Naturalizing a Programming Language via Interactive Learning

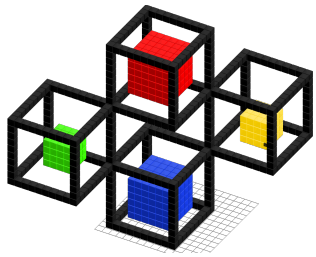
Sida I. Wang, Samuel Ginn, Percy Liang, Christopher D. Manning
Computer Science Department
Stanford University
{sidaw, samginn, pliang, manning}@cs.stanford.edu

Abstract

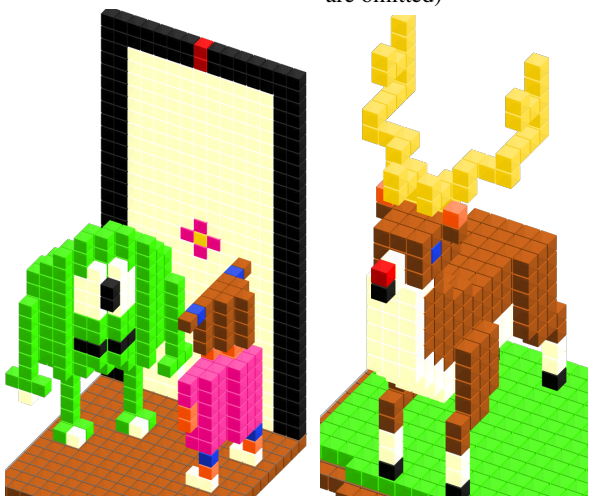
Our goal is to create a convenient natural language interface for performing well-specified but complex actions such as analyzing data, manipulating text, and querying databases. However, existing natural language interfaces for such tasks are quite primitive compared to the power one wields with a programming language. To bridge this gap, we start with a core programming language and allow users to "naturalize" the core language incrementally by defining alternative, more natural syntax and increasingly complex concepts in terms of compositions of simpler ones. In a voxel world, we show that a community of users can simultaneously teach a common system a diverse language and use it to build hundreds of complex voxel structures. Over the course of three days, these users went from using only the core language to using the naturalized language in 85.9% of the last 10K utterances.

1 Introduction

In tasks such as analyzing and plotting data (Gulwani and Marron, 2014), querying databases (Zelle and Mooney, 1996; Berant et al., 2013), manipulating text (Kushman and Barzilay, 2013), or controlling the Internet of Things (Campagna et al., 2017) and robots (Tellex et al., 2011), people need computers to perform well-specified but complex actions. To accomplish this, one route is to use a programming language, but this is inaccessible to most and can be tedious even for experts because the syntax is uncompromising and all statements have to be precise. Another route is to convert natural language into a formal language, which has been the subject of work in semantic parsing (Zettlemoyer and Collins, 2005; Artzi and Zettlemoyer, 2011, 2013; Pasupat and Liang, 2015). However, the capability of semantic parsers is still quite primitive compared to the power one wields with a programming language. This gap is increasingly limiting the potential of



Cubes: initial – select left 6 – select front 8 – black 10x10x10 frame – black 10x10x10 frame – move front 10 – red cube size 6 – move bot 2 – blue cube size 6 – green cube size 4 – (some steps are omitted)



Monsters, Inc: initial – move forward – add green monster – go down 8 – go right and front – add brown floor – add girl – go back and down – add door – add black column 30 – go up 9 – finish door – (some steps for moving are omitted)

Deer: initial – bird's eye view – deer head; up; left 2; back 2; { left antler }; right 2; {right antler} – down 4; front 2; left 3; deer body; down 6; {deer leg front}; back 7; {deer leg back}; left 4; {deer leg back}; front 7; {deer leg front} – (some steps omitted)

Figure 1: Some examples of users building structures using a naturalized language in Voxelurn: **http://www.voxelurn.com**

Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, pages 929–938
Vancouver, Canada, July 30 - August 4, 2017.
https://doi.org/10.18653/v1/P17-1086

both text and voice interfaces as they become more ubiquitous and desirable.

In this paper, we propose bridging this gap with an interactive language learning process which we call *naturalization*. Before any learning, we seed a system with a core programming language that is always available to the user. As users instruct the system to perform actions, they augment the language by *defining* new utterances — e.g., the user can explicitly tell the computer that 'X' means 'Y'. Through this process, users gradually and interactively teach the system to understand the language that they *want to use*, rather than the core language that they are forced to use initially. While the first users have to learn the core language, later users can make use of everything that is already taught. This process accommodates both users' preferences and the computer action space, where the final language is both interpretable by the computer and easier to produce by human users.

Compared to interactive language learning with weak denotational supervision (Wang et al., 2016), definitions are critical for learning complex actions (Figure 1). Definitions equate a novel utterance to a sequence of utterances that the system already understands. For example, 'go left 6 and go front' might be defined as 'repeat 6 [go left]; go front', which eventually can be traced back to the expression 'repeat 6 [select left of this]; select front of this' in the core language. Unlike function definitions in programming languages, the user writes concrete values rather than explicitly declaring arguments. The system automatically extracts arguments and learns to produce the correct generalizations. For this, we propose a grammar induction algorithm tailored to the learning from definitions setting. Compared to standard machine learning, say from demonstrations, definitions provide a much more powerful learning signal: the system is told directly that 'a 3 by 4 red square' is '3 red columns of height 4', and does not have to infer how to generalize from observing many structures of different sizes.

We implemented a system called Voxelurn, which is a command language interface for a voxel world initially equipped with a programming language supporting conditionals, loops, and variable scoping etc. We recruited 70 users from Amazon Mechanical Turk to build 230 voxel structures using our system. All users teach the system at once, and what is learned from one user can be used by another user. Thus a *community* of users evolves the language to becomes more efficient over time, in a distributed way, through interaction. We show that the user community defined many new utterances—short forms, alternative syntax, and also complex concepts such as 'add green monster, add yellow plate 3 x 3'. As the system learns, users increasingly prefer to use the naturalized language over the core language: 85.9% of the last 10K accepted utterances are in the naturalized language.

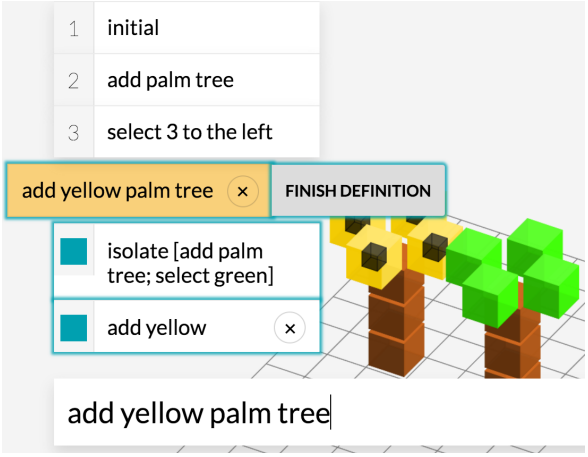


Figure 2: Interface used by users to enter utterances and create definitions.

2 Voxelurn

World. A world state in Voxelurn contains a set of voxels, where each voxel has relations 'row', 'col', 'height', and 'color'. There are two domain-specific actions, 'add' and 'move', one domain-specific relation 'direction'. In addition, the state contains a selection, which is a set of positions. While our focus is Voxelurn, we can think more generally about the world as a set of objects equiped with relations — events on a calendar, cells of a spreadsheet, or lines of text.

Core language. The system is born understanding a core language called Dependency-based Action Language (DAL), which we created (see Table 1 for an overview).

The language composes actions using the usual but expressive control primitives such as 'if', 'foreach', 'repeat', etc. Actions usually take sets as arguments, which are represented using lambda dependency-based compositional semantics (lambda DCS) expressions (Liang, 2013). Besides standard set operations like union, intersec-

Rule(s)	Example(s)	Description
$A \rightarrow A$; A	select left; add red	perform actions sequentially
$A \rightarrow$ repeat N A	repeat 3-1 add red top	repeat action N times
$A \rightarrow$ if S A	if has color red [select origin]	action if S is non-empty
$A \rightarrow$ while S A	while not has color red [select left of this]	action while S is non-empty
$A \rightarrow$ foreach S A	foreach this [remove has row row of this]	action for each item in S
$A \rightarrow [A]$	[select left or right; add red; add red top]	group actions for precedence
$A \rightarrow \{A\}$	{select left; add red}	scope only selection
$A \rightarrow$ isolate A	isolate [add red top; select has color red]	scope voxels and selection
$A \rightarrow$ select S	select all and not origin	set the selection
$A \rightarrow$ remove S	remove has color red	remove voxels
$A \rightarrow$ update R S	update color [color of left of this]	change property of selection
S	this	current selection
S	all | none | origin	all voxels, empty set, $(0, 0)$
R of S | has R S	has color red or yellow | has row [col of this]	lambda DCS joins
not S | S and S | S or S	this or left and not has color red	set operations
N | N+N | N-N	1,...,10 | 1+2 | row of this + 1	numbers and arithmetic
argmax R S | argmin R S	argmax col has color red	superlatives
R	color | row | col | height | top | left | $\cdots$	voxel relations
C	red | orange | green | blue | black | $\cdots$	color values
D	top | bot | front | back | left | right	direction values
$S \rightarrow$ very D of S	very top of very bot of has color green	syntax sugar for argmax
$A \rightarrow$ add C [D] | move D	add red | add yellow bot | move left	add voxel, move selection

Table 1: Grammar of the core language (DAL), which includes actions (A), relations (R), and sets of values (S). The grammar rules are grouped into four categories. From top to bottom: domain-general action compositions, actions using sets, lambda DCS expressions for sets, and domain-specific relations and actions.

tion and complement, lambda DCS leverages the tree dependency structure common in natural language: for the relation 'color', 'has color red' refers to the set of voxels that have color red, and its reverse 'color of has row 1' refers to the set of colors of voxels having row number 1. Tree-structured joins can be chained without using any variables, e.g., 'has color [yellow or color of has row 1]'.

We protect the core language from being redefined so it is always precise and usable.[1] In addition to expressivity, the core language *interpolates* well with natural language. We avoid explicit variables by using a *selection*, which serves as the default argument for most actions.[2] For example, 'select has color red; add yellow top; remove' adds yellow on top of red voxels and then removes the red voxels.

To enable the building of more complex structures in a more modular way, we introduce a notion of *scoping*. Suppose one is operating on one of the palm trees in Figure 2. The user might want to use 'select all' to select only the voxels in that tree rather than all of the voxels in the scene. In general, an action A can be viewed as taking a set of voxels v and a selection s, and producing an updated set of voxels v' and a modified selection s'. The default scoping is '[A]', which is the same as 'A' and returns (v', s'). There are two constructs that alter the flow: First, '{A}' takes (v, s) and returns (v', s), thus restoring the selection. This allows A to use the selection as a temporary variable without affecting the rest of the program. Second, 'isolate [A]' takes (v, s), calls A with (s, s) (restricting the set of voxels to just the selection) and returns (v'', s), where v'' consists of voxels in v' and voxels in v that occupy empty locations in v'. This allows A to focus only on the selection (e.g., one of the palm trees). Although scoping can be explicitly controlled via

[1] Not doing so resulted in ambiguities that propagated uncontrollably, e.g., once 'red' can mean many different colors.

[2] The selection is like the turtle in LOGO, but can be a set.

'[]', 'isolate', and '{}', it is an unnatural concept for non-programmers. Therefore when the choice is not explicit, the parser generates all three possible scoping interpretations, and the model learns which is intended based on the user, the rule, and potentially the context.

3 Learning interactively from definitions

The goal of the user is to build a structure in Voxelurn. In Wang et al. (2016), the user provided interactive supervision to the system by selecting from a list of candidates. This is practical when there are less than tens of candidates, but is completely infeasible for a complex action space such as Voxelurn. Roughly, 10 possible colors over the $3 \times 3 \times 4$ box containing the palm tree in Figure 2 yields 10^{36} distinct denotations, and many more programs. Obtaining the structures in Figure 1 by selecting candidates alone would be infeasible.

This work thus uses *definitions* in addition to selecting candidates as the supervision signal. Each definition consists of a *head* utterance and a *body*, which is a sequence of utterances that the system understands. One use of definitions is paraphrasing and defining alternative syntax, which helps naturalize the core language (e.g., defining 'add brown top 3 times' as 'repeat 3 add brown top'). The second use is building up complex concepts hierarchically. In Figure 2, 'add yellow palm tree' is defined as a sequence of steps for building the palm tree. Once the system understands an utterance, it can be used in the body of other definitions. For example, Figure 3 shows the full definition tree of 'add palm tree'. Unlike function definitions in a programming language, our definitions do not specify the exact arguments; the system has to learn to extract arguments to achieve the correct generalization.

The interactive definition process is described in Figure 4. When the user types an utterance x, the system parses x into a list of candidate programs. If the user selects one of them (based on its denotation), then the system executes the resulting program. If the utterance is unparsable or the user rejects all candidate programs, the user is asked to provide the definition body for x. Any utterances in the body not yet understood can be defined recursively. Alternatively, the user can first execute a sequence of commands X, and then provide a head utterance for body X.

When constructing the definition body, users can type utterances with multiple parses; e.g., 'move forward' could either modify the selection ('select front') or move the voxel ('move front'). Rather than propagating this ambiguity to the head, we force the user to commit to one interpretation by selecting a particular candidate. Note that we are using interactivity to control the exploding ambiguity.

```
def: add palm tree
    def: brown trunk height 3
        def: add brown top 3 times
            repeat 3 [add brown top]
    def: go to top of tree
        select very top of has color brown
    def: add leaves here
        def: select all sides
            select left or right or front or back
        add green
```

Figure 3: Defining 'add palm tree', tracing back to the core language (utterances without **def:**).

begin execute x:
  if x *does not parse* **then** define x;
  if *user rejects all parses* **then** define x;
  execute user choice

begin define x:
  repeat starting with $X \leftarrow [\,]$
    user enters x';
    if x' *does not parse* **then** define x';
    if *user rejects all* x' **then** define x';
    $X \leftarrow [X; x']$;
  until *user accepts* X *as the def'n of* x;

Figure 4: When the user enters an utterance, the system tries to parse and execute it, or requests that the user define it.

4 Model and learning

Let us turn to how the system learns and predicts. This section contains prerequisites before we describe definitions and grammar induction in Section 5.

Semantic parsing. Our system is based on a semantic parser that maps utterances x to programs z, which can be executed on the current state s (set of voxels and selection) to produce the next state $s' = [\![z]\!]_s$. Our system is implemented as the interactive package in SEMPRE (Berant et al., 2013);

Feature	Description
Rule.ID	ID of the rule
Rule.Type	core?, used?, used by others?
Social.Author	ID of author
Social.Friends	(ID of author, ID of user)
Social.Self	rule is authored by user?
Span	(left/right token(s), category)
Scope	type of scoping for each user

Table 2: Summary of features.

see Liang (2016) for a gentle exposition.

A *derivation* d represents the process by which an utterance x turns into a program $z = \text{prog}(d)$. More precisely, d is a tree where each node contains the corresponding span of the utterance $(\text{start}(d), \text{end}(d))$, the grammar rule $\text{rule}(d)$, the grammar category $\text{cat}(d)$, and a list of child derivations $[d_1, \ldots, d_n]$.

Following Zettlemoyer and Collins (2005), we define a log-linear model over derivations d given an utterance x produced by the user u:

$$p_\theta(d \mid x, u) \propto \exp(\theta^\mathsf{T} \phi(d, x, u)), \qquad (1)$$

where $\phi(d, x, u) \in \mathbb{R}^p$ is a feature vector and $\theta \in \mathbb{R}^p$ is a parameter vector. The user u does not appear in previous work on semantic parsing, but we use it to personalize the semantic parser trained on the community.

We use a standard chart parser to construct a chart. For each chart cell, indexed by the start and end indices of a span, we construct a list of partial derivations recursively by selecting child derivations from subspans and applying a grammar rule. The resulting derivations are sorted by model score and only the top K are kept. We use $\text{chart}(x)$ to denote the set of all partial derivations across all chart cells. The set of grammar rules starts with the set of rules for the core language (Table 1), but grows via grammar induction when users add definitions (Section 5). Rules in the grammar are stored in a trie based on the right-hand side to enable better scalability to a large number of rules.

Features. Derivations are scored using a weighted combination of features. There are three types of features, summarized in Table 2.

Rule features fire on each rule used to construct a derivation. ID features fire on specific rules (by ID). Type features track whether a rule is part of the core language or induced, whether it has been used again after it was defined, if it was used by someone other than its author, and if the user and the author are the same ($5 +$ #rules features).

Social features fire on properties of rules that capture the unique linguistic styles of different users and their interaction with each other. Author features capture the fact that some users provide better, and more generalizable definitions that tend to be accepted. Friends features are cross products of author ID and user ID, which captures whether rules from a particular author are systematically preferred or not by the current user, due to stylistic similarities or differences (#users+#users×#users features).

Span features include conjunctions of the category of the derivation and the leftmost/rightmost token on the border of the span. In addition, span features include conjunctions of the category of the derivation and the 1 or 2 adjacent tokens just outside of the left/right border of the span. These capture a weak form of context-dependence that is generally helpful ($<\approx V^4 \times$ #cats features for a vocabulary of size V).

Scoping features track how the community, as well as individual users, prefer each of the 3 scoping choices (none, selection only '{A}', and voxels+selection 'isolate {A}'), as described in Section 2. 3 global indicators, and 3 indicators for each user fire every time a particular scoping choice is made ($3 + 3 \times$ #users features).

Parameter estimation. When the user types an utterance, the system generates a list of candidate next states. When the user chooses a particular next state s' from this list, the system performs an online AdaGrad update (Duchi et al., 2010) on the parameters θ according to the gradient of the following loss function:

$$-\log \sum_{d:[\![\text{prog}(d)]\!]_s = s'} p_\theta(d \mid x, u) + \lambda ||\theta||_1,$$

which attempts to increase the model probability on derivations whose programs produce the next state s'.

5 Grammar induction

Recall that the main form of supervision is via user definitions, which allows creation of user-defined concepts. In this section, we show how to turn

these definitions into new grammar rules that can be used by the system to parse new utterances.

Previous systems of grammar induction for semantic parsing were given utterance-program pairs (x, z). Both the GENLEX (Zettlemoyer and Collins, 2005) and higher-order unification (Kwiatkowski et al., 2010) algorithms overgenerate rules that liberally associate parts of x with parts of z. Though some rules are immediately pruned, many spurious rules are undoubtedly still kept. In the interactive setting, we must keep the number of candidates small to avoid a bad user experience, which means a higher precision bar for new rules.

Fortunately, the structure of definitions makes the grammar induction task easier. Rather than being given an utterance-program (x, z) pair, we are given a definition, which consists of an utterance x (head) along with the body $X = [x_1, \ldots, x_n]$, which is a sequence of utterances. The body X is fully parsed into a derivation d, while the head x is likely only partially parsed. These partial derivations are denoted by $\text{chart}(x)$.

At a high-level, we find *matches*—partial derivations $\text{chart}(x)$ of the head x that also occur in the full derivation d of the body X. A grammar rule is produced by substituting any set of non-overlapping matches by their categories. As an example, suppose the user defines

'add red top times 3' as 'repeat 3 [add red top]'.

Then we would be able to induce the following two grammar rules:

$$A \rightarrow \text{add } C\ D \text{ times } N\ :$$
$$\lambda CDN.\text{repeat } N\ [\text{add } C\ D]$$
$$A \rightarrow A \text{ times } N\ :$$
$$\lambda AN.\text{repeat } N\ [A]$$

The first rule substitutes primitive values ('red', 'top', and '3') with their respective pre-terminal categories (C, D, N). The second rule contains compositional categories like actions (A), which require some care. One might expect that greedily substituting the largest matches or the match that covers the largest portion of the body would work, but the following example shows that this is not the case:

$$\overbrace{\underbrace{\text{add red left}}_{A_2}}^{A_1} \text{ and here} = \overbrace{\underbrace{\text{add red left;}}_{A_2}}^{A_1}\ \overbrace{\text{add red}}^{A_1}$$

Here, both the highest coverage substitution (A_1: 'add red', which covers 4 tokens of the body), and the largest substitution available (A_2: 'add red left') would generalize incorrectly. The correct grammar rule only substitutes the primitive values ('red', 'left').

5.1 Highest scoring abstractions

We now propose a grammar induction procedure that optimizes a more global objective and uses the learned semantic parsing model to choose substitutions. More formally, let M be the set of partial derivations in the head whose programs appear in the derivation d_X of the body X:

$$M \stackrel{\text{def}}{=} \{d \in \text{chart}(x) :$$
$$\exists d' \in \text{desc}(d_X) \wedge \text{prog}(d) = \text{prog}(d')\},$$

where $\text{desc}(d_X)$ are the descendant derivations of d_X. Our goal is to find a *packing* $P \subseteq M$, which is a set of derivations corresponding to non-overlapping spans of the head. We say that a packing P is maximal if no other derivations may be added to it without creating an overlap.

Let $\text{packings}(M)$ denote the set of maximal packings, we can frame our problem as finding the maximal packing that has the highest score under our current semantic parsing model:

$$P_L^* = \underset{P \in \text{packings}(M);}{\text{argmax}} \sum_{d \in P} \text{score}(d). \quad (2)$$

Finding the highest scoring packing can be done using dynamic programming on P_i^* for $i = 0, 1, \ldots, L$, where L is the length of x and $P_0^* = \emptyset$. Since $d \in M$, $\text{start}(d)$ and $\text{end}(d)$ (exclusive) refer to span in the head x. To obtain this dynamic program, let D_i be the highest scoring maximal packing containing a derivation ending *exactly* at position i (if it exists):

$$D_i = \{d_i\} \cup P^*_{\text{start}(d_i)}, \quad (3)$$
$$d_i = \underset{d \in M; \text{end}(d) = i}{\text{argmax}} \text{score}(d \cup P^*_{\text{start}(d)}). \quad (4)$$

Then the maximal packing of up to i can be defined recursively as

$$P_i^* = \underset{D \in \{D_{s(i)+1}, D_{s(i)+2}, \ldots, D_i\}}{\text{argmax}} \text{score}(D) \quad (5)$$
$$s(i) = \max_{d: \text{end}(d) \leq i} \text{start}(d), \quad (6)$$

Input : x, d_X, P^*
Output: rule
$r \leftarrow x$;
$f \leftarrow d_X$;
for $d \in P^*$ **do**
  $r \leftarrow r[\text{cat}(d)/\text{span}(d)]$
  $f \leftarrow \lambda\, \text{cat}(d).f[\text{cat}(d)/d]$
return *rule* $(\text{cat}(d_X) \rightarrow r : f)$

Algorithm 1: Extract a rule r from a derivation d_X of body X and a packing P^*. Here, $f[t/s]$ means substituting s by t in f, with the usual care about names of bound variables.

where $s(i)$ is the largest index such that $D_{s(i)}$ is no longer maximal for the span $(0, i)$ (i.e. there is a $d \in M$ on the span $\text{start}(d) \geq s(i) \wedge \text{end}(d) \leq i$.

Once we have a packing $P^* = P^*_L$, we can go through $d \in P^*$ in order of $\text{start}(d)$, as in Algorithm 1. This generates one high precision rule per packing per definition. In addition to the highest scoring packing, we also use a "simple packing", which includes only primitive values (in Voxelurn, these are colors, numbers, and directions). Unlike the simple packing, the rule induced from the highest scoring packing does not always generalize correctly. However, a rule that often generalizes incorrectly should be down-weighted, along with the score of its packings. As a result, a different rule might be induced next time, even with the same definition.

5.2 Extending the chart via alignment

Algorithm 1 yields high precision rules, but fails to generalize in some cases. Suppose that 'move up' is defined as 'move top', where 'up' does not parse, and does not match anything. We would like to infer that 'up' means 'top'. To handle this, we leverage a property of definitions that we have not used thus far: the utterances themselves. If we align the head and body, then we would intuitively expect aligned phrases to correspond to the same derivations. Under this assumption, we can then transplant these derivations from d_X to $\text{chart}(x)$ to create new matches. This is more constrained than the usual alignment problem (e.g., in machine translation) since we only need to consider spans of X which corresponds to derivations in $\text{desc}(d_X)$.

Algorithm 2 provides the algorithm for extending the chart via alignments. The aligned function is implemented using the following two heuristics:

Input : x, X, d_X
for $d \in \text{desc}(d_X)$, $x' \in \text{spans}(x)$ **do**
  if $\text{aligned}(x', d, (x, X))$ **then**
    $d' \leftarrow d$;
    $\text{start}(d') \leftarrow \text{start}(x')$;
    $\text{end}(d') \leftarrow \text{end}(x')$;
    $\text{chart}(x) \leftarrow \text{chart}(x) \cup d'$
  end
end

Algorithm 2: Extending the chart by alignment: If d is aligned with x' based on the utterance, then we pretend that x' should also parse to d, and d is transplanted to $\text{chart}(x)$ as if it parsed from x'.

- **exclusion**: if all but 1 pair of short spans (1 or 2 tokens) are matched, the unmatched pair is considered aligned.
- **projectivity**: if $d_1, d_2 \in \text{desc}(d_X) \cap \text{chart}(x)$, then $\text{ances}(d_1, d_2)$ is aligned to the corresponding span in x.

With the extended chart, we can run the algorithm from Section 5.1 to induce rules. The transplanted derivations (e.g., 'up') might now form new matches which allows the grammar induction to induce more generalizable rules. We only perform this extension when the body consists of one utterance, which tend to be a paraphrase. Bodies with multiple utterances tend to be new concepts (e.g., 'add green monster'), for which alignment is impossible. Because users have to select from candidates parses in the interactive setting, inducing low precision rules that generate many parses degrade the user experience. Therefore, we induce alignment-based rules conservatively—only when all but 1 or 2 tokens of the head aligns to the body and vice versa.

6 Experiments

Setup. Our ultimate goal is to create a community of users who can build interesting structures in Voxelurn while naturalizing the core language. We created this community using Amazon Mechanical Turk (AMT) in two stages. First, we had *qualifier* tasks, in which an AMT worker was instructed to replicate a fixed target exactly (Figure 5), ensuring that the initial users are familiar with at least some of the core language, which is the starting point of the naturalization process.

Figure 5: The target used for the qualifier.

Next, we allowed the workers who qualified to enter the second *freebuilding* task, in which they were asked to build any structure they wanted in 30 minutes. This process was designed to give users freedom while ensuring quality. The analogy of this scheme in a real system is that early users (or a small portion of expert users) have to make some learning investment, so the system can learn and become easier for other users.

Statistics. 70 workers passed the qualifier task, and 42 workers participated in the final freebuilding experiment. They built 230 structures. There were over 103,000 queries consisting of 5,388 distinct token types. Of these, 64,075 utterances were tried and 36,589 were accepted (so an action was performed). There were 2,495 definitions combining over 15,000 body utterances with 6.5 body utterances per head on average (96 max). From these definitions, 2,817 grammar rules were induced, compared to less than 100 core rules. Over all queries, there were 8.73 parses per utterance on average (starting from 1 for core).

Is naturalization happening? The answer is yes according to Figure 6, which plots the cummulative percentage of utterances that are core, induced, or unparsable. To rule out that more induced utterances are getting rejected, we consider only accepted utterances in the middle of Figure 6, which plots the percentage of induced rules among accepted utterances for the entire community, as well as for the 5 heaviest users. Since unparsable utterances cannot be accepted, accepted core (which is not shown) is the complement of accepted induced. At the conclusion of the experiment, 72.9% of all accepted utterances are induced—this becomes 85.9% if we only consider the final 10,000 accepted utterances.

Three modes of naturalization are outlined in Table 3. For very common operations, like moving the selection, people found ‘select left’ too verbose and shorterned this to `|, left, >, sel |`. One user preferred ‘go down and right’ instead of ‘select bot; select right’ in core and defined it as ‘go down; go right’. Definitions for high-level

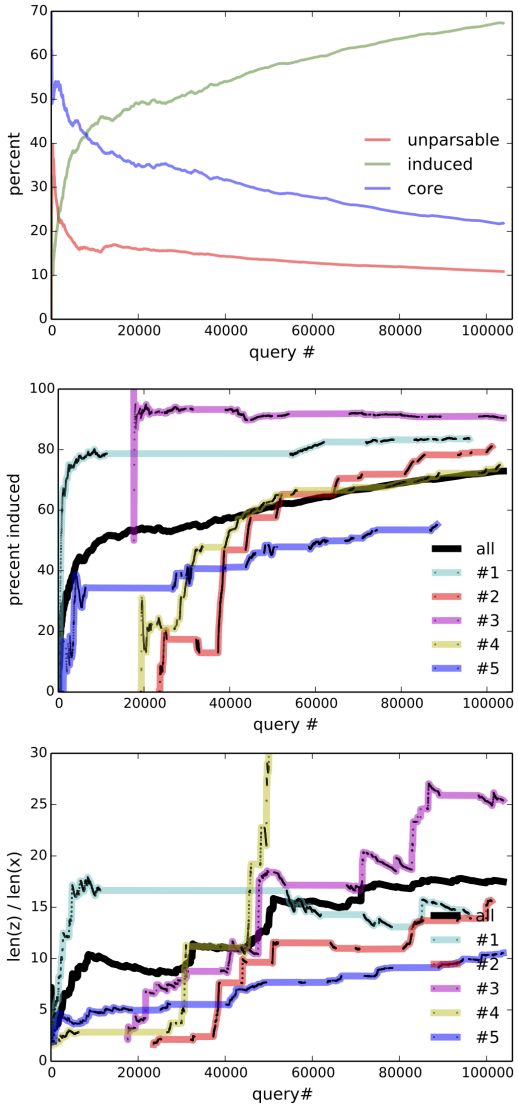


Figure 6: Learning curves. **Top:** percentage of all utterances that are part of the *core* language, the *induced* language, or *unparsable* by the system. **Middle:** percentage of accepted utterances belonging to the induced language, overall and for the 5 heaviest users. **Bottom:** expressiveness measured by the ratio of the length of the program to the length of the corresponding utterance.

concepts tend to be whole objects that are not parameterized (e.g., ‘dancer’). The bottom plot of Figure 6 suggests that users are defining and using higher level concepts, since programs become longer relative to utterances over time.

As a result of the automatic but implicit grammar induction, some concepts do not generalize correctly. In definition head ‘3 tall 9 wide white tower centered here’, arguments do not match the body; for ‘black 10x10x10 frame’, we failed to tokenize.

Short forms
left, l, mov left, go left, <, sel left
br, black, blu, brn, orangeright, left3
add row brn left 5 := add row brown left 5
Alternative syntax
go down and right := go down; go right
select orange := select has color orange
add red top 4 times := repeat 4 [add red top]
l white := go left and add white
mov up 2 := repeat 2 [select up]
go up 3 := go up 2; go up
Higher level
add red plate 6 x 7, green cube size 4,
add green monster, black 10x10x10 frame,
flower petals, deer leg back, music box, dancer

Table 3: Example definitions. See CodaLab worksheet for the full leaderboard.

Learned parameters. Training using L1 regularization, we obtained 1713 features with nonzero parameters. One user defined many concepts consisting of a single short token, and the Social.Author feature for that user has the most negative weight overall. With user compatibility (Social.Friends), some pairs have large positive weights and others large negative weights. The 'isolate' scoping choice (which allows easier hierarchical building) received the most positive weights, both overall and for many users. The 2 highest scoring induced rules correspond to 'add row red right 5' and 'select left 2'.

Incentives. Having complex structures show that the actions in Voxelurn are expressive and that hierarchical definitions are useful. To incentivize this behavior, we created a leaderboard which ranked structures based on recency and upvotes (like Hacker News). Over the course of 3 days, we picked three prize categories to be released daily. The prize categories for each day were bridge, house, animal; tower, monster, flower; ship, dancer, and castle.

To incentivize more definitions, we also track *citations*. When a rule is used in an accepted utterance by another user, the rule (and its author) receives a citation. We pay bonuses to top users according to their h-index. Most cited definitions are also displayed on the leaderboard. Our qualitative results should be robust to the incentives scheme, because the users do not overfit to the incentives—e.g., around 20% of the structures are not in the prize categories and users define complex concepts that are rarely cited.

7 Related work and discussion

This work is an evolution of Wang et al. (2016), but differs crucially in several ways: While Wang et al. (2016) starts from scratch and relies on selecting candidates, this work starts with a programming language (PL) and additionally relies on definitions, allowing us to scale. Instead of having a private language for each user, the user community in this work shares one language.

Azaria et al. (2016) presents Learning by Instruction Agent (LIA), which also advocates learning from users. They argue that developers cannot anticipate all the actions that users want, and that the system cannot understand the corresponding natural language even if the desired action is built-in. Like Jia et al. (2017), Azaria et al. (2016) starts with an ad-hoc set of initial slot-filling commands in natural language as the basis of further instructions—our approach starts with a more expressive core PL designed to interpolate with natural language. Compared to previous work, this work studied interactive learning in a shared community setting and hierarchical definitions resulting in more complex concepts.

Allowing ambiguity and a flexible syntax is a key reason why natural language is easier to produce—this cannot be achieved by PLs such as Inform and COBOL which look like natural language. In this work, we use semantic parsing techniques that can handle ambiguity (Zettlemoyer and Collins, 2005, 2007; Kwiatkowski et al., 2010; Liang et al., 2011; Pasupat and Liang, 2015). In semantic parsing, the semantic representation and action space is usually designed to accommodate the natural language that is considered constant. In contrast, the action space is considered constant in the naturalizing PL approach, and the language adapts to be more natural while accommodating the action space.

Our work demonstrates that interactive definitions is a strong and usable form of supervision. In the future, we wish to test these ideas in more domains, naturalize a real PL, and handle paraphrasing and implicit arguments. In the process of naturalization, both data and the semantic grammar have important roles in the evolution of a language that is easier for humans to produce while still parsable by computers.

Acknowledgments. We thank our reviewers, Panupong (Ice) Pasupat for helpful suggestions and discussions on lambda DCS, DARPA Communicating with Computers (CwC) program under ARO prime contract no. W911NF-15-1-0462, and NSF CAREER Award no. IIS-1552635.

Reproducibility. All code, data, and experiments for this paper are available on the CodaLab platform:
`https://worksheets.codalab.org/worksheets/0xbf8f4f5b42e54eba9921f7654b3c5c5d`
and a demo: `http://www.voxelurn.com`

References

Y. Artzi and L. Zettlemoyer. 2011. Bootstrapping semantic parsers from conversations. In *Empirical Methods in Natural Language Processing (EMNLP)*. pages 421–432.

Y. Artzi and L. Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics (TACL)* 1:49–62.

A. Azaria, J. Krishnamurthy, and T. M. Mitchell. 2016. Instructable intelligent personal agent. In *Association for the Advancement of Artificial Intelligence (AAAI)*. pages 2681–2689.

J. Berant, A. Chou, R. Frostig, and P. Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*.

G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam. 2017. Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant. In *World Wide Web (WWW)*. pages 341–350.

J. Duchi, E. Hazan, and Y. Singer. 2010. Adaptive subgradient methods for online learning and stochastic optimization. In *Conference on Learning Theory (COLT)*.

S. Gulwani and M. Marron. 2014. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *International Conference on Management of Data, SIGMOD*. pages 803–814.

R. Jia, L. Heck, D. Hakkani-Tür, and G. Nikolov. 2017. Learning concepts through conversations in spoken dialogue systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*.

N. Kushman and R. Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Human Language Technology and North American Association for Computational Linguistics (HLT/NAACL)*. pages 826–836.

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Empirical Methods in Natural Language Processing (EMNLP)*. pages 1223–1233.

P. Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408* .

P. Liang. 2016. Learning executable semantic parsers for natural language understanding. *Communications of the ACM* 59.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*. pages 590–599.

P. Pasupat and P. Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Association for Computational Linguistics (ACL)*.

S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. J. Teller, and N. Roy. 2011. Understanding natural language commands for robotic navigation and mobile manipulation. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

S. I. Wang, P. Liang, and C. Manning. 2016. Learning language games through interaction. In *Association for Computational Linguistics (ACL)*.

M. Zelle and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Association for the Advancement of Artificial Intelligence (AAAI)*. pages 1050–1055.

L. S. Zettlemoyer and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence (UAI)*. pages 658–666.

L. S. Zettlemoyer and M. Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*. pages 678–687.