Composite robot learning, and teaching Terry to use a computer

Owen Gallagher
10 November 2019
CS Seminar

INDEX

I. ABSTRACT

I propose to create a personal assistant program, both to explore the integration of multiple robot learning concepts into one system, and to innovate in a field whose mainstream design and paradigms have remained largely unchanged in recent years. At the same time, research in robot learning has been highly active and has produced many conclusions and resources waiting to be used in new ways. My project will incorporate key concepts from instruction learning, demonstration learning, and reinforcement learning. The main novelty of this proposed assistant is the addition of a different communication channel: the operating system's graphical user interface.

II. BACKGROUND

A. The Current Paradigm

Today's state-of-the-art virtual assistants are well known, unlike most of the codes and acronyms of the software development world, by simple names. Siri (Apple), Alexa (Amazon), and Cortana (Microsoft) are a few such examples. Having no problem with this tradition, I've decided to name this one Terry[1]. However, apart from the naming convention I hope to depart from the constrictive, essentially command-line interface (only superficially enhanced by speech-text and text-speech conversion) that popular virtual assistants use. Of course, there are doubtless advantages to this simplication of the robot-user interface; it's typically used on mobile devices as a hands-free alternative to the smartphone's touch GUI, and resembles text messaging, a mode of interaction that most users are already comfortable using.

But there are also some clear and compelling advantages to using a GUI for the assistant. A virtual assistant generally accomplishes tasks that are already doable through the operating system's GUI. Instead of clicking the messaging app shortcut, typing the recipient's name as "Ava Adams", clicking the composer bar, typing the message body, and clicking the send button, the user can instead open the virtual assistant and say, "Tell Ava Adams that I'm going to pick up the card on my way to the station". However, in certain scenarios the user may think of a task that the assistant does not know how to do, like "draw a hand waving with six fingers and send it to Brandon Bishop", or "find a picture of a green egg and set it as my background". In this case, it would be unreasonable for the original programming team to predict that a user might give such an instruction, but not unreasonable for the user to want to be able to do it. Here is where robot learning enters the arena, namely learning via instruction, demonstration, and reinforcement.

[1] I was between multiple names: Terry (the virtual secreTERRY), Morgan (ready to MORGANize), Neo (lord of *The Matrix*), and Otto (the OTTOmater), to name a few personal favorites.

B. How Terry Implements Composite Robot Learning

There are a number of papers that educate on these methods in robot learning. Researchers Thomaz and Breazeal found that in reinforcement learning, the concept of gaze is very important in facilitating teaching (Thomaz and Breazeal 2008). Terry achieves a viable way for the user/teacher to direct gaze via the cursor. They also found that people adapt their teaching strategy as they develop a mental model for how the robot learns. The OS essentially provides a way for the user to share a familiar perspective with Terry, facilitating the user's understanding of how it learns. Lastly, they point out that reinforcement from the user can be classified into two groups: feedback/reward/penalty and guidance. The OS in tandem with a speech interface works in favor of this also, in that the user can offer guidance with the mouse, feedback with speech, and properly time both modes of reinforcement by watching Terry progress through the action.

The paper by Argall et al. pointed out challenges with embodiment mapping (between the demonstration and the robot action) and record mapping (between the demonstrator and the demonstration) prevalent in demonstration learning (Argall et al. 2008). Terry avoids to a large extent having to create these mappings because the user/teacher can demonstrate using the same peripherals (keyboard and cursor) that Terry will use to carry out the instructions.

Lauria et al. implemented a system for instruction learning, applied to a scenario where a user teaches a robot how to navigate a map resembling a suburban neighborhood (Lauria et al. 2002). Here the challenge was to take complex natural language instructions and break them down until arriving at their component instruction primitives, which are a set of initially known actions. I hope to use a very similar system for Terry, where instructions are given through speech. A key improvement that Terry will hopefully contribute to this system is the ability to learn to recognize new widgets, analogous to landmarks in the case of the paper, in addition to recognizing new primitive combinations.

III. OBJECTIVES

The personal objective of my research project is to explore methods of learning in robots. Initially, I planned to explore this either by comparing various learning methods (demonstration, instruction, reinforcement), or by combining features of multiple and focusing more on the use case. I have decided to do the latter.

In terms of general objectives, completion of even a minimally functioning Terry prototype would mean that the core innovation behind it (using the OS GUI interface and incorporating learning techniques) has a useable proof of concept. Such a prototype is particularly feasible with this project because it involves many aspects

that can be broken into multiple levels of completion, each level representing a subsequent prototype. For example, the user interface portion will at the fully complete stage incorporate multiple communication categories for: supervision, feedback, guidance, lessons/teaching, commands, queries, and logging. Each of these can be incrementally layered into the interface over time, and each in turn can also be broken into stages.

So, to extend on the UI example, such a "minimally functioning" prototype could rely solely on text-based communication of commands and logging messages, use the demonstration-based learning approach, execute exactly-worded commands from the user, and demonstrate basic control of the mouse and keyboard. The minimal prototype would connect all the modules together, and a fuller release would include full speech-to-text and text-to-speech support, handle all types of input and output, and all components related to effective learning: instruction derivation, widget definition, demonstration, and reinforcement.

IV. SPECIFICATIONS BREAKDOWN

Terry is OS-independent (probably implemented with Java) desktop program which acts like a trainable computer-user robot, or virtual assistant. It's essentially a program that runs as a customizable interface between the user and the operating system, both to simplify certain actions done on the workstation and to potentially automate them. The final deliverable is a working prototype with a set of observations derived from its successes and shortcomings. In the pursuit of more successes, I plan to incorporate at least two methods of correction for the implementation and design choices that I've made and will make:

1. Get samples of testers and collect their feedback via online forms for each release of Terry.
2. Create multiple alternative interface designs (at least two) for each subsequent release and include them in the feedback form that evaluates the previous one, prior to implementation of the next one.

A. User interface

1. Speech Input

Admittedly, distinction between feedback and guidance is somewhat cloudy. These come into play when dealing with reinforcement learning, which is more easily employed with flexible and less structured learners, as is the case with neural networks. To rectify feedback and guidance with demonstration and instruction learning, I plan to use feedback as a way to control simulated confidence in Terry's execution speed, and guidance to help resolve ambiguities. Negative feedback would

lower confidence/speed (and raise hesitation), giving the user more of a time window for guidance. In a situation of ambiguity, spoken guidance could clarify previously implied instruction words.

Commands and lessions will also be communicated via speech input. A command is a request for execution of a learned action, and a lesson provides definitions for unknown widgets included in commands, and for unknown composite actions.

2. Cursor Input

Cursor input, as in clicks, moves, and drags, are to be used for both guidance and lessons. In situations of hesitation/ambiguity, the cursor can be directed by the user, providing guidance. This could perhaps be applied to the keyboard as well, but so far how is not clear to me. For lessons, at least half of the demonstration of new actions will be done by directing the cursor. Also, at times a Terry query will open a visual prompt to which the user can respond by cursor interaction with its own widgets.

3. Keyboard Input

Keyboard is to be used for lessons, and I'll try to avoid using it as a general alternative to speech input. In terms of lessons, the other half (the first being cursor input) of new action demonstration will be done by typing with the keyboard.

4. Speech and Text Output

Terry performs minor narration of execution and internal state, which is logging. The other half of printed and spoken output will be used for querying, or asking for clarification/guidance, or a definition for an unknown action or widget.

5. Terry GUI

Terry's own GUI consists of auxiliary windows and widgets to accompany queries and text output.

6. Operating System GUI

The last part of the full user interface to consider is the operating system's GUI, which is both the view by which the user supervises as actions are undertaken, and the view that Terry uses to find and interact with widgets.

B. Natural Language Instruction Parsing

Below are some considerations and components involved in Terry's language processor.

1. Dictionary

Terry's dictionary is a collection of relevant vocabulary with grammar tagging (as in a label for the part of speech of a word: noun, verb, adjective, preposition, etc) and references (a list of action and lesson patterns[2] and widget names that include this word). See the pseudocode's data structures section for further explanation of actions, lessons and widgets.

2. Dependency Parser

The dependency parser essentially builds sentence diagrams for instructions' semantic meanings. The primary use of the semantic representation of an instruction in general natural language processing is to know what kind of information is expected to be found at each word's position in the instruction. However, since Terry would be looking for a set of patterns to match against the instruction, it should in theory already know the kind of word to expect at each position. Given this consideration, perhaps a dependency parser will not be needed.

3. Comprehension

To be able to understand, or comprehend what the instruction means, the required abilities can be thought of as the following pipeline of modules, through which an instruction is passed:

1. Speech-to-text: this would definitely come from a library written by someone else (probably built into the host OS), which converts speech audio to a string of likely word tokens.
2. Grammar tagger: does a dictionary lookup for the given token. Based on the grammar tag it can remove the token from the pipeline (many determinants like "the" or "a" are trivial and can be ignored) or flag it for the message classifier.
3. Message classifier: here is where the type of instruction is determined, if the context doesn't make it already obvious (as in, Terry sent a query requesting the next command and is therefore waiting for a command instruction). A key example is the difference between a lesson and a command. In the latter case, the message classifier is looking for an implied "you" subject, while in a lesson the subject should be the unknown action or widget ("[you] move the mouse" instead of "[~~the~~[3] exit button] is red"). Valid types of instructions are commands, action lessons, and widget lessons.
4. Pattern mapper: given the type of instruction, this module (or group of modules, rather; each instruction type has its own mapper) searches the

[2] A pattern is a group of words/tokens that follows a specific order, and that accounts for expected arguments in and around the group. The idea of a pattern here closely resembes a regular expression.
[3] Here I wanted to show that "the", being a trivial word, would have been thrown away by the grammar tagger.

corresponding corpus (actions or lessons) for expected tokens or token types, and then matches the tokens in the instruction to those in the expected corpus entry. Once the keywords and arguments are mapped, the corresponding code is pushed to the appropriate subsequent module, either for the driver to execute the action, for the widget learner to create a new widget object with associated characteristics, or for the action learner to define a new composite action from subactions.

C. Learning

The task of learning for the Terry program can be broken into two principle components: learning memory, or what needs to be learned/remembered and the datatypes that represent them, and learning acquisition, or the ways that Terry handles incoming lessons through demonstration, definition, derivation, and reinforcement.

1. Learning Memory

There are two principle things that Terry must learn to recognize: widgets and actions. Later on in the **VI. Forseen Issues** section I entertain the possibility of another type of learned thing, but for now I'll just consider these two.

Widgets are GUI elements that the user visually recognizes and can be interactive in varying ways (buttons, text inputs), or not interactive (labels, graphics). They are stored as a set of characteristics associated with the widget name:

- Type is one of the following (to be expanded later): button, textbox, radio, label, graphic.
- Label is identifying text within the widget, which is leveraged by visual character recognition.
- Examples is a set of screenshot images showing the widget on-screen. The images are then compressed into sets of visual characteristics[4].
- Bounds is a bounding rectangle defining the widget's area with respect to its location.

Actions can be primitive or composite, and are stored as a pattern associated with a functional mapping defining destination states. In **VI.B Reconciling Demonstration and Instruction Learning** I go into further detail regarding what these states mean for action learning. A primitive action is one whose state transitions do incorporate

[4] I'm still not certain myself what these "visual characteristics" will be, but this is referring to some compressed version of the images. See the Ideally this representation of the image would be robust against color shifts and scaling, which would require further research into existing compression and image modification detection libraries. Perhaps an autoencoder neural network could be used, both for compressing an image

those of any other action. A composite action's state transitions correspond to a sequence of subaction state transitions.

2. Learning Acquisition

Learning acquisition is broken into four lesson categories. The first is a demonstration lesson. This refers to when the user performs an action themselves and the manipulations of the cursor and keyboard are recorded. Further discussion of how to properly represent a demonstration is covered in **VI.B Reconciling Demonstration and Instruction Learning**.

The second is a definition, or widget lesson. These define new GUI elements to be recognized and interacted with according to a set of characteristics as mentioned in the previous section.

An action lesson uses derivation to learn new actions as compositions of subactions. At the root(s) of this action inheritace tree are primitive actions, whose state transitions have direct driver mappings, or code to manipulate the mouse and keyboard peripherals.

Reinforcement, according to my currect planned approach, is the weakest lesson type, which determines action confidence/speed and bias/priority for ambiguities (where the same input phrase maps to multiple corpus entries).

D. Pseudocode

1. Data Structures

dictionary:
[{
    token: string
    grammar_tag: something like nn (noun), vb (verb), aa (adj/adv), etc
    references: list of references to actions, widgets, or lessons
}]

actions:
[{
    pattern: regular expression as a list of keywords and arguments
    dest_states: a list of state name,value pairs for updating states
}]

widgets:
[{

name: phrase string (usually is just a single word)
characteristics: {
type: widget type, like one of {button|textbox|radio|label|graphic|check}
label: string for visual character-based search
examples: list of compressed screenshots[5] of this widget
bounds: a rectangle that defines the widget area with respect to its origin
}
}]

lessons:
[{
pattern: regular expression as a list of keywords and arguments
definition_mapping: code creates the newly learned action or widget
}]

states: hash_table<state_name,state_value>

transition_table: hash_table<state_name,drive_mapping>

current_instruction: array of tokens

execution_queue: ordered list of actions that Terry will execute once the current instruction is done processing

2. Execution Flow

I plan to develop Terry using Java, which is the language I'm most comfortable with and the one I expect to require the least hassle getting to work across different platforms, and which is also an object-oriented language. However, I've written some pseudocode to describe Terry's general flow of execution, without much regard to encapsulation of data with methods or to class names, just to have a better starting point for carrying out this research project and to establish the connection between the planned architecture and the actual code.

[5] Or perhaps instead of a list of screenshots there would be a single compressed representation derived from multiple. If two screenshots are captured of the same widget with two different appearances, the image compression algorithm should find similarities between them and use those similarities.

```
/*
 * Terry pseudocode
 * Owen Gallagher
 * 2019.12.15
 */

initialize {
      //dictionary,actions,widgets,lessons,conditions,transitions
      load_data_structures()

      welcome() {
            speaker.say(“Hello...")
            prompter.spawn_console()
            logger.write(“Hello...")
      }
      prompter.spawn_terry_button()
}

terry_button.on_click({
      scribe.record_instructions() {
            pipe(token_queue)
      }
})

instruction_classifier.read(token_queue) {
      token = this.next()
      if (trivials.contains(token)) {
            //ignore token
      }
      else {
            dict_entry = dictionary.get(token)

            resolved = resolve(dict_entry) {
                  //possibilities is a list of trees, where each tree
                  //represents an instruction, and each node is a token.
                  //child nodes are possible subsequent tokens.

                  advanced = false

                  foreach(pi in possibilities) {
                        if (pi.expects(dict_entry.token)) {
                              pi.append(dict_entry) {
                                    t = dict_entry.token
                                    for (r in dict_entry.references) {
                                          //advance instruction possibility
                                          //to next position
```

```
                                    add_child(new possibility(t,r))
                              }
                        }
                        advanced = true
                  }
                  else {
                        //no longer possible instruction
                        possibilities.remove(pi)
                  }
            }
            if (possibilities.length == 1) {
                  if (advanced) {
                        //we know which instruction this is, but
                        //we've not reached the end of it
                        return false
                  }
                  else {
                        //we know which instruction this is and the
                        //current token is not part of it; therefore
                        //this instruction is finished
                        return true
                  }
            }
            else if (advanced) {
                  //we still are not sure which instruction this is
                  return false
            }
            else {
                  //this token is not part of the instruction and
                  //we're still waiting for more instruction tokens
                  //handle unfinished known instruction
                  //or handle unknown instruction
            }
      }

      if (resolved) {
            //create token sequence (instruction) with associated
            //type and reference
            pass_to_corresponding_mapper(possibilities.sequence())

            //start a new instruction
            possibilities.reset(token)
      }
  }
}
```

```
action_mapper.on_instruction(instruction) {
      switch(instruction.type) {
      case command:
            action = actions.get(instruction.reference)
            for (t in instruction.tokens) {
                  action.advance(t) {
                        //create state transitions from the given
                        //action command
                        if (expecting_arg(t)) {
                              dest_states.next().set(t)
                        }
                  }
            }
            execution_queue.add(action)
            break

      case lesson:
            lesson = lessons.get(instruction.reference)
            for (t in instruction.tokens) {
                  lesson.advance(t)
            }
            actions.add(lesson)
            break
      }
}
//also handle widget buttons, condition commands, condition lessons

driver.compiler.read(execution_queue) {
      action = this.next()

      for (s in action.states) {
            transitions.get(s).execute() {
                  //driver.pointer, or
                  //driver.typer, or
                  //driver.controller, or
                  //driver.listener
            }
      }
}
```

V. EXISTING RESOURCES

There are a number of existing resources to aid in robot learning development that I could investigate before attempting to implement everything from scratch. Below is a preliminary list:

A. Tell Me Dave Corpus

This is an annotated corpus of example natural language domestic task instructions to a robot, accompanied by a paper explaining how the research team used the instructions. It also discusses the general challenges of natural language commands, like ambiguity, anaphoric reference ("it", "them"), and omitted details. The instructions are specific to an environment that doesn't resemble mine and the scope of this research goes beyond mine in terms of probabilistic language parsing, so I won't be able to use much directly from this project. However, the instruction set is a good model for creating the primitive actions for Terry, and the paper's analysis of common problems in natural language parsing helps guide my implementation of Terry's language processor.

B. DeepText

DeepText is a Facebook project that seeks to build a model to understand natural language in the most general sense, without regard to instruction domain or context.

C. SLING

This is a system implemented by a group within Google's AI development teams that does both grammar tagging and dependency parsing in one pass, using frames in the form frame[arg0,arg1]. A neural network was trained to determine correct word-frame pairings without an intermediate grammar-tagged representation. To integrate SLING I would need to either train a new neural network for work-frame entries, or use their existing network. The latter is more feasible, but would likely lack the ability to parse commands. An alternative would be to use this approach in a more general sense for my own custom implementation that doesn't use a neural net. It's worth noting that this would not eliminiate the need for a corpus. As of now I'm more inclined not to incorporate this program into Terry, but it's certainly a case to learn from.

D. Autoencoders

I have not found a specific one yet, but I'm aware that there are existing autoencoder neural networks leveraged to do image compression (the vector that represents the smallest network layer is stored as a compressed version of the input image). This is one option for implementing image compression for widget images.

E. Visual Copyright Detection

Another option for visual widget recognition would be an algorithm used to detect image copyright infringement, which looks for distinctive similarities between images. I could leverage such a program to find similarities between a section of the screen and a widget example.

VI. FORSEEN ISSUES

A. Functional Actions

There is a distinction to be considered between fixed actions and functional actions. There may also be room for more kinds (see **VI.C. Conditional Actions** for another kind I thought of later) depending on what unpredictable commands a use might issue that introduce new complexities. But to only consider fixed and functional for now, here's the difference:

A fixed action is only a set of primitives or a composition/sequence of subactions. An example of a fixed action would look like this one: "trace a circle", which has a sequence of two subactions: press the mouse button, and then a path of moves following a circular trajectory.

A functional action includes arguments as a function would. "Look up water's boiling point in fahrenheit" matches a command pattern like look_up(query), where the query is an argument and maps to "water's boiling point in fahrenheit".

A solution to the problem of handling functional actions could be to include action arguments in primitives. In other words, create functional primitives. In the above case the primitive action would be something like type(string).

B. Reconciling Demonstration and Instruction Learning

Actions can be both demonstrated and described in language as a lesson. The problem with this becomes that an actions demonstration, which is a recording of cursor and keyboard manipulations, needs to be compatible with an action's description, which is a list of subactions. A way to create a sort of mapping between these two action teaching methods is to define primitives not simply as direct code to control the mouse and keyboard, but rather as transitions between states.

Such state transitions defined in corpus primitives would also need to be recorded during action demonstrations, and could perhaps replace the need to represent actions as raw peripheral manipulations. So this poses the question of how to properly represent a demonstration in learning memory as well. My current answer is to create

a demonstration mapper module, which finds corresponding state transitions for given mouse and keyboard inputs. This requires a defined bidirectional relationship between driver methods and state transitions. Below are some examples to illustrate:

- mouse_to(location) ⟷ state.set("mouse_at", location)
- mouse_to(widget) ⟷ state.set("mouse_at", widget)
- click(widget) ⟷ state.set("selected", widget)
- type(string) ⟷ state.set("typed", string)

C. Conditional Actions

In addition to fixed and functional actions I would also aim to have Terry support conditional actions, which is not as simple, but has high potential for increasing Terry's utility for users. A conditional action allows for control structures akin to selection, repetition, and event handling. Here, an action may be required to have the equivalent of a return value, which I plan to incorporate into the existing states paradigm. For conditional actions, states can be used as outputs for one action and arguments for another.

A difficulty with this kind of action is that it can exist in at least two fairly different forms, as in control structures versus event handling. I can imagine using one or the other paradigm to handle both cases. In a purely event-driven approach, selection creates a listener for the condition to become true, executes the associated action once, and dies on condition-is-true. Repetition creates a listener for the condition become false, executes the associated action continuously, and then dies on condition-is-false. Event handling creates a perpetual listener for a condition, which executes the associated action every time the condition is true. In a purely control-process-output (as in not event-driven) approach, selection uses an if statement to execute the associated action (or if-else in the case of actions for different outcomes). Repetition uses a while loop to execute the associated action until the condition is false. Event handling uses an infinite loop with selected if to execute an action whenever the condition is true, and have a certain delay between iterations.

So far I plan to use a mix between the two approaches, as in create an event listener that triggers an action for event-handling conditional actions, and use if-else and while statements for control structure ones. However, my decision was not based on any definitive reasoning, and may change because I see merits in using a single approach for all conditional actions. Examples of each are listed below:

- "<action> whenever I get a text" would be an event-handling conditional action. This is a condition I plan to carry out by creating an event listener for new text

messages that sets state got_text to true when a message is received, and an event handler that executes <action> when got_text is true. This kind of condition needs to recognize when a new text arrives, done by looking for notifications in the GUI, much in the same way Terry would look for a widget. See section **VI.F. Conditions** for more information about handling condition types.

- "text mary hello five times" → "<action> five times" is a control structure action. This could be done by creating a while loop with a counter variable that increments from 0 to 5. The loop body executes the text(message,recipient) action.

D. Argument Tokens

Sometimes a token is not going to have any dictionary references to actions, lessons, or widgets because it is an argument. A number is an easy example, but there's also the case of the string literal. Consider these two commands: "say close the window" and "close the window". The first asks Terry to output the string "close the window", while the second asks Terry to close the current application window. Apart from the difficulty for Terry's language processor to distinguish where a string literal begins and ends in spoken language, this situation at a basic level can be handled by recognized preceding action tokens that expect arguments of certain types ("say" expects a string literal, "close" expects a widget).

In terms of the issue I mentioned for delimiting a string literal, consider now this command: "say hello world and close the window". The command is spoken, so punctuation is not within the expected token set. Should Terry say "hello world" and close the window, or say "hello world and close the window"? A human would know that the former is much more likely because of the semantic meaning behind each string literal; "hello world" makes much more sense than "hello world and close the window". But Terry would not be able to know this because it's learning memory does not support such semantics. Therefore, my proposed solution is to support clarifying punctuation where desired, as in "say hello world end-quote and close the window", or "say begin-quote hello world end-quote and close the window". In fact, most speech-to-text libraries would support this natively, and would be able to pass the quotation character through to Terry's natural language processor.

E. Execution Queue

In some cases a single instruction could contain multiple actions and conditions. In the simple case where actions are strung together sequentially, the instruction does the sequencing with conjunctions ("and", "then", "but first"), but complexity can go beyond this. To deal with multiple actions and conditions in an instruction, the execution queue is short-term memory can compile the full execution sequence of

state transitions from the corpus mappers, and then upon reaching the end of the instruction convert the state transitions to driver mappings for Terry to carry out those actions.

F. Conditions

The diversity of types of conditions necessitates that they have their own data structure, as a sort of hybrid between a widget and an action. I did not include this in **D.1. Data Structures** because I do not intend to support them in my research project unless I have extra time after implementing everything else. The conditions structure described by the following definition:

conditions: [{
    pattern: regular expression as a list of keywords and arguments
    expression: a constructed boolean expression that determines whether the condition is true or false
}]

I say this data structure is like a hybrid between widgets and actions because, like a widget, a condition will require searching for something visually in the OS GUI, and like an action, the condition includes a pattern member, and its expression, via logical operators, could be a composition of multiple subconditions.

Implementing the condition's expression member may be a challenge. The expression for "I get a text" is that a notification widget appears (for example, if a message slides in from the right side of the screen labeled "New Message…"). It's not clear to me how to enable Terry to learn new conditions, but a primitive condition to handle this case would be appeared(widget). The expression for "five times" (as in "do an action five times") is that a given action's counter changes and is less than six. This expression is different from that of the previous example because the first requires a listener to then activate a conditional action, while the second requires only the conditional action itself; the first creates a state that could potentially be used by multiple conditional actions, but the state in the second scenario would realistically only be used within the context of that single conditional action.

Therefore, I plan only to use conditions for event-driven conditional actions. For control structure conditional actions, internal selection and repetition statements and variables should suffice, without the need to create external states.

G. Vocabulary Lessons

Another type of lesson that could augment Terry's utility would be a synonym or vocabulary lesson, where a token is described with other tokens much as a new word

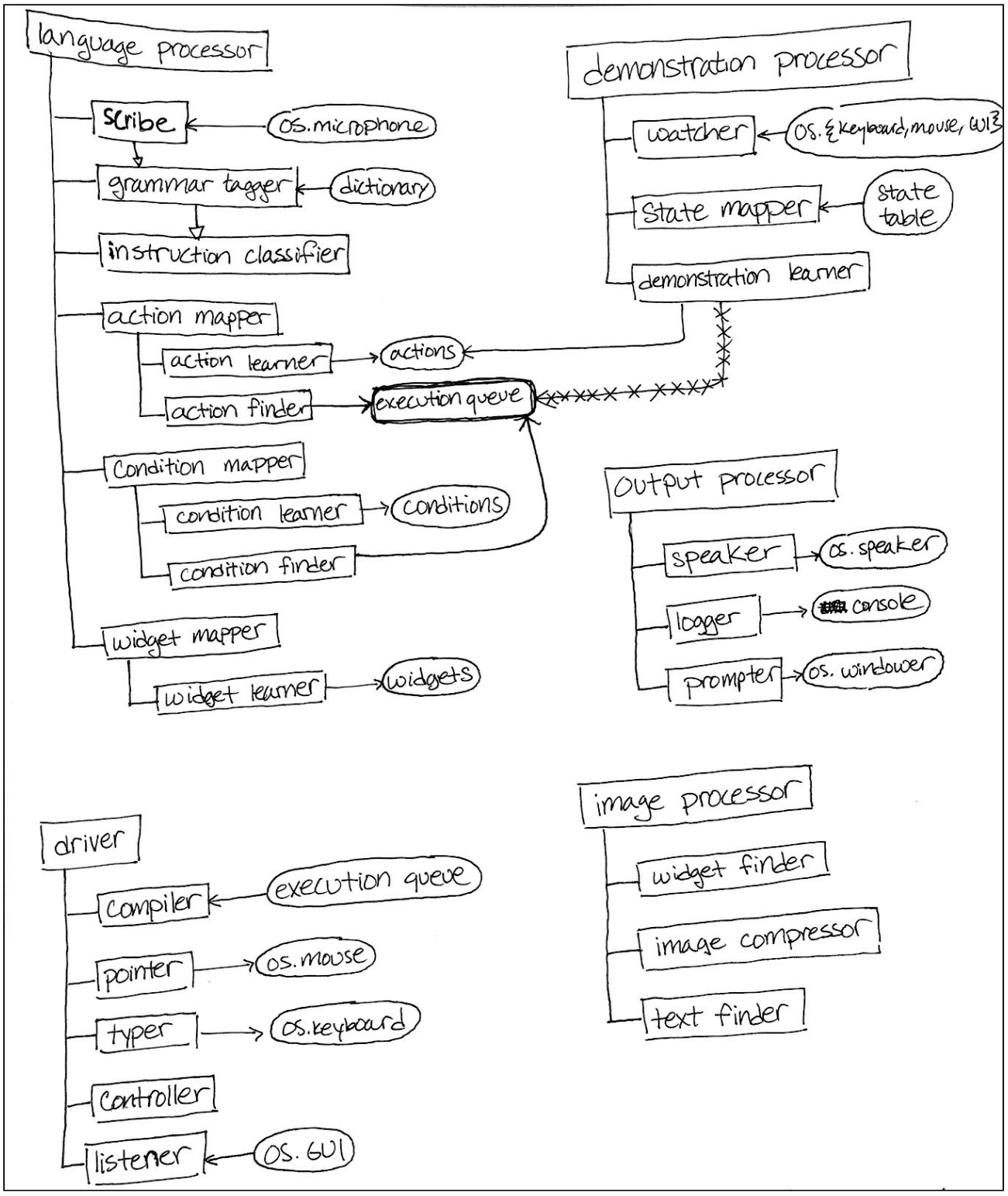
would be defined in a dictionary. However, I predict this could become very complicated and is not something I plan to support, at least initially.

H. Paths

Take this example of an action: "draw a hand". This is a functional action where the argument is "hand". The subactions would include, among others that "draw" requires open the designated drawing app, something like trace(path). This is not something my current data structures support. Widgets are the only supported argument type, which correspond to objects to be found in the GUI. Paths are a different concept because they refer, much like the idea of a path in the Scalable Vector Graphics (SVG) file type, to a line or curve described by control points. This curve defines a trajectory along which the cursor moves by interpolating discrete cursor locations along its length.

In addition to special actions like the one described above, paths could also be used for simpler primitives, like move_cusor(location). Instead of just specifying the location an alternative method signature would be move_cursor(path), where the path could be as trivial as a straight line from the cursor's current location to the destination location, but also as complex as a squiggle, a polygon, a spiral, or, as in the scenario above, a hand.

VII. ARCHITECTURE

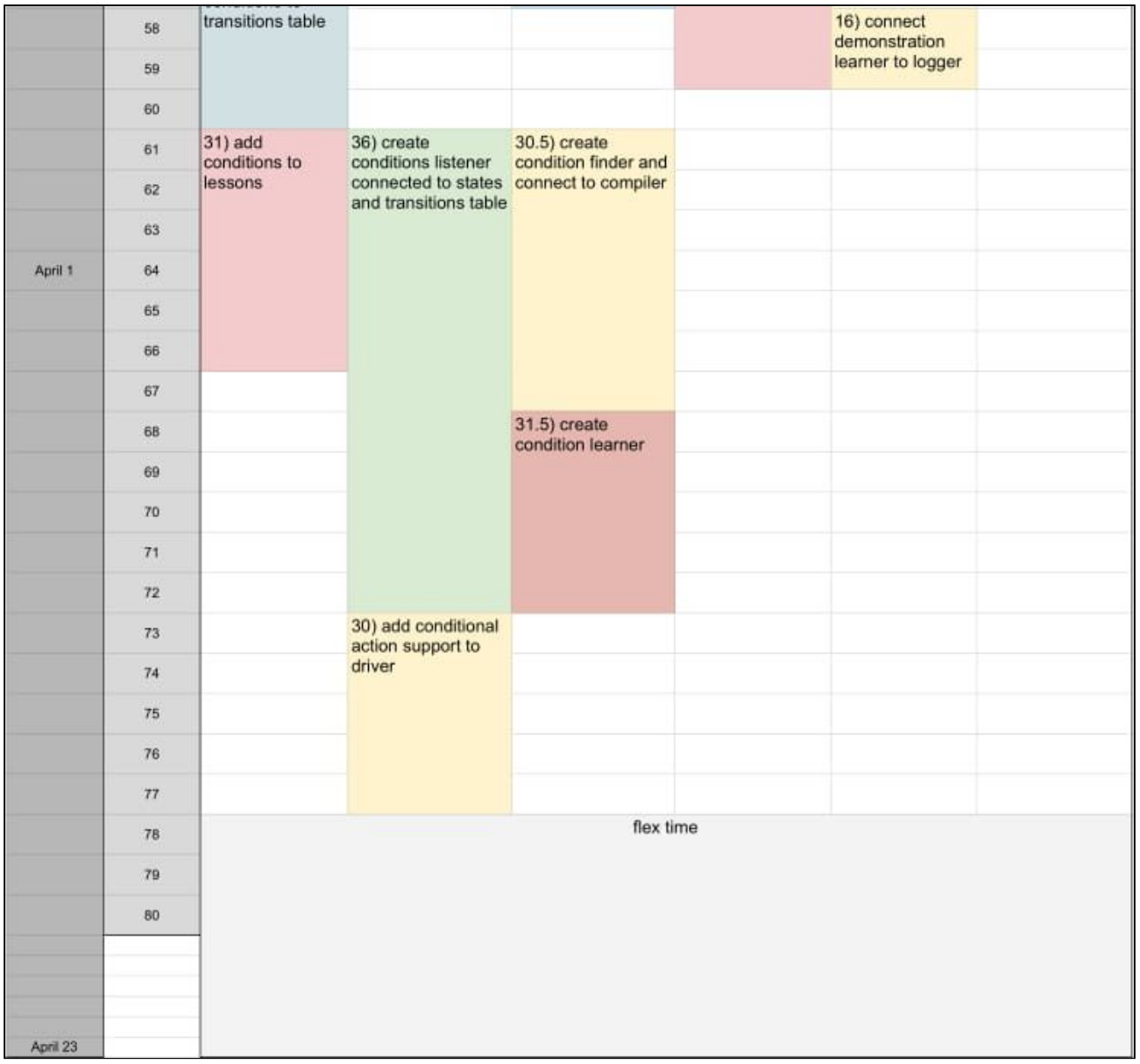


VIII. TIMELINE

This is a compressed preview of my timeline. I created it as a Google Sheet here: https://docs.google.com/spreadsheets/d/1glXGNizhLQWzd2gLkU9p1N5i0znVTbEfotKJe1nwoOY/edit?usp=sharing.

	day col						
work item row		data structures	driver	language processor	demonstration processor	output processor	image processor
January 21	1	1) create dictionary	5) connect pointer to mouse and typer to keyboard, and demonstrate baby driver	1.5) create scribe from exisiting OS speech-to-text programs	3) create watcher connected to the keyboard and mouse; prep data for primitive mapper	4) create logger and console window	19) create text finder; search for a character recognition lib
	2						
	3						
	4	6) create transitions table and state variable					
	5						
	6					8) create speaker using existing OS text-to-speech programs	
	7						
	8						
	9					23) create prompter to open dialog windows	
	10						
	11	7) create grammar tagger and connect to dictionary		9) create action finder to find associated actions given a command			20) create image compressor; find library and decide on output type
February 1	12						
	13						
	14					24) extend prompter to handle responses	
	15						
	16						
	17					15) connect logger to grammar tagger	
	18	12) create actions with initial primitives		2) basic instruction classifier: command vs lesson			
	19						
	20						
	21					25) connect image drop prompt with image compressor	
	22	13) create primitive action state dests and add entries to transition table		10) connect action finder to execution queue to compiler			
	23						
	24						
	25	11) create lessons with primitives					21) create widget finder using label and examples
	26						
	27						
	28						

	29						
	30						
	31						
	32	35) create widgets			34) connect watcher to widget finder		
	33						
	34		22) demonstrate the controller with some primitive actions, like "shut down" and "show state"				
	35						
	36	14) create action learner connected to lessons and dictionary		32) create widget learner, attach to lessons and widgets			
	37				17) create state mapper; connect to watcher and actions to ouput dest states		
	38						
	39						
March 1	40						
	41						
	42						
	43						
	44						
	45			33) connect widget learner to prompter			
March 7	46						
March 15	47						
	48						
	49			26) upgrade action learner to support multiple patterns for the same action			
	50	28) create conditions with primitives					
	51				18) create demonstration learner connected to state mapper and actions		
	52						
	53						
	54			27) upgrade action learner to update existing actions			
	55						
	56						
	57	29) add primitive conditions to					


transitions table
16) connect demonstration learner to logger
31) add conditions to lessons
36) create conditions listener connected to states and transitions table
30.5) create condition finder and connect to compiler
31.5) create condition learner
30) add conditional action support to driver
flex time
April 1
April 23

IX. BIBLIOGRAPHY

Abdulkader, Lakshmiratan, Zhang (2016) "Introducing DeepText: Facebook's text understanding engine". *Facebook: Engineering*. engineering.fb.com/ml-applications /introducing-deeptext-facebook-s-text-understanding-engine

> This article explains in accessible language how a group within Facebook's AI team worked on an engine for understanding user text content (written in natural language) for extracting things like facts, context, intent, and preference. The engine uses a convolutional neural network, and transforms the input text with a word embedding that preserves position in meaning space, both for semantics within a language and across languages.

Argall, Chernova, Veloso, Browning (2008) "A survey of robot learning from demonstration". *Robotics and Autonomous systems*, vol. 57, pp. 469-483

> The overview of processes and methods involved in demonstration learning from this paper is very thorough. It touches on many important concepts that have helped me find the right vocabulary for explanation and inquiry, evaluate the effectiveness of demonstration learning, and identify and prioritize common challenges that I'll need to address when implementing it. It explains, for example, that demonstration learning is broken into two phases: example gathering, and policy deriving. It's from this paper that I first became aware of the challenge of correspondence, and the solutions of record mapping and embodiment mapping.

Barrett, Bronikowski, Yu, Siskind (2018) "Driving under the Influence (of Language)". *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 7

> This approach uses natural language route commands for navigation and spatial understanding. A sentence's meaning for this particular application of natural language control gives information about both the robot's path and the objects in a floorplan, which can be thought of as a mix of commands and assertions. A robot constructs a path through a floorplan by satisfying constraints over time (t0=right of cone, t1=behind stool, t3=between stool and chair) with a very probabilistic model. The outcome of this research is a method of "grounding natural language semantics in robotic driving", which can be applied, at least in theory, to my application since Terry will be driving the cursor, so to speak, in a spatial environment (the GUI).

Calinon (2018) "Learning from Demonstration (Programming by Demonstration)". Springer, Berlin, Heidelberg. doi.org/10.1007/978-3-642-41610-1

> This paper gives an overview of demonstration learning. It lists various techniques involved, different applications, and comparisons with other methods used in robot learning. A key classification within demonstration learning that I've seen in other papers as well is divided into observational and kinesthetic learning, and a key problem prevalent in demonstration learning is correspondence between the demonstrator and robot.

Chen, Mooney (2011) "Learning to Interpret Natural Language Navigation Instructions from Observations". *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pp. 859-865

> The researchers tackle the problem of following natural language instructions for navigation (without robot learning) using a virtual indoor environment with clear visual cues. The complexity remains fairly constrained though, interestingly, they require that the human navigators give the instruction set from memory, which introduces a seemingly unnecessary variable to muddy the results.

King (2015) "Top 8 virtual personal assistants". *Raconteur*, Technology/Artificial Intelligence for Business. raconteur.net/technology/top-8-virtual-personal-assistants

> Leo King gives a list of 8 exemplary virtual personal assistants from differing environments. When I wanted to compare Terry to existing mainstream assistant programs I primarily used this post, in addition to personal experience. This list supports my assertion that the unique idea that Terry brings to the table is its use of the OS GUI, the same that the user uses, for demonstration learning.

Lauria, Bugmann, Kyriacou, Klein (2002). "Mobile robot programming using natural language". *Robotics and Autonomous Systems*, vol. 38, pp. 171-181

> This paper was essentially my entry into the robot learning field, and was the one that I presented in class. The authors' research is in response to the question of how to design robots in a world where the applications breach environments with increasingly naive users, as is the case of domestic robots. They decide to use an instruction-based learning approach, where the robot learns (is programmed) to do new tasks by breaking complex natural language commands into their primitive pieces. The paper also has a useful amount of detail regarding implementation specifics.

Misra, Sung, Lee, Saxena (2016) "Tell me Dave: Context-sensitive grounding of natural language to manipulation instructions". *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 281–300

> This is the project within which the Tell Me Dave corpus was created. It will require more reading on my part to see how the corpus was used exactly, but so far it looks like the corpus is a collection of natural language domestic task instructions.

Ringaard, Gupta (2017) "SLING: A Natural Language Frame Semantic Parser". *Google AI Blog*. ai.googleblog.com/2017/11/sling-natural-language-frame-semantic.html

> This post proposes an alternative to the modular pipeline solution for language parsing. This typical Natural Language Processing (NLP) system does grammar tagging, followed by dependency parsing. SLING does both steps at once by creating context-dependent frames for words.

Saponaro, Jamone, Bernardino, Salvi (2017) "Interactive Robot Learning of Gestures, Language and Affordances". arxiv.org/pdf/1711.09055.pdf

> Here the researchers propose a learning model to unify input from word descriptions, affordance exploration, and human gestures. This scenario is relevant to my research because, apart from dealing with robot learning in general, it also seeks to combine multiple modes of input for learning.

Thomaz, Breazeal (2008) "Teachable robots: Understanding human teaching behavior to build more effective robot learners". *Artificial Intelligence*, vol. 172, pp. 716-737

> This paper closely investigated the relationship between the teacher and the learner and the different types of communication that pass between them. It makes a number of assertions that I have accepted as truths moving forward, which dictate which methods of human-to-robot teaching should work better. They state that people, in general, want to direct the robot's attention to guide exploration, have a positive reward bias, and adapt their teaching strategy as they develop a mental model for how the robot learns. To address these findings, the authors suggest ways for the teacher to control the robot's gaze, or area of focus for visual input, and to view what the robot is sensing.

Zamani, Magg, Weber, Wermter (2018) "Deep reinforcement learning using symbolic representation for performing spoken language instructions". *Journal of Behavioral Robotics*, vol. 9, pp. 358-373. doi.org/10.1515/pjbr-2018-0026

These authors, like Lauria et al., considered the domestic task domain as justification for their work. It also references the advantages of combining instruction learning with the reinforcement learning studied by Thomaz and Breazal. Here, they emphasize the need to compartmentalize the task of hybrid learning into distinct modules: intention detection (parse user’s natural language instruction), deriving environment state (environment model and context for instruction), and the reward function (handling positive and negative reinforcement).