. The total time for tasks ranged from 19.4 minutes to 25 minutes for the 10 participants. Figure 5 shows the overall average task completion time of each task, as well as the comparison between the non-programmers and the programmers. The average total time-on-task for programmers (22.12 minutes, SD=2.40) was slightly shorter than that for non-programmers (23.06 minutes, SD=1.57), but the difference was not statistically significant.

Most of the issues encountered by participants were actually from the Google Cloud speech recognition system used in PUMICE. It would sometimes misrecognize the user's voice input, or cut off the user early. These errors were handled by the participants using the "undo" feature in PUMICE. Some participants also had parsing errors. PUMICE's current semantic parser has limited capabilities in understanding references of pronouns (e.g., for an utterance *"it takes longer than 30 minutes to get to work"*, the parser would recognize it as "it" instead of "the time it takes to get to work" is greater than 30 minutes). Those errors were also handled by participants through undoing and rephrasing. One participant ran into the "confusion of Boolean operator" problem in Task 2 when she used the phrase "*commute [time is] worse than 30 minutes*", for which the parser initially recognized incorrectly as "commute is *less than* 30 minutes." She was able to correct this using the mixed-initiative mechanism, as described in the Error Recovery and Backtracking section.

Overall, no participant had major problem with the multimodal interaction approach and the top-down recursive concept resolution conversational structure, which was encouraging. However, all participants had received a tutorial with an example task demonstrated. We also emphasized in the tutorial that they should try to use concepts that can be found in mobile apps in their explanations of new concepts. These factors might contributed to the successes of our participants.

In a post survey, we asked participants to rate statements about the usability, naturalness and usefulness of PUMICE on a 7-point Likert scale from "strongly disagree" to "strongly agree". PUMICE scored on average 6.2 on "*I feel* PUMICE *is easy to use*", 6.1 on "*I find my interactions with* PUMICE *natural*", and 6.9 on "*I think* PUMICE *is a useful tool for automating tasks on smartphones,*" indicating that our participants were generally satisfied with their experience using PUMICE.

Discussion

In the informal interview after completing the tasks, participants praised PUMICE for its naturalness and low learning barriers. Non-programmers were particularly impressed by the multi-modal interface. For example, P7 (who was a non-programmer) said: "*Teaching* PUMICE *feels similar to explaining tasks to another person...[Pumice's] support for demonstration is very easy to use since I'm already familiar with how to use those apps.*" Participants also considered PUMICE's top-down interactive concept resolution approach very useful, as it does not require them to define everything clearly upfront.

Participants were excited about the usefulness of PUMICE. P6 said, "*I have an Alexa assistant at home, but I only use them for tasks like playing music and setting alarms. I tried some more complicated commands before, but they all failed. If it had the capability of* PUMICE*, I would definitely use it to teach Alexa more tasks.*" They also proposed many usage scenarios based on their own needs in the interview, such as paying off credit card balance early when it has reached a certain amount, automatically closing background apps when the available phone memory is low, monitoring promotions for gifts saved in the wish list when approaching anniversaries, and setting reminders for events in mobile games.

Several concerns were also raised by our participants. P4 commented that PUMICE should "just know" how to find out weather conditions without requiring her to teach it since "all other bots know how to do it", indicating the need for a hybrid approach that combines EUD with pre-programmed common functionalities. P5 said that teaching the agent could be too time-consuming unless the task was very repetitive since he could just "do it with 5 taps." Several users also expressed privacy concerns after learning that PUMICE can see all screen contents during demonstrations, while one user, on the other hand, suggested having PUMICE observe him at all times so that it can learn things in the background.

LIMITATIONS AND FUTURE WORK

The current version of PUMICE has no semantic understanding of information involved in tasks, which prevents it from dealing with implicit parameters (e.g., "when it snows" means "the current weather condition is *snowing*") and understanding the relations between concepts (e.g., Iced Cappuccino and Hot Latte are both instances of *coffee*; Iced Cappuccino has the property of being *cold*). The parser also does not process references, synonyms, antonyms, or implicit conjunctions/disjunctions in utterances. We plan to address these problems by leveraging more advanced NLP techniques. Specifically, we are currently investigating bringing in external sources of world knowledge (e.g., Wikipedia, Freebase [7], ConceptNet [29], WikiBrain [45], or NELL [35]), which can enable more intelligent generalizations, suggestions, and error detection. The agent can also make better guesses when dealing with ambiguous user inputs. As discussed previously, PUMICE already uses relational structures to store the context of app GUIs and its learned knowledge, which should make it easier to incorporate external knowledge graphs.

In the future, we plan to expand PUMICE's expressiveness in representing conditionals and Boolean expressions. In the current version, it only supports single basic Boolean operations (i.e., greater than, less than, equal to) without support for logical operations (e.g., when the weather is cold *and* raining) or arithmetic operations (e.g., if is at least *$10 more expensive* than Lyft), or counting GUI elements (e.g., "highly rated" means *more than 3 stars are red*) We plan to explore the design space of new interactive interfaces to support these features in future versions. Note that it will likely require more than just adding grammar rules to the semantic parser and expanding the DSL, since end users' usage of words like "and" and "or", and their language for specifying computation are known to often be ambiguous [41].

Further, although PUMICE supports generalization of procedures, Boolean concepts and value concepts across different task domains, all such generalizations are stored locally on the phone and limited to one user. We plan to expand PUMICE to support generalizing learned knowledge across multiple users. The current version of PUMICE does not differentiate between personal preferences and generalizable knowledge in learned concepts and procedures. An important focus of our future work is to distinguish these, and allow the sharing and aggregation of generalizable components across multiple users. To support this, we will also need to develop appropriate mechanisms to help preserve user privacy.

In this prototype of PUMICE, the proposed technique is used in conversations for performing immediate tasks instead of for completely automated rules. We plan to add the support for automated rules in the future. An implementation challenge for supporting automated rules is to continuously poll values from GUIs. The current version of underlying SUGILITE framework can only support foreground execution, which is not feasible for background monitoring for triggers. We plan to use techniques like virtual machine (VM) to support background execution of demonstrated scripts.

Lastly, we plan to conduct an open-ended field study to better understand how users use PUMICE in real-life scenarios. Although the design of PUMICE was motivated from results of a formative study with users, and the usability of PUMICE has been supported by an in-lab user study, we hope to further understand what tasks users choose to program, how they switch between different input modalities, and how useful PUMICE is for users in realistic contexts.

CONCLUSION

We have presented PUMICE, an agent that can learn concepts and conditionals from conversational natural language instructions and demonstrations. Through PUMICE, we showcased the idea of using multi-modal interactions to support the learning of unknown, ambiguous or vague concepts in users' verbal commands, which were shown to be common in our formative study.

In PUMICE's approach, users can explain abstract concepts in task conditions using more concrete smaller concepts, and ground them by demonstrating with third-party mobile apps. More broadly, our work demonstrates how combining conversational interfaces and demonstrational interfaces can create easy-to-use and natural end user development experiences.

ACKNOWLEDGMENTS

This research was supported in part by Verizon and Oath through the InMind project, a J.P. Morgan Faculty Research Award, and NSF grant IIS-1814472. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors. We would like to thank our study participants for their help, our anonymous reviewers, and Michael Liu, Fanglin Chen, Haojian Jin, Brandon Canfield, Jingya Chen, and William Timkey for their insightful feedback.

REFERENCES

[1] James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. PLOW: A Collaborative Task Learning Agent. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2 (AAAI'07)*. AAAI Press, Vancouver, British Columbia, Canada, 1514–1519.

[2] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A Survey of Robot Learning from Demonstration. *Robot. Auton. Syst.* 57, 5 (May 2009), 469–483. DOI: http://dx.doi.org/10.1016/j.robot.2008.10.024

[3] Amos Azaria, Jayant Krishnamurthy, and Tom M. Mitchell. 2016. Instructable Intelligent Personal Agent. In *Proc. The 30th AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 4.

[4] Bruce W. Ballard and Alan W. Biermann. 1979. Programming in Natural Language "NLC" As a Prototype. In *Proceedings of the 1979 Annual Conference (ACM '79)*. ACM, New York, NY, USA, 228–237. DOI: http://dx.doi.org/10.1145/800177.810072

[5] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 1533–1544.

[6] Alan W. Biermann. 1983. Natural Language Programming. In *Computer Program Synthesis Methodologies (NATO Advanced Study Institutes Series)*, Alan W. Biermann and Gerard Guiho (Eds.). Springer Netherlands, 335–368.

[7] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1247–1250. http://dl.acm.org/citation.cfm?id=1376746

[8] 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 (UIST '18)*. ACM, New York, NY, USA, 963–975. DOI: http://dx.doi.org/10.1145/3242587.3242661

[9] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

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

[11] Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, and Michael S. Bernstein. 2018. Iris: A Conversational Agent for Complex Tasks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 473:1–473:12. DOI: http://dx.doi.org/10.1145/3173574.3174047

[12] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating Photo Manipulation Tutorials by Demonstration. In *ACM SIGGRAPH 2009 Papers (SIGGRAPH '09)*. ACM, New York, NY, USA, 66:1–66:9. DOI: http://dx.doi.org/10.1145/1576246.1531372

[13] T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (June 1996), 131–174. DOI: http://dx.doi.org/10.1006/jvlc.1996.0009

[14] H Paul Grice, Peter Cole, Jerry Morgan, and others. 1975. Logic and conversation. *1975* (1975), 41–58.

[15] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 2007. Programming by a Sample: Rapidly Creating Web Applications with D.Mix. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 241–250. DOI: http://dx.doi.org/10.1145/1294211.1294254

[16] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J. Brostow. 2019. HILC: Domain-Independent PbD System Via Computer Vision and Follow-Up Questions. *ACM Trans. Interact. Intell. Syst.* 9, 2-3, Article 16 (March 2019), 27 pages. DOI: http://dx.doi.org/10.1145/3234508

[17] Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to Transform Natural to Formal Languages. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3 (AAAI'05)*. AAAI Press, Pittsburgh, Pennsylvania, 1062–1068. http://dl.acm.org/citation.cfm?id=1619499.1619504

[18] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2001. Your Wish is My Command. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Learning Repetitive Text-editing Procedures with SMARTedit, 209–226. http://dl.acm.org/citation.cfm?id=369505.369519

[19] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 1719–1728. DOI: http://dx.doi.org/10.1145/1357054.1357323

[20] 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 (CHI '17)*. ACM, New York, NY, USA, 6038–6049. DOI: http://dx.doi.org/10.1145/3025453.3025483

[21] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Tom M. Mitchell, and Brad A. Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Verbal Instructions. In *Proceedings of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2018)*.

[22] Toby Jia-Jun Li, Yuanchun Li, Fanglin Chen, and Brad A. Myers. 2017. Programming IoT Devices by Demonstration Using Mobile Apps. In *End-User Development*, Simone Barbosa, Panos Markopoulos, Fabio Paterno, Simone Stumpf, and Stefano Valtolina (Eds.). Springer International Publishing, Cham, 3–17.

[23] Toby Jia-Jun Li and Oriana Riva. 2018. KITE: Building conversational bots from mobile apps. In *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2018)*. ACM.

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

[25] Henry Lieberman and Hugo Liu. 2006. Feasibility studies for programming in natural language. In *End User Development*. Springer, 459–473.

[26] Henry Lieberman, Hugo Liu, Push Singh, and Barbara Barry. 2004. Beating Common Sense into Interactive Applications. *AI Magazine* 25, 4 (Dec. 2004), 63–63. DOI:http://dx.doi.org/10.1609/aimag.v25i4.1785

[27] H. Lieberman and D. Maulsby. 1996. Instructible agents: Software that just keeps getting better. *IBM Systems Journal* 35, 3.4 (1996), 539–556. DOI: http://dx.doi.org/10.1147/sj.353.0539

[28] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. ACM, New York, NY, USA, 97–106. DOI: http://dx.doi.org/10.1145/1502650.1502667

[29] H. Liu and P. Singh. 2004. ConceptNet — A Practical Commonsense Reasoning Tool-Kit. *BT Technology Journal* 22, 4 (01 Oct 2004), 211–226. DOI: http://dx.doi.org/10.1023/B:BTTJ.0000047600.45421.6d

[30] Christopher J. MacLellan, Erik Harpstead, Robert P. Marinier III, and Kenneth R. Koedinger. 2018. A Framework for Natural Cognitive System Training Interactions. *Advances in Cognitive Systems* (2018).

[31] Pattie Maes. 1994. Agents That Reduce Work and Information Overload. *Commun. ACM* 37, 7 (July 1994), 30–40. DOI:http://dx.doi.org/10.1145/176789.176792

[32] Jennifer Mankoff, Gregory D Abowd, and Scott E Hudson. 2000. OOPS: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers & Graphics* 24, 6 (2000), 819–834.

[33] Christoper Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52Nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. DOI: http://dx.doi.org/10.3115/v1/P14-5010

[34] Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (Natural Language Processing) for NLP (Natural Language Programming). In *Computational Linguistics and Intelligent Text Processing (Lecture Notes in Computer Science)*, Alexander Gelbukh (Ed.). Springer Berlin Heidelberg, 319–330.

[35] Tom Mitchell, William Cohen, Estevam Hruschka, Partha Talukdar, Bo Yang, Justin Betteridge, Andrew Carlson, B Dalvi, Matt Gardner, Bryan Kisiel, and others. 2018. Never-ending learning. *Commun. ACM* 61, 5 (2018), 103–115.

[36] Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. DOI: http://dx.doi.org/10.1109/MC.2016.200

[37] Brad A. Myers, Andrew J. Ko, Chris Scaffidi, Stephen Oney, YoungSeok Yoon, Kerry Chang, Mary Beth Kery, and Toby Jia-Jun Li. 2017. Making End User Development More Natural. In *New Perspectives in End-User Development*. Springer, Cham, 1–22. DOI: http://dx.doi.org/10.1007/978-3-319-60291-2_1

[38] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52. DOI: http://dx.doi.org/10.1145/1015864.1015888

[39] Sharon Oviatt. 1999a. Mutual disambiguation of recognition errors in a multimodel architecture. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 576–583.

[40] Sharon Oviatt. 1999b. Ten Myths of Multimodal Interaction. *Commun. ACM* 42, 11 (Nov. 1999), 74–81. DOI:http://dx.doi.org/10.1145/319382.319398

[41] John F. Pane, Brad A. Myers, and others. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies* 54, 2 (2001), 237–264. http://www.sciencedirect.com/science/article/pii/S1071581900904105

[42] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*. http://arxiv.org/abs/1508.00305 arXiv: 1508.00305.

[43] Fabio Paterno and Volker Wulf. 2017. *New Perspectives in End-User Development* (1st ed.). Springer.

[44] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A Natural Language Interface for Programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces (IUI '00)*. ACM, New York, NY, USA, 207–211. DOI: http://dx.doi.org/10.1145/325737.325845

[45] Shilad Sen, Toby Jia-Jun Li, WikiBrain Team, and Brent Hecht. 2014. Wikibrain: democratizing computation on wikipedia. In *Proceedings of The International Symposium on Open Collaboration*. ACM, 27. http://dl.acm.org/citation.cfm?id=2641615

[46] Shashank Srivastava, Igor Labutov, and Tom Mitchell. 2017. Joint concept learning and semantic parsing from natural language explanations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 1527–1536.

[47] Anselm Strauss and Juliet M. Corbin. 1990. *Basics of qualitative research: Grounded theory procedures and techniques.* Sage Publications, Inc.

[48] David Vadas and James R Curran. 2005. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005*. 191–199.

[49] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (March 2006), 33–35. DOI: http://dx.doi.org/10.1145/1118178.1118215

[50] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology (UIST '09)*. ACM, New York, NY, USA, 183–192. DOI: http://dx.doi.org/10.1145/1622176.1622213

[51] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 476–486.

[52] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. *CoRR* abs/1704.01696 (2017). http://arxiv.org/abs/1704.01696

[53] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. 2017. Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6024–6037. DOI: http://dx.doi.org/10.1145/3025453.3025846