Outline of a symbolic and PAC-learning approach to instruct a Turing-complete system in natural language

Anonymous ACL submission

Abstract

A machine's capability of interacting in natural language and processing in its full Turing-complete computing power is our ultimate expectation in natural language processing (NLP). When utterance carries knowledge, i.e., utterance are syntactical expressions for knowledge, we assume the former is isomorphic to a bounded function and the latter a partial recursive function. A bounded function is a total mapping from an infinite domain to a finite co-domain and a class of such functions is PAC learnable. We learn an infinite set of utterance by randomly collecting a finite set of sample utterance. Along with the sample utterance collection and processing, we also collect, precisely learn, a set of rules of parsing and semantic mapping as a side product of the utterance sample collection and processing. We call this process as a PAC learning, i.e., with a guaranteed performance, because the parsing and semantic mapping rules are interpreted as bounded function as well. In this paper, we propose utterance representation in the Enterprise-Participant (EP) data model that defines a class of bounded functions. We introduce Froglingo, a programming language extended from EP, to support the parsing and semantic mapping rules and additionally to support Turing-complete executions over utterance to vet knowledge.

1 Introduction

We assume knowledge is isomorphic to a partial recursive function involving a infinite number of entities. The utterance carrying knowledge that have occurred so far is finite in number and reference only a finite number of entities including some with infinite properties. In a natural language, there are (potentially infinitely) many paraphrases referencing a single entity (or a finite set of entities). A bounded function is a total mapping from an infinite domain to a finite co-domain, e.g., a cyclic graph possessing a infinite number of paths from a finite number of vertices is a bounded function. Therefore, we assume utterance is isomorphic to a bounded function.

The EP data model is a database language representing a class of bounded functions that is semantically equivalent to Turing machine, i.e., what Turing machine can produce within a finite time and space are nothing else but bounded functions. An applicative function, as long as it is total, is bounded. A (cyclic) graph with (infinite) paths can be represented in EP. A class of bounded functions is also Probably Approximately Correct (PAC) learnable, i.e., a finite set of sample EP expressions enables the EP data model to support an infinite set of fresh and meaningful EP expressions. In this paper, we propose to use the EP data model to represent utterance. These utterance are accumulated and learned into a database, i.e., a finite set of utterance is admitted into the database that supports an infinite set of fresh and meaningful utterance. When utterance are admitted to a database in the form of EP expression, they are Turing-complete expressions representing knowledge. Froglingo is a programming language based on EP, which extends the space of bounded functions to include partial recursive functions. We use Froglingo beyond EP to execute the Turing-complete expressions in the database carried in through utterance.

To efficiently map utterance to EP expressions, we need a precise parser. Instead of relying on a formal grammar or a semantic parser that maps utterance to knowledge representation without a grammar, we collect, precisely learn, a parser by constructing a parsing and semantic mapping rule in Froglingo whenever a sample utterance is sampled. Such a rule, which no longer needs to be reconstructed, is used to guide the parsing and semantic mapping process when a new utterance meeting the rule is to be processed. We say the process of collecting parsing and mapping rules is a process

of learning because the parsing and mapping rules are interpreted also as bounded functions, a class of which is PAC learnable.

The parsing and semantic mapping rules map utterance to intermediate syntactical structures first and eventually map the syntactical structures to their semantics, i.e., data in the database (note any EP and Froglingo expressions are data in the database as all are uniformly inventoried in the same database). The parser, including the rules, references data in the database when it processes an utterance and makes a decision on which rule is chosen for the utterance. Therefore, even if a utterance has a perfect grammatical structure and makes a full sense in the real world, the system may not recognize (read) it, as if a child didn't understand what an adult was talking about. This process is critical to maintain the realizability assumption of the PAC learnability and ensure that the database for knowledge will eventually converge. It is also critical to resolve and many times to avoid the challenges from natural language ambiguity that are important to linguistics and philosophy but unnecessary to most of NLP applications.

Constructing the parsing and mapping rules by PAC learning guarantees the performance converges to a designed level when the size of samples reaches upto a foreseeable amount. In other words intuitively, the performance is guaranteed because we only need to deal with a finite set of sample rules, where in the worst case we are able to tune individual rules that in general don't depends on each other because they are constructed from independently selected utterance samples.

In this paper, we only discuss knowledge representation and parsing in Froglingo because these two areas are the most critical ones in NLP.

In Section 2, we give related work in programming language and statistical machine learning. In Section 3, we briefly describe EP that has been covered in (Xu, 2017), (Xu et al., 2010), and (Xu, 2025). In Section 4, we briefly introduce Froglingo and a formal discussion is provided in Appendix A. In Section 5, we give example utterance and the corresponding Froglingo expressions to demonstrate what the PAC learnability is meant in using EP and constrained Froglingo expressions. In Section 6, we show why the Froglingo parsing and mapping rules can be precise. In Section 7, we introduce a user interface on how to interact with Froglingo in the native Froglingo expressions and in natural language.

2 Related work

Developing a shared functional space for both machine learning and symbolic computing has been a long and hard topic. For machine learning to utilize knowledge through symbolic computing, we must have a knowledge representation that is accessible to machine learning. Today's statistical machine learning accepts data such as raw images and utterance through flat-structured vectors. Embedding (injecting) knowledge like in graph into statistical machine learning is a process to integrating symbolic computing to statistical machine learning by converting knowledge with more complex data structure like graph into vectors and eventually into a dimension-reduced geometric space. This effort however has not been successful theoretically and practically, (Seshadhri et al., 2020), (Berant et al., 2012), (Nickel and Kiela, 2017), (Apidianaki, 2022).

There is an effort to integrate Turing machine computability into deep learning's data representation space using Recurrent Neural Networks (RNNs) and Transformers (Merrill and Sabharwal, 2024) as they are claimed to be Turing-complete (while this capability doesn't use the most powerful features in learning, e.g., weights and biases adjustment and backpropogation). Because Turing-machine is theoretically not PAC learnable (Gold, 1967), however, it is a question on how much properties of Turing-machine can be truly injected (transformed) to a data representation deep learning can access in terms of learnability. The bottom line is a controllable precision toward machine learning from symbolic computing is an advantage over statistical machine learning that doesn't keep the realizability assumption.

In today's learning theory supporting machine learning practice, all we have been talking about is binary and multiclass PAC learnability (Daniely and Shalev-Shwartz, 2014), (Shalev-Shwartz and Ben-David, 2014),(Brukhim et al., 2022), (Natarajan, 1989), which are actually bounded functions definable in EP and Froglingo. We believe a class of bounded functions is the mathematical limit for an efficient PAC learnability.

Formal grammars, such as combinatory categorical grammar (CCG) and phase structure grammar (PSG), are aimed to map the syntactical structure of all possible sentences (utterance) by only analyzing their grammatical structures. This approach is difficult to consider all various meanings of a utterance

and variant utterance that don't following the standard grammars. There hasn't been a known grammar implemented yet that would closely represent a natural language (Barton et al., 1987; Shieber, 1985; Gazdar et al., 1985).

Semantic parsers directly map utterance to their symbolic representations like in relational data Jiang-2024, logical form (Wang et al., 2020), and the lambda calculus (Poon, 2013). The expressiveness of such symbolic representations is either limited or too powerful to halt. Although statistical machine learning has advanced semantic parsing technologies that led popular applications in our daily life, such as language translation, question answering, and code generation, it is still a challenge to produce a parser that would be as reliable as a human being.

3 The EP data model

Identifiers are the most basic building blocks in EP. Like in programming languages, we can choose alphanumeric tokens as identifiers, such as $abc123$, $_abc$, and more often we take words from a natural language vocabulary as identifiers, such as $hello$, $John$, $sport$, law, and $person$. The entire set of identifiers, isomorphic to the set of integers, is denoted as $\mathbf{I}$. We also introduce a set constants $\mathbf{C}$, including elements from infinite domains such as integers, reals, and strings.

An EP term, or briefly term, is either an identifier or an application, i.e., using $\mathbf{E^0}$ to denote the entire set of terms, we have:

$$m \in \mathbf{I} \implies m \in \mathbf{E^0}$$
$$m \in \mathbf{E^0}, n \in \mathbf{E^0} \cup \mathbf{C} \implies (m\ n) \in \mathbf{E^0}$$

where a parenthesis '(' and ')' pair is omitted whenever possible. For example, $x\ x$, $a\ b\ c$ (actually for$((a\ b)\ c)$), $a\ b\ c\ (d\ e\ a\ (d\ t\ a))$, and $sq\ 3.14$ are legitimate terms where $x, a, b, c, d, e, t \in \mathbf{I}$ and $3.14 \in \mathbf{C}$. By EP terms alone, we can represent containment relationships. For example, the hierarchical structure of geographical locations can be expressed: $Massachusetts\ Boston\ Somerville$ and $Manhattan\ (Wall\ Street)$.

The containment relationships among subterms of a given term are tree-structured. The first kind is the relation of a term being contained within one of its leftmost subterms, denoted as {+. For example, we have: $Massachusetts\ Boston\ Somerville$ {+ $Massachusetts\ Boston$ {+ $Massachusetts$. The second kind is the relation of a term being contained within one of its rightmost subterms, denoted as {-. For example, we have $Manhattan\ (Wall\ Street)$ {- $(Wall\ Street)$ {- $Street$. Further the relations {+ and {- are extended to have tree-structured transitive relations {=+ and {=- respectively, e.g., $Massachusetts\ Boston\ Somerville$ {=+ $Massachusetts$ and $Manhattan\ (Wall\ Street)$ {=- $Street$.

An EP database, denoted as D, is a finite set of EP terms from $\mathbf{E^0}$ and a finite set of assignments such that for each assignment $p := q \in D$, where $p, q \in \mathbf{E^0}$, the following constraints are met:

1. p has only one assigner, i.e., $p := q$ and $p := q' \in D \implies q \equiv q'$;
2. a proper subterm of p cannot be an assignee, i.e., $p := q \in D \implies \forall x \in SUB^+(p)\ [\forall m \in \mathbf{E^0}\ [x := m \notin D]]$;
3. q must be a term without assigner $p := q \in D \implies \forall m \in \mathbf{E^0}\ [q := m \notin D]$

where $SUB^+(p)$ denote the proper subterms of p, e.g., the terms $a\ b\ c$ has the proper subterms $a, b, c, a\ b$. When a term is in D, so is one of its subterms. Here are a few sample assignments that can be in an EP database:

- $\{a\ b := b; b\ a := a; b\ c := c\}$, for a directed and cyclical graph with connections from a to b, from b to a, and from b to c.
- $\{College\ John\ major := College\ math;$ $College\ math\ math100\ (College\ John)\ grade := A\}$, for a college administration database.
- $\{New\ York\ City := NYC\}$, for a fixed English phrase and its abbreviation in EP.

A database D has a finite set of EP normal forms, denoted as $NF(D)$. A term $n \in \mathbf{E^0}$ is a normal form if and only if

1. n is $null$, i.e, $n \equiv null$, where $null$ is a special identifier in $\mathbf{I}$, or
2. n is a term in D and not an assignee, i.e., $n \in D$ and $\forall b \in \mathbf{E^0}\ [n := b \notin D]$.

To support queries, we give another set of EP terms, denoted as $\mathbf{E^1}$:

$$m \in \mathbf{I} \cup \mathbf{C} \implies m \in \mathbf{E^1}$$
$$m, n \in \mathbf{E^1} \implies (m\ n) \in \mathbf{E^1}$$

Given a database D, and an arbitrary term $a \in \mathbf{E^1}$, we have one-step reduction rules, denoted as $\Rightarrow$:

1. if a is an assignee, then it is reduced to the assigner, i.e., $a := b \in D \implies a \Rightarrow b$
2. if a is an identifier not in the database, then it is reduced to $null$, i.e., $a \in \mathbf{I}, a \notin D \implies a \Rightarrow null$

3. If $a \equiv a'\ b'$, a' and b' are normal forms, and $a'\ b' \notin D$, then $a'\ b'$ is reduced to $null$, i.e., $a', b' \in NF(D), a'\ b' \notin D \implies a \equiv a'\ b' \Rightarrow null$
4. $a \Rightarrow a' \implies C[a] \Rightarrow C[a']$, where $C[\ ]$ denotes a context, i.e., given any term $e \in \mathbf{E^0}$, we have $C[e] \in \mathbf{E^0}$ and e is a subterm in $C[e]$.

Let $a \Rightarrow a_0, a_0 \Rightarrow a_1, \ldots, a_{n-1} \Rightarrow a_n$ for a number $n \in \mathbf{N}$. We say that a is effectively, i.e., in finite steps, reduced to a_n, denoted as $a \rightarrow_D a_n$, and further we say that a and a_n are equal, denoted as $a == a_n$.

A term a has a normal form b if b is in normal form and $a \rightarrow_D b$.

Here are a few sample reductions to their normal forms under the example databases provided earlier: $a\ b\ a \rightarrow_D a$, $a\ b\ c \rightarrow_D c$, $c\ b \rightarrow_D null$, $college\ John\ major\ math100\ (college\ John)\ grade \rightarrow_D A$, and $New\ York\ City \rightarrow_D NYC$.

$\mathbf{E^1}$ is slightly bigger than $\mathbf{E^0}$ as $\mathbf{E^1}$ contains terms that have a constant as a leftmost subterm and are always reduced to $null$, e.g., $3.14\ a \rightarrow_D null$.

Any term $m \in \mathbf{E^1}$ has one and only one normal form and the reduction system is strongly normalizing, i.e., there is another term $n \in NF(D)$ such that $m \rightarrow_D n$. The set of all the normal forms $NF(D)$ is finite, i.e., $|NF(D)| \leq s$ for a given $s \in \mathcal{N}$.

An EP database is interpreted as a bounded function that is recursive and has a finite co-domain while an infinite domain. Let $f : X \rightarrow Y$ be a function, where X has an arbitrary number of objects and Y a finite number of objects, let $A \subseteq X$ and a (e.g., $null$ in the EP data model) be a special object in Y, i.e., $a \in Y$, and further let

$$\begin{aligned} f(x) &= b, \text{ where } b \in Y \text{ and } b \neq a, && \text{if } x \in A \\ &= a && \text{if } x \in X \backslash A. \end{aligned}$$

Then we say f is bounded, particularly when $A \equiv X$. When A is finite, i.e., $A \subset X$, we say f has a finite support, or simply we say f is finite. A finite function is bounded, but a bounded function may not be finite as it potentially has an infinite domain X. Further, a bounded function is always recursive, i.e., the computation on $f(x)$ terminates and $f(x) \in Y$ for any $x \in X$.

By saying a function being bounded, we mean that under a given EP database D, potentially an arbitrary number of EP expressions are meaningful. For example, the EP database defining a directed cyclic graph earlier has infinite paths, i.e., the terms $a\ b$, $b\ a$, $a\ b\ a$, ..., $a\ b ... a\ b$ will be reduced to non $null$ value. Even a path is not cyclic, we also call it derivable information. For example, we have $a\ b\ c \rightarrow_D c$ in the given example graph. These derivable (or called fresh) information from an EP database is the source of the PAC learnability of a class of bounded functions.

From the equality relation $==$, additional preordering relations are available. One of those relations, denoted as (=+, can be used to express if there is a path or cycle between two vertexes. For example, we have a (=+ $c \rightarrow_D true$ to verify a path from a to c, a (=+ $b\ and\ b$ (=+ $a \rightarrow_D true$ to verify a cycle between a and b, and c (=+ $a \rightarrow_D false$ to verify there is no path from c to a.

Like the relational data model, EP has a set-oriented operations using the operators $select$ and $there_is$. For example, the expression $select\ \$x\ where\ \$x$ {=+ $Massachusetts$ would retrieve all the terms, including $Massachusetts$, $Massachusetts\ Boston$, and $Massachusetts\ Boston\ Somerville$, that have $Massachusetts$ as a leftmost subterm. The expression $there_is\ \$x\ where\ c$ (=+ $\$x$ for the given example graph would inquire if there is a vertex that has a directed connection from c, which would be responded with the answer of $false$ because there is not a directed connection started from c in the graph.

The EP data model is said to be semantically equivalent to Turing machine because an approximation to the lambda calculus can be syntactically converted to an EP database.

4 Froglingo

Now we are ready to extend EP to Froglingo programming language. First of all, we allow variables, identifiers preceded with the \$ symbol and denoted as $\mathbf{V}$ for an entire set, to be part of an extended database, denoted as D again. For example, the factorial function can be expressed as $fac\ \$n := (\$n \times (fac\ (\$n - 1)))$, where $\$n$ is a variable, $\times$ is a built-in function for multiplication. The factorial function is a typical example demonstrating a language is Turing-complete as it has 1) a structure of recursion over fac and 2) an arithmetical function ($\times$) with an infinite co-domain that EP doesn't have. For simplifying our discussion in this paper, we consider an arithmetical function (operator) like $\times$ as an identifier in this paper while it would be more convenient to syntactically treat

an arithmetical function differently.

Secondly, we introduce types, denoted as $\mathbf{T}$ for an entire set, to provide constraints on variables. There are commonly available built-in types such as integers, denoted as $integer$, and strings, denoted as $string$, with infinite domains. There is a built-in transitive relational operator isA, representing an instance or a subtype is a type and further if we have $A\ isA\ B, B\ isA\ C$, we have $A\ isA\ C$. With the operator isA, we can constrain the variable $\$n$ in the factorial function to a non-negative integer: $fac\ \$n\ :\ [\$n\ isA\ integer\ and\ \$n \geq 0]\ :=\ (\$n\ \times (fac\ (\$n - 1)))$.

We use $\mathbf{B}_\infty$ to denote the set of all built-in arithmetical functions (with the most essential function $f(x) = x + 1$) as well as those built-in types.

In this paper, we are more interested in user-defined types because we can infer among types along with the operator isA. A type is syntactically an identifier. While an identifier remains to be itself when it is introduced to database using the built-in command $create$, a type must be introduced to a database by the built-in command $schema$. For example, we can have $schema\ vehicle$ and $schema\ person$ to declare types $vehicle$ and $person$, and $create\ joe$ to declare an object instance joe. We can further use isA to associate a type with another type, or a type with an instance. For example: $schema\ car\ isA\ vehicle$ and $create\ joe\ isA\ person$.

With the addition of variables $\mathbf{V}$ and types $\mathbf{T}$, we classify Froglingo databases into three categories:

1. $\mathbf{D_{EP}}$: a Froglingo database without variables and types is an EP database. We call the EP expressions in such a database EP data and denote all such database as $\mathbf{D_{EP}}$. Different from a traditional programming language, Froglingo always include the EP data model as an additional component that defines bounded functions[1]. Therefore EP is PAC learnable and terminates on arbitrary queries.
2. $\mathbf{D_{F_0}}$: a Froglingo database with identifiers, constants, user-defined types and variables constrained by the types through boolean expressions. Such a constrained database contains EP data and constrained expressions. While not all such databases terminate on all possible queries, there is a set of such databases that terminate all the time. We use $\mathbf{D_{F_0}}$ to denote all the databases containing EP data and constrained expressions that terminate all the time. We conclude:

   Theorem 2 $\mathbf{D_{F_0}}$ defines nothing else but a set of bounded functions, and therefore it is PAC learnable. (See Appendix A for the proof.)

3. $\mathbf{D_{F_1}}$: a Froglingo database with identifiers, constants, variables (constrained or not), user-defined types, and $\mathbf{B}_\infty$ (built-in types, and built-in arithmetical functions). Such a database contains EP data, constrained expressions, and Froglingo expressions with $\mathbf{B}_\infty$ that pushes Froglingo to be Turing-complete, a full programming language. We use $\mathbf{D_{F_1}}$ to denote all possible databases, particularly including expressions having $\mathbf{B}_\infty$ beyond $\mathbf{D_{EP}}$ and $\mathbf{D_{F_0}}$.

[1] strictly speaking, EP data only without constants purely defines bounded functions (Xu, 2017). But it doesn't make a big difference when we call all EP data, including constants, bounded functions.

5 Fresh information from sample utterance

Assume all utterance is isomorphic to EP expressions, we can map utterance to EP expressions first and then apply the converted EP terms to the PAC learnability. In this section, we give a few examples in natural language to demonstrate this idea that the EP-driven PAC learnability can actually be applied to the instance space of natural language.

A primary task of learning from utterance using Froglingo is to implement a parser that can map utterance to Froglingo expressions that represent the syntactical structure of the given utterance. Instead of a formal grammar, we use Froglingo expressions, called rules or templates, serving as a semantic parser for both syntactical parsing and semantic mapping. For the geographical containment relationships described in Section 2, for example, we can have the following template: $location\ (bE\ be)\ part\ (preP\ of)\ \$x\ :\ [\$x\ isA\ location]\ :=\ there_is\ \$y\ :\ [\$y\ isA\ location]\ \ where\ \$y$ {=+ $\$x\ and\ \$y$ {- $location$, where $location$ is a type for a geographical location, such as $Florida$, $Miami$, and $Florida\ Miami$, $\$x$ and $\$y$ are variables typed as $location$. When a sentence like: "Somerville is part of Boston" is to be parsed, the template above would guide the parser to break the utterance into the EP expression:

$Somerville\ (bE\ is)\ part\ (preP\ of)\ Boston.$

The template above guides the syntactical structure $Somerville\ (bE\ is)\ part\ (preP\ of)\ Boston$ to be reduced (mapped) to: $there_is\ \$y$: $[\$y\ isA\ location]\ where\ \$y$ {=+ $Boston\ and\ \$y$ {- $Somerville.$

Froglingo works in at least two modes as to be further discussion in Section 7: a learning (L) model and an operation (O) mode. When it is in L mode, it creates new data that would meet the $where$ clause, i.e., add $Boston\ Somerville\ isA\ location$ as a new fact into the database, where $Boston\ Somerville$ {=+ $Boston$ [2].

With the same template, the same construction process would be able to receive additional utterance such as "Boston is part of Massachusetts", which causes the database to be updated to have the term $Massachusetts\ Boston\ Somerville\ isA\ location.$ Now the database supports "Somerville is part of Massachusetts", which is a new piece of information. Reconstructing the database from the two separate sentences is a learning process enhanced from the learning algorithm on the instance space of EP terms.

We also allow utterance to be part of an assigner. For example: $person\ (verB\ drive)\ home\ :=\ Botran\ (person\ \text{"}come\ home\ by\ car\text{"})$. This assignment defines a rule that "a person drives home" is equivalent to "a person comes home by car", where we assume a template $person\ (verB\ come)\ home\ (preP\ by)\ car$ has already being constructed in the database. The identifier $Botran$ is a built-in term (operator) that obtains an instance of type $person$, e.g., $Jessie$, that is passed from the assignee at the left hand side, concatenates and adjusts the assigner to a text, e.g., "Jassie comes home by car", and parses the text back to a term in Froglingo, e.g., $Jassie\ (verB\ comes)\ home\ ((preP\ by)\ car)$. As a result, the following conversation would be newly meaningful to the NLP process: "Did Jassie come home by car?", "Yes, Jassie drove home". Given the utterance "Jassie came home by car" is a sample the NLP process learns, the utterance "Jassie drove home" is a prediction of the NLP process.

Breaking complex sentences into simple sentences is another way of learning (note a complex sentence is always broken down to multiple simple sentences in Froglingo). For example, the sentence "Joe with a hat walked on a street" is broken into two EP expressions: $Joe\ (verB\ wore)\ h0001$ and $Joe\ (verB\ walked)\ ((preP\ on)\ s0001$, where $h0001\ isA\ hat$ and $s0001\ isA\ street$, both the sentences "Joe wore a hat" and "Joe walked on a street" will be recognized and are predictions.

Learning through utterance can also be applied to types. When we define $cat\ isA\ animal$ and $tiger\ isA\ cat$, we will have $tiger\ isA\ animal$. As another example, we can specify two rules (templates) when $action$ had been declared as a type: $improvement\ isA\ action$, and $person\ (verB\ paint)\ house\ isA\ improvement$. When we have an instance: $joe\ (verB\ painted)\ ((coreF\ his)\ house)$, where the built-in $coreF$ abbreviates "coreference", we can infer the equivalence while $paint$ and $improvement$ are not paraphrases: $joe\ (verB\ improved)\ ((coreF\ his)\ house)$.

Many inferences in the real world are not based on rules from common knowledge but rather on individual instances. Froglingo supports a sequence of terms as an assigner (which is not formally introduced in Appendix). Froglingo uses assignments with a sequence of terms as an assigner to support such inferences. For example:

$John\ (verB\ took)\ (delimiT\ a)\ vacation\ :=$
$\quad John\ (verB\ visited)\ Jen,$
$\quad John\ (verB\ spent)\ ((delimiT\ a)\ day)$
$\quad\quad ((preP\ on)\ beach);$
$John\ (verB\ visited)\ Jen\ :=$
$\quad John\ (verB\ gave)\ Jen\ ((delimiT\ a)\ gift),$
$\quad John\ (verB\ had)\ dinner\ ((preP\ (together$
$\quad\quad with))\ Jen);$

An assignee above is an abstraction and the corresponding assigner is a sequence of sub actions that give the details of the abstraction. When an abstraction is equivalent to its sub actions, we can infer from the abstraction to a specific action, for example, from the assignments above, one can infer "John visited Jen" from "John took a vacation", and inversely we can speculate: "John took a vacation" when one hears "A gentleman gave Jen a gift". These inferences are predictions in the PAC learnability.

In the end of this section, we summarize what we expect:

[2] When Froglingo is in O mode, it would try to retrieve data $\$y$ from database to see if it meets the condition: $\$y$ {=+ $Boston\ and\ \$y$ {- $Somerville.$

Corollary 1 Provided the assumption that the set of utterance in a natural language is isomorphic to a bounded function, the utterance are efficiently PAC learnable.

Given the assumption, in other words, accumulating randomly selected sample utterance eventually converges to a machine that communicates with its surroundings in NL as if it was a human being.

6 Parsing and ambiguity

Collecting parsing templates from sample utterance is a bottom-up approach. A parsing template contains variables and (or) types for entity instances that are represented by phrases or clauses such as Joe and $the\ man\ with\ ahat$. A template may include individual verbs, such as $verB\ walk$, $verB\ drive$, and $bE\ is$. However, a template doesn't include a variable or a type for verbs. A template may include variables (types) for dependent clauses, such as the template $relative_clausE\ whom\ action$ where the type $action$ can be replaced by another template $person\ (verB\ meet)\ \$p : [\$p\ isA\ person]$ that is matched with a relative clause: "whom I met". But there is not a single rule (variable) to represent all possible sentences as a formal grammar does.

Given that the parsing and mapping rules have their isomorphic counterparts in utterance, we have an immediate corollary:

Corollary 2 Provided utterance is isomorphic to bounded functions, a set of parsing and semantic mapping rules can be defined by a database in $\mathbf{D_{F_0}}$. Therefore it is PAC learnable. (See the proof in Appendix A.)

When we say the parsing and mapping rules are learnable, it doesn't mean that we have an algorithm to construct such rules from utterance (it is an open question on if it is possible). We are saying that certain rules in formal grammars are difficult to be constructed in traditional programming, but they may not need to be constructed explicitly at all after we collected other relevant rules. If we need to give rule specifications for what a graph is, for example, we need to specify what a vertex, an edge, and a path are in traditional programming language. But we only need to specify what a vertex and an edge are without additional specification for a path.

The assumption that we deal only a finite entities give us another way to understand why a reliable performance of the parsing and mapping rules is guaranteed. The parser accepts a sentence only if the parsed structure is consistent with the up-to-date database, where a finite entities (those arithmetical expressions are not considered). One of the most important inquiries the parser performs is to find out if there is data (a set of EP terms) in the database that represents an entity or a category that a noun phrase in a given sentence is mapped to. Given a database defines $PNC\ isA\ bank$, where $bank$ has already defined as a type, and given John's bank account is represented as $PNC\ account\ John\ balance := 1000$, for example, we will be certain that the phrases "John's PNC Bank account" is mapped to $PNC\ account\ John$. Given a database having the set: $\{Family\ John; Family\ House\ (bE\ is)\ (preP\ in\ Florida);$ $Builder\ (verB\ built)\ (Family\ House)\ (preP\ in\ 2025);\}$ where $Family$, $John$, $House$, and $Builder$ are instances defined with types of $organization$, $person$, $house$, and $organization$ respectively, as another example, will certainly support the phrase "John's newly built Florida residence".

Even with the flexibility of the bottom-up approach in parsing, can we still end up with a difficulty not being able to represent certain grammatical structures by collecting parsing rules from sample utterance? The answer is no because of the assumption that we deal with a finte set of entities, i.e., each user-defined types has a finite set of instance. First, let's mathematically replace all variables and types with their instances in a parsing rule, e.g., replace $person$ with $Jessie$, Joe, ..., $Zack$ in $person\ (verB\ drive)\ home$ and we end up with multiple parsing rules $Jessie\ (verB\ drive)\ home$, $Joe\ (verB\ drive)\ home$, ..., $Zack\ (verB\ drive)\ home$. Assuming only a finite number of objects is our concern in representing the world knowledge, we still end up with a finite number of parsing rules if we replace the variables and the types with their instances in all collected parsing rules. Therefore, the rules without types and variables would be still effective and most critically precise in parsing utterance. We say such a parsing is precise because for each utterance: 1) if it is unique in meaning regardless of its context, we give its unique grammatical structure as it is; 2) if it is ambiguous in syntax, such as "Joe saw the girl with a binoculars", we can give its syntactical structure based on what the

speaker intended, i.e., referencing the Froglingo database for its true meaning before deciding its syntactical structure; 3) if it is ambiguous in semantics, such as "Joe got what he wanted" that is unique in syntax but could mean many things in meaning, we can still search the database for the context: who "Joe" is, What object Joe wanted and got, and if "got" is a "took", "received", or "purchased"; and 4) if it is ambiguous in pragmatics, such as "put the coffee on the table" while there are two tables next to each other as people often purposely or unconsciously say, we have to raise a question for clarification after a search on the database concludes the ambiguity.

Before we end this section, we need to have a good understanding how utterance isomorphic to bounded functions carries knowledge isomorphic to partial recursive functions. We assumed all utterance are in one-to-one correspondence with $\mathbf{D_{F_0}}$ so that utterance is PAC learnable. When Corollary 2 says the parsing and mapping rules are PAC learnable, it doesn't mean that we have an algorithm to construct such rules from utterance. How do we treat certain utterance for Turing-complete expressions, e.g., "A function takes a number and produces another number that is the given number plus 1" for $f(n) = n+1$ where n is a variable? We treat all utterance as EP expressions that are extended with variables, types, and $\mathbf{B}_\infty$. Therefore, when f \$n := + $\$n$ 1 for $f(n) = n+1$ is treated as an EP expression, we will have $f\ x = +\ x\ 1 \rightarrow_D null$ for arbitrary $x \in \mathcal{N}$ because $+$ is not defined in a corresponding EP database. This doesn't contradict with the reality when we have a Froglingo database, where we know $+$ is a built-in identifier and we execute it using its built-in operations instead of the standard EP reductions. Therefore we will have $f\ 1 = +\ 1\ 1 \rightarrow_F 2$, where $\rightarrow_F$ is the reduction rule from Froglingo and $\rightarrow_D$ is the reduction rules from the EP data model. By treating built-in types and arithmetical functions as constants C (Xu, 2017), we conclude :

Proposition 2 Froglingo expressions are in the form of EP expressions. (See more detailed in Appendix A)

7 Natural language along system language

Like in statistical machine learning, Froglingo consists of two major components: 1) a learning model and 2) a predictor - generated EP expressions in a database from positive utterance samples. A learning model consists of a parser, mainly a set of parsing and mapping rules. The learning model uses the parser to parse utterance by referencing the rules and the database to generate the predictor. Given a sample utterance, users can start to write templates against sample utterance. A predictor can be used anytime as soon as it is generated. The more sample utterance are provided, the richer model and predictor will be. When the system is accumulated with certain amount of rules/templates, additional templates may no longer needed even more sample utterance is needed to further enrich the learning model and the predictor. At this point, we would say that the NLP system "fully understands" a natural language, and that individuals without a software development experience would be able to operate in both L and O modes, driving knowledge accumulation and giving instructions in natural language for tasks that are required to be reliably executed.

In Appendix B, we focus on L mode using 4 sentences, including the first 3 from the famous folktale "Jack and Bean Stalk", to demonstrate how developers (users) interact with the Froglingo-based NLP system using both NL and Froglingo expressions to construct a database representing knowledge. The fourth example show how the factorial function can be described in utterance and converted to an EP expression that is convertible to $null$ in EP but a Turing-complete expression in Froglingo.

8 Conclusion

Provided that utterance is isomorphic to a bounded functions and knowledge is isomorphic to a partial recursive function, we showed utterance are expressions for knowledge, where the former is represented in an EP extended with variables, types, and built-in arithmetical functions and the latter in Froglingo.

Instead of the word "sentence", we used the word "utterance" in this paper in terms of NLP. We meant the text that can be processed includes not only grammatically correct sentences, but also various utterance including grammatically incorrect sentences and slangs. EP can model any language structures without a formal grammar. In addition, many similarity operations are supported by certain EP operators (Xu et al., 2010).

9 Limitations

Unlike a statistical machine learning approach, the symbolic approach to the bottom-up parsing rules needs manual development, at least supervising if a statistical machine learning can help automatically collect parsing rules.

The symbolic approach to the bottom-up parsing rules cannot directly be applied to the statistical machine learning technologies, though a mapping between the two may be developed. The latter uses similarities on geometric measures such as distance or cosine function among objects that are embedded to a Euclidean space to represent their relationships. The former uses transitive relations among the symbolic objects to calculate their relationships.

References

J. Allen. 1995. *Natural Language Understanding, second edition*. The Benjamin/Cummings Publishing Company, Inc.

M. Apidianaki. 2022. *From word types to tokens and back: a Survey of approaches to word meaning representation and interpretation*. Computational Linguistics (2023) 49 (2): 465–523.

G.E. Barton, R. Berwick, and E.S. Ristad. 1987. *Computational Complexity and Natural Language*. The MIT Press Classics.

J. Berant, I. Dagan, and J. Goldberger. 2012. *Learning entailment relations by global graph structure optimization*. publisher of Computational Linguistics, 38(1):73-111.

N. Brukhim, D. Carmon, I. Dinur, S. Moran, and A. Yehudayof. 2022. *A characterization of multiclass learnability*. 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS).

A. Daniely and S. Shalev-Shwartz. 2014. *Optimal learners for multiclass problems*. Proceedings of The 27th Conference on Learning Theory, PMLR 35:287-316, 20144.

Gazdar, Klein, Pullum, and Sag. 1985. *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge.

E.M. Gold. 1967. *Language Identification in the Limit*. Information and Controls, 10, 447-474 (1967).

W. Merrill and A. Sabharwal. 2024. *The expressive power of transformers with chain of thought*. International Conference on Learning Representations.

B. K. Natarajan. 1989. *On Learning Sets and Functions*. Machine learning 4, 67-97, 1989.

M. Nickel and D. Kiela. 2017. *Poincaré embeddings for learning hierarchical representations*. NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems Pages 6341 - 635.

H. Poon. 2013. *Grounded Unsupervised Semantic Parsing*. Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, pages 933–943, Sofia, Bulgaria, August 4-9 2013.

C. Seshadhri, A. Sharma, A. Stolman, and A. Goel. 2020. *The impossibility of low rank representation for triangle-rich complex network*. The Proceedings of National Academy of Sciences, March 2020.

S. Shalev-Shwartz and S. Ben-David. 2014. *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014.

S.M. Shieber. 1985. *Evidence against the context-freeness of natural langauge*. Linguistics and philosophy 8 (1985) 333-343, Reidel Publishing Company.

C. Wang, C.Ross, Y. Kuo, B. Katz, and A. Barbu. 2020. Learning a natural-language to LTL executable semantic parser for grounded robotics. *4th Conference on Robot Learning (CoRL 2020), Cambridge MA, USA*.

K. Xu. 2017. *A class of bounded functions, a database language and an extended lambda calculus*. publisher of Theoretical Computer Science, Vol. 691, August 2017, Page 81 - 106.

K. Xu. 2022. *A deductive system based on Froglingo for natural language processing*. DOI: http://dx.doi.org/10.13140/RG.2.2.13859.32807.

K. Xu. 2025. *Classes of bounded functions that are semantically equivallent to Turing machine are PAC learnable*. To present in the 38th Annual Conference on Learning Theory (COLT 2025) - Theory of AI for Scientific Computing Workshop, June 30–July 4, 2025 in Lyon, France. Preprint: DOI: 10.13140/RG.2.2.28499.49443.

K. Xu, J. Zhang, and S. Gao. 2010. *Higher-level functions and their ordering-relations*. The Fifth International Conference on Digital Information Management, 2010.

A Froglingo language specification

Now we are ready to extend EP to Froglingo programming language. First of all, we allow variables, identifiers preceded with the $ symbol and denoted as **V** for an entire set, to be part of an extended database, denoted as D again. For example, the factorial function can be expressed as $fac\,\$n := (\$n \times (fac\,(\$n - 1)))$, where $\$n$ is a variable, $\times$ is a built-in function for multiplication. The factorial function is a typical example demonstrating a language is Turin-complete as it has 1) a

structure of recursion over fac and 2) an arithmetical function ($\times$) with an infinite co-domain that EP doesn't have. We will use $\mathbf{B}_\infty$ to denote a set of all such arithmetical functions for a later discussion. For the time being and for simplifying our discussion, we consider an arithmetical function (operator) like $\times$ as an identifier in the following syntactical structure definitions even this consideration doesn't exactly fit the reality.

Secondly, we introduce types, denoted as $\mathbf{T}$ for an entire set, to provide constraints on variables. There are commonly available types such as integers, denoted as $integer$, and strings, denoted as $string$, with infinite domains. There is a built-in transitive relational operator isA, representing an instance or a subtype is a type and further if we have $A\ isA\ B, B\ isA\ C$, we have $A\ isA\ C$. With the operator isA, we can constrain the variable $\$n$ in the factorial function to a non-negative integer: $fac\ \$n\ :\ [\$n\ isA\ integer\ and\ \$n \geq 0]\ :=\ (\$n\ \times (fac\ (\$n - 1)))$.

In this paper, we are more interested in user-defined types because we can infer among types along with the operator isA. A type is syntactically an identifier. While an identifier remains to be itself when it is introduced to database using the built-in command $create$, a type must be introduced to a database by the built-in command $schema$. For example, we can have $schema\ vehicle$ and $schema\ person$ to declare types $vehicle$ and $person$, and $create\ joe$ to declare an object instance joe. We can further use isA to associate a type with another type, or a type with an instance. For example: $schema\ car\ isA\ vehicle$ and $create\ joe\ isA\ person$.

Now, we are ready to introduce the notion of Froglingo databases. We first give definitions on what are those assignees and assigners:

Definition 1 *The terms $\mathbf{E^2}$ potentially to be assignees in a Froglingo database:*

$$m \in \mathbf{I} \cup \mathbf{T} \implies m \in \mathbf{E^2}$$

$$m \in \mathbf{E^2}, n \in (\mathbf{E^2} \cup \mathbf{V} \cup \mathbf{C}) \implies (m\ n) \in \mathbf{E^2}$$

where when a variable $\$x$ appears in a term $m \in \mathbf{E^2}$, it can be optionally restricted by a boolean expression in the form of

$$\$x : [\texttt{bool-expression}]$$

where `bool-expression` is a sequence of binary expressions connected by the boolean operator and or or. Much like the boolean expressions used in traditional programming language and database management system, each binary expression has mathematical binary operators like $\leq$. It is further extended with Froglingo's transitive relational operators, such as {=+ and isA. For example, we define $tax\ \$x : [\$x \leq 200000] := (0.3 * \$x)$; $tax\ \$y : [\$y > 200000] := (0.4 * \$x)$; ($\$z$ {=+ $Boston$); ($\$p\ isA\ person$).[3]

Definition 2 *The terms $\mathbf{E}^3$, serving as potential assigners in a Froglingo database, is defined as:*

$$m \in \mathbf{I} \cup \mathbf{T} \cup \mathbf{C} \cup \mathbf{V} \implies m \in \mathbf{E}^3$$

$$m, n \in \mathbf{E}^3 \implies (m\ n) \in \mathbf{E}^3$$

A variable and a constant can be a leftmost subterm of a $\mathbf{E}^3$ term but not in $\mathbf{E}^2$, which is the main difference between $\mathbf{E^2}$ and $\mathbf{E^3}$.

Given a term $m \in \mathbf{E}^2$ or $m \in \mathbf{E}^3$, we use $FV(m)$, where FV abbreviates "Free Variables", to denote all types as global variables and local variables in m, i.e., $FV(m) = \{v : v \in m,\ v \in \mathbf{T} \cup \mathbf{V}\}$.

Definition 3 A Froglingo assignment is in the form of $m := n$, where $m \in \mathbf{E}^2$, $n \in \mathbf{E}^3$. We denote all such assignments as $\mathbf{A}$, i.e., $m := n \in \mathbf{A}$.

Definition 4 *A Froglingo database, denoted as D, consists of a finite set of terms from $\mathbf{E^0}$ and a finite set of Froglingo assignments from $\mathbf{A}$, which are entered in the following procedure:*

1. *if an identifier p is prompted in one of the following commands:*
   (a) $schema\ p$; *or*
   (b) $schema\ p\ isA\ q$; *where $q \in \mathbf{T}$,*
   then p is entered to D as a type, i.e., $pm \in D$ and $p \in \mathbf{T}$.
2. *if a term $p \in \mathbf{I}$ is prompted in the command $create\ p$, then p, remaining to be an identifier, is entered to D, i.e., $p \in D$.*
3. *if a term $p \in \mathbf{I}$ is prompted in the command $create\ p\ isA\ q$, where $q \in \mathbf{T}$, then p, remaining to be an identifier, is entered to D as an instance of q, i.e., $p\ isA\ q \in D$.*
4. *if $p \equiv p_1\ p_2\ ...\ p_i$ is prompted in the command $create\ p$, where $p_1 \in \mathbf{I}$, $0 < i < \infty$, and $p_j \in \mathbf{E^0}$ for each $1 \leq j \leq i$,*

[3] In the syntax of EP and Froglingo, which is originated from the lambda calculus, a binary operation like $x < y$ should be written as $< x\ y$. Aiming for a better readability, we keep the binary form as it is by always surrounding it with parentheses, e.g., $(x < y)$.

(a) *if p_1 was not defined as an instance of a type, i.e., $\forall t \in \mathbf{T}$, p_1 isA $t \notin D$, we continue to next step. Otherwise,*

(b) *if p_1 was defined as an instance of a type, i.e., $\exists t \in \mathbf{T}$, p_1 isA $t \in D$, we call "Type's instance creation process" provided in Definition 8, which depends on the reduction process to be given in Definition 7. After coming from the call, we finish the process by returning the results from the call*[4].

5. *if an assignment $p := q \in \mathbf{A}$ is prompted through $schema$ $p := q$ or $create$ $p := q$:*

   (a) *Perform the following validations:*

      i. *p has only one assigner, i.e., $p := q$ and $p := q' \in D \implies q \equiv q'$;*

      ii. *a proper subterm of p cannot be an assignee, i.e., $p := q \in D \implies \forall x \in SUB^{+}(p)\ [\forall m \in \mathbf{E^3}\ [x := m \notin D]]$; and*

      iii. *if $create$ $p := q$ is prompted, $p, q \in \mathbf{E^0}$, then q must be a term without assigner, i.e., $p := q \in D \implies \forall m \in \mathbf{E^3}\ [q := m \notin D]$*

      iv. *if the assignment is entered as $create$ $p := q$, ensure $FV(q) \subseteq FV(p)$*

      v. *for any subterm $m \in q$ and $m \in \mathbf{I}$, ensure m has already been in D, i.e., $m \in D$.*

   (b) *If one of the above validations fails, abort the process. Otherwise, continue to process:*

      i. *inventory p as it is, i.e., commit $p \in D$.*

      ii. *inventory each subterm e sequentially in the order it appears in q and keep all necessary parentheses (and) as they are.*

      iii. *finally, commit $p := q \in D$.*

Here are a few example expressions to accumulate Froglingo data in a database:

1. $create$ $multi$ $\$x$: $[\$x$ isA $integer]$ $\$y$: $[\$y$ isA $integer]$:= $(\$x * \$y)$. This expression represent a multiplication function (another example function having an infinite codomain beyond the $factorial$ function given earlier).

2. $create$ m $\$x := \$x$ $\$x$. This expression creates a function m that is equivalent to a lambda term $\lambda\ x.(x\ x))$ such that $m\ m$ is equivalent to the lambda term $\Omega \equiv (\lambda\ x.(x\ x))\ (\lambda\ x.(x\ x))$ that don't have a termination reduction process. In addition, $create$ n $\$x := \$x$ defines the identity function $\lambda x.x$. Therefore, Froglingo extended with variables can become Turing complete even without the built-in arithmetical functions $\mathbf{B}_\infty$. To focus on bounded functions, the Turing-complete aspects of Froglingo are not formally discussed in the paper.

3. $schema$ $bank$ $account$ $\$x$: $[\$x$ isA $person]$ $balance$:= $currency$, where $currency$ has been declared as a type inherited from the type real and $currency$ doesn't have to appear in the assignee. This expression serves as a constraint for the following instance creation:

4. $create$ PNC isA $bank$, $create$ PNC $account$ $Jacob$ $balance$:= 10000.00, where a number must be provided to match the type $currency$.

5. $schema$ $person$ $(bE$ $is)$ $\$p$: $[\$p$ isA $person]$ S $mother$:= $person$ $(verB$ $give)$ $birth$ $(preP$ $to)$ $\$p$, where S is kept here to match the apostrophe s in the English phrase "a person is another person's mother". Note that there are no new variables or types in the assigner other than those from the assignee, which will impact the instance creation process as to be seen in the following instance creation:

6. $create$ $Jassie$ isA $person$, $create$ $Jassie$ $(bE$ $is)$ Joe S $mother$, which commits $Jassie$ $(bE$ $is)$ Joe S $mother$ $\in D$, where the constraint's assigner $person$ $(verB$ $give)$ $birth$ $(preP$ $to)$ $\$p$ is not used in introducing $Jassie$ $(bE$ $is)$ Joe S $mother$ into database, but it will be used later to reason $Jassie$ $(verB$ $gave)$ $birth$ $(preP$ $to)$ Joe.

Different from an EP database that has only a finite set of normal forms, Froglingo have an infi-

[4]This case is for creating an instance by following a constraint. For example, a new instance "$Jassie$ $(verB$ $went)$ $(preP$ $to)$ $shopping$" must be matched with, i.e., meet, the constraint "$person$ $(verB$ $go)$ $(preP$ $to)$ $shopping$" where "$went$" and "go" are considered as a match before it can be committed into D.

nite set of "normal forms" that queries can end up with. An infinite set of objects as the co-domain can come from two sources: 1) the built-in arithmetical functions $\mathbf{B}_\infty$ and 2) Froglingo expressions with variables representing partial recursive functions like those in the 2nd example above. We will keep the ability of the built-in operators $\mathbf{B}_\infty$ as they are in Froglingo. On the other hand, we will develop a set of reduction rules over Froglingo expressions (without $\mathbf{B}_\infty$) such that we force Froglingo without $\mathbf{B}_\infty$ doesn't behave like the lambda calculus but like the EP data model.

First of all, we take the exact set of terms $\mathbf{E^1}$ as queries that are used for the EP data model. Secondly we use EP's normal forms $NF(D)$ for the co-domain of a Froglingo database:

Definition 5 *Froglingo normal forms - terms $NF(D)$ an $\mathbf{E^1}$ term will be reduced to:* A term n is an normal form, i.e., $n \in NF(D)$, if and only if

1. n is $null$, i.e, $n \equiv null$, where $null$ is a special identifier in $\mathbf{I}$, or
2. $n \in \mathbf{E^0}$, $n \in D$, and for any q such that $n := q \notin D$.

It's clear that $NF(D)$ under a Froglingo database is defined in the same way for the $NF(D)$ of an EP database as discussed in Section 3. As a matter of fact, variabls and types in Froglingo expressions do not contribute any additional normal forms into $NF(D)$. $NF(D)$ is finite as well:

Lemma 1 *Given a Froglingo database D, the set $NF(D)$ is finite.*

Proof 1 *This is because D consists of a finite number of terms in $\mathbf{E}^2$ and a finite number of assignments from $\mathbf{A}$. When D is finite, the normal forms, either $null$ or a subterms $e \in D$ and $e \in \mathbf{E^0}$ are also finite.*

We don't consider the entire set of the built-in arithmetical function operators $\mathbf{B}_\infty$ in the following reduction rules because the reduction rules to be defined will not allow an arithmetical function to produce infinitely possible outputs.

Given a database $D \setminus \mathbf{B}_\infty$ and any term $m \in \mathbf{E^1}$, we are now going to reduce it to a normal form in $NF(D)$. Before we define Froglingo reduction rules, we need to introduce another notation ENV, abbreviating "environment" that is commonly needed in programming language, which is originally initiated as $null$, i.e., ENV = $null$, to memorize intermediate computing results. Given a term $c\ v \in D$ where $c \in \mathbf{E^0}$ and v is a variable, we say a term $c\ a \in \mathbf{E^1}$ matches $c\ v$ if a meets v's constraint when v is defined with a boolean expression as the constraint in D. A term $c\ a \in \mathbf{E^1}$ always matches $c\ v$ if v doesn't have a boolean expression as its constraint. When $c\ a$ matches $c\ v$, we use $(c\ v)\ \{a/v\}$ to denote the state of the match, i.e., the new term having the instance of the variable v substituted with the value a. We remember this state by assigning it to an intermediate value ENV, i.e., ENV = $(c\ v)\ \{a/v\}$. When we continue to process $c\ a\ b \in \mathbf{E^1}$ to match $c\ v\ u$ where $u \in \mathbf{V}$ and $c\ v\ u \in D$, we will assign a new state to ENV, i.e., ENV = $(c\ v\ u)\ \{a/v, b/u\}$, when b meets u's constraint. We use ENV.term to reference the matched term and ENV.subs to reference the sequence of variable and value pairs, e.g., given ENV = $(c\ v\ u)\ \{a/v, b/u\}$, we have ENV.term = $(c\ v\ u)$ and $ENV.subs = \{a/v, b/u\}$. Even when we have found that a leftmost subterm n of a given $m \in \mathbf{E^1}$ is identical to a term $q \in D$, i.e., $n \equiv q$, we still need ENV to memorize q about where the reduction process has gone so far, i.e., $ENV = q\{\}$.

Definition 6 *(Froglingo reduction rules)* Given a Froglingo database D and a term $m \in \mathbf{E^1}$, we have the following Froglingo reduction rules:

1. $m \in D, m \in \mathbf{I} \implies m \rightarrow_F ENV = m\{\}$.
2. $m \notin D, m \in \mathbf{I} \implies m \rightarrow_F ENV = null\{\}$.
3. If $m \notin D, m\ isA\ t \in D$, then we build an ENV such that $ENV.term = t$ and $ENV.subs = \{m/t\}$ and $m \rightarrow_F ENV$.
4. In the case of $m \equiv a\ b$. By induction, we assume a has been matched to a term $a' \in D$ and $a' \in \mathbf{E^0}$, where a' is memorized in an environment $ENV1$, i.e., $ENV1.term \equiv a'$. Further by induction, we assume b has been matched to a term $b' \in D$ and $b' \in \mathbf{E^0}$, where b' is memorized in another environment $ENV2$. Further,
   (a) if $\exists\ p \in NF(D)$ such that $p \equiv ENV2.term$ and $a'\ p \in D$ and $a'\ p \in \mathbf{E^0}$, then we successfully found a match and memorize $ENV3$ = $(a'\ p)\ \{ENV1.subs\}$, and return $m \rightarrow_F ENV3$.
   (b) if there is a variable v such that $a'\ v \in D$, and further ENV2 meets v's constraint, then we successfully found a

match and memorize ENV3 = $(a'\ v)$ $\{ENV1.subs \cup \{ENV2.subs/v\}\}$. Return $m \rightarrow_F ENV3$.

(c) if there is a type t such that $a'\ t \in D$, and further $ENV2.term\ isA\ t$, then we successfully found a match and memorize ENV3 = $(a'\ t)$ $\{ENV1.subs \cup \{ENV2.subs/t\}\}$. Return $m \rightarrow_F ENV3$.

(d) otherwise, if there is not a variable or a type such that ENV2 meets the variable's constraint or belongs to the type respectively, then we assign $ENV3 = null\{\}$ and return $m \rightarrow_F null\{\}$.

5. When m is evaluated with an environment ENV, if there is no longer a term c such that $ENV.term\ c \in D$,

(a) if $ENV.term$ is not an assignee (without an assigner), then we conclude $m \rightarrow_F ENV$. (Note it should be $m \equiv ENV.term$ if ENV.term doesn't have a variable or type as a subterm.)

(b) when $ENV.term$ has an assigner, i.e.,$\exists n \in \mathbf{E^3}, ENV.term := n \in D$, we substitute the local variables and the global types appearing in n with the values memorized in ENV, i.e., we obtain a sequence of symbols $n\{ENV.subs\}$ by substituting all free variables in n with the corresponding values in ENV. Then we initiate $ENV = null\{\}$ for the sequence of symbols $n\{ENV.subs\}$, evaluate it by following the steps 1 to 3 above and obtain another environment ENV2. We return $m \rightarrow_F ENV2$.

In the end of the reduction process above, we expect an environment ENV with $ENV.subs = \{\}$ and we return $ENV.term$. To complete Definition 4.4 that depends on the reduction rules given above, we provide another definition:

Definition 7 *(Type's instance creation process)* During the database inventory process provided in Definition 4, when $p \equiv p_1\ p_2\ ...\ p_i$ is prompted in the command $create\ p$, where $p_1 \in \mathbf{I}$, $0 < i < \infty$, $p_j \in \mathbf{E^0}$ for each $1 \leq j \leq i$, and if p_1 was defined as an instance of a type, i.e., $\exists t \in \mathbf{T}$, $p_1\ isA\ t \in D$, we search D by providing $p_1\ p_2\ ...\ p_i$ as the input query term, particularly the search starts with the step given in Definition 7.3. If the search found a constraint $t\ t_2\ ...\ t_i := q \in D$ such that $p_1\ p_2\ ...\ p_i$ matches $t\ t_2\ ...\ t_i$, we obtain an environment ENV from the search and we continue to evaluate q with ENV according to Definition 7 as following:

1. if $FV(q) \setminus FV(t\ t_2\ ...\ t_i) \neq \emptyset$[5], then we require p is prompted along with an assigner matching the types in $FV(q) \setminus FV(t\ t_2\ ...\ t_i)$, i.e., $create\ p_1\ p_2\ ...\ p_i := y$[6]. Further we validate the input $p := q$ using the rules given in Definition 4.5.(a). Finally we commit $p_1\ p_2\ ...\ p_i := y$[7].
2. if $FV(q) \setminus FV(t\ t_2\ ...\ t_i) = \emptyset$[8], then we require p to not have an assigner. Further we validate the input p using the rules given in Definition 4.5.(a). Finally we commit $p_1\ p_2\ ...\ p_i \in D$. Note that the constraint's assigner q is not used when an instance is created. But it will be used later to reason the instance's property that is given in q[9].

The reduction rules in Definition 7 forces a term $m \in \mathbf{E^1}$ under a Froglingo database D to be mapped to a normal form in NF(D). Therefore a database represents a (partial) bounded function though not all terms $m \in \mathbf{E^1}$ are guaranteed to have a normal form.

Lemma 2 *Given a Froglingo database D defined in Definition 4, when the built-in arithmetical functions $\mathbf{B}_\infty$ are not considered, and given an expression $m \in \mathbf{E^1}$, the reduction process defined in Definition 7 ends with a normal form $n \in NF(D)$ that is finite when D is finite if the reduction process terminates.*

Proof 2 *Definition 7 defines a reduction process that reduces a term in $\mathbf{E^1}$ to a normal form in $NF(D)$ (as defined Definition 5) for a given database D, where the size of $NF(D)$ is finite per Lemma 6.*

[5] e.g., $bank\ account\ \$x : [\$x\ isA\ person]\ balance := currency$ where $currency$ is not given in the assignee

[6] e.g., $create\ PNC\ account\ Jacob\ balance := 10000.00$ where 10000.00 must be provided to match $currency$

[7] e.g., $PNC\ account\ Jacob\ balance := 10000.00 \in D$

[8] e.g., $person\ (bE\ is)\ \$p : [\$p\ isA\ person]\ S\ mother := person\ (verB\ give)\ birth\ (preP\ to)\ \$p$ where the assigner doesn't have new types or variables other than those from the assignee

[9] For example, when "$Jassie\ (bE\ is)\ Joe\ S\ mother$" is created, the $Jassie$ is automated given with the property: $Jassie\ (verB\ gave)\ birth\ (preP\ to)\ Joe$, which is a paraphrase of the created instance in English

Without variables, types, and arithmetical functions $\mathbf{B}_\infty$, the Froglingo system given in Definition 4 and the reduction rules given in Definition 7 precisely defines the EP data model that is described in Section 3 (we use $\mathbf{D_{EP}}$ to denote all such EP databases):

Proposition 1 *The EP data model is included within Froglingo.*

Proof 3 *It is clear from the definitions:* $\mathbf{E^0} \subset \mathbf{E^2}$, $\mathbf{E^0} \subset \mathbf{E^3}$, *as well as Definition 4 and Definition 7 that include the reduction rules for the EP data model as discussed in Section 3.*

Theorem 1 $\mathbf{D_{EP}}$ *is PAC learnable.*

Proof 4 *A class of bounded functions is PAC learnable (Xu, 2017). The EP data model defines classes of bounded functions and it is part of Froglingo.*

Corollary 1 *Provided the assumption that the set of utterance in a natural language is isomorphic to a bounded function, the utterance are efficiently PAC learnable.*

Proof 5 *Immediately derivable from Theorem 1.*

We formally introduce the notion of *partial* bounded function: a function $f : X \times Y$ is a partial bounded if the size of Y is finite and not all inputs $x \in X$ are guaranteed to be mapped to an element $y \in Y$.

Lemma 3 *Given a Froglingo database D defined in Definition 4, when the built-in arithmetical functions $\mathbf{B}_\infty$ are not considered, the database D represents partial bounded functions.*

Proof 6 *We can define m $\$x := \$x$ $\$x \in D$. This expression creates a function m that is equivalent to a lambda term $\lambda\, x.(x\ x)$ such that $m\ m$ would never be able to terminate according the reduction rules in Definition 7. As a matter of fact, it is equivalent to the lambda term $\Omega \equiv (\lambda\, x.(x\ x))\ (\lambda\, x.(x\ x))$ that don't have a termination reduction process.*

A Froglingo database is said to be type free, also said Froglingo type-free, if all the variables $v \in D$ are not constrained by a boolean conditional expression. Because the reduction rules in Definition 7 force the reduction results to be a finite set of normal forms, the type-free Froglingo without the built-in arithmetical functions $\mathbf{B}_\infty$ cannot be Turing-complete unless the size of a database is infinite.

We are more interested in a typed Froglingo: when all or some variables defined in $\mathbf{E^2}$ are assigned with boolean conditional expressions as constraints, we call a Froglingo database typed. A typed Froglingo database may not have to guarantee every (guery) term in $\mathbf{E^1}$ to be effectively reduced to a normal form, but it provides a tool to facilitate the development of a Froglingo database that will always have a chance to halt in execution all the time. In this sense, Froglingo is a programming language as a typed system.

We use $\mathbf{D_{F_0}}$ to denote the entire set of Froglingo databases such that each database terminates on all queries from $\mathbf{E^1}$. While we know a class of total recursive functions is not recursively enumerable (r.e.), it is an open question on if $\mathbf{D_{F_0}}$ is r.e or not. Nevertheless, we claim $\mathbf{D_{F_0}}$ is PAC learnable:

Theorem 2 $\mathbf{D_{F_0}}$ *is PAC learnable, i.e., the class of functions defined by the databases in* $\mathbf{D_{F_0}}$ *is PAC learnable.*

Proof 7 *Given a Froglingo database $D \in \mathbf{D_{F_0}}$, D defines a bounded function according to Lemma 9 and the definition of* $\mathbf{D_{F_0}}$. *Because a class of bounded functions is PAC learnable, we also say* $\mathbf{D_{F_0}}$ *is PAC learnable.*

As a matter of fact, a PAC learnability learnable doesn't require it is r.e.. For example the PAC learnable class of rectangle learning games is not r.e. because there are uncounable number of target rectangles in the class.

The theorem doesn't have to say we have a learning algorithm to construct a Froglingo database from $\mathbf{D_{F_0}}$, particularly those serving as constraints having types and variables. Instead, we say the PAC learnability of $\mathbf{D_{F_0}}$ can be applied to rule specifications in programming language. Specifically, we argue that specifying rules in the top-down approach of traditional programming languages is more expensive than specifying rules in the bottom-up machine learning approaches, including $\mathbf{D_{F_0}}$. In other words, specifying certain rules that may not be easily done practically in traditional programming language can be done feasibly by database accumulation in Froglingo. Such database accumulation process can be said to be as effective as the targeted formal rules that are available theoretical but can not be easily implemented in programming languages. If we need to give rule specifications for what a graph is, for example, we need to specify what a vertex, an edge, and a path are in traditional programming language.

But we only need to specify what a vertex and an edge are without additional specification for a path. Therefore we say the bottom-up approach using Froglingo is easier because we don't need to continue to accumulating rules for paths after we collected the rules for vertices and edges.

Formally, we give a corollary immediately available from Theorem 2:

Corollary 2 *Provided utterance is isomorphic to bounded functions, a set of parsing and semantic mapping rules can be defined by a database in* $\mathbf{D_{F_0}}$. *Therefore it is PAC learnable.*

Proof 8 *Because utterance are isomorphic to bounded functions, we can build constrained expressions having variables and types, serving as the parsing and semantic mapping rules, to guide utterance parsing and semantic mapping, because the rules have isomorphic counterparts in utterance. According to Theorem 2, such rules are nothing but Frolingo expressions from* $bf\,D_{F_0}$. *Therefore we conclude the rules are bounded functions and are PAC learnable.*

Is there an algorithm to learn and construct the exact parsing and mapping rules from utterance? It is an open question. We don't need to have such algorithm theoretically because the PAC learnability is about algorithm to construct a second function (hypothesis) that approximate a target function that doesn't need to be constructed. However, through statistical learning like today's Transformers and RNNs, we should be able to learn a version of the rules that may not be as succinct and accurate as we expect but a slightly different version would still be helpful to reduce the manual work in constructing the rules.

We have assumed utterance are Turing-complete expressions for knowledge and utterance is isomorphic to a bounded function represented in an EP database. To understand an EP database expresses Turing-complete expressions including types, variables, and built-in arithmetical functions, we have to add those extra symbols syntactically into EP and assign them to be practically meaningless. We enhance the EP reduction rules to accept terms $\mathbf{E^3}$ but just simply treat them meaningless ($null$):

Definition 8 *Extended EP reduction rules for saying EP carries Turing complete expressions:*

$$\forall m \in \mathbf{B}_\infty \cup \mathbf{V} \cup \mathbf{T}, \forall n \in \mathbf{E}^2 \implies m\,n \Rightarrow null$$

While the extended EP reduction rules will never be used in reality, as the system can tell these non-native EP terms are part of Froglingo expressions, we conclude:

Proposition 2 *Froglingo expressions from* $\mathbf{E^3}$ *are in the form of EP expressions.*

Proof 9 *We take a term from* $\mathbf{E^3}$ *and feed it to the EP reduction rules (described in Section 2, formally in (Xu, 2017)). Because of the extended rule in Definition 8, the EP reduction will not produce any additional new normal form. In other words, EP keeps what it is in representing bounded functions although EP is added with extra non-essential syntactical forms from variables, types, and built-in arithmetical functions.*

B Example utterance learning using a user interface

This supplementary material serves as a demonstration only, which may contain inaccurate information. But through this section, readers may have a brief understanding on how a symbolic NLP process works and what are the concerns needed in tackling NLP challenges with such a symbolic approach.

ID	Templates, part of Learning Algorithm	Predictors, automatically generated	Effect	Description
`Once up on a time there lived a poor widow and her son Jack.`				
1	*person isA thing;* *wife isA person;* *widow isA wife;*	None	No immediate impact yet, but any instances including *p0001* in Row #3 with the type *widow* can be predicted as a *wife, person*, and $thing$ through the lifecycle of the NLP process.	Before the three templates entered, the database was empty except for $thing$ the only user defined data representing the sole root type.
2	*adverB (once up on a time) := there_is $t: [$t isA time] where $t start < $t end and $t end < '1/1/1000';*	*t0001 isA time;* (where an internal structure is created for: *tp0001 start < tp0001 end < '1/1/1000'*).	The L mode process actually started an inquiry first by executing the expression $there_is$ .... It created the data because it didn't find any relevant data in the new context that was established for the Jack and Bean Stalk forktale.	The node *t0001* contains partial information because its $start$ and end are not assigned with an exact date. A time always has a start time and an end time for a period. When the start and end times are the same, a time becomes a point of time. The value '1/1/1000' is randomly chosen for a demonstration purpose to reference a time in the past.
3	*family isA thing;* *a widow := there_is $f: [$f isA family], $p: [$p isA person] where $f $p isA widow;*	*p0001 isA person; f0001 isA family; f0001 p0001 isA widow;*	*p0001, f0001, and f0001 p0001* reflect the basic information from the sample text, an interpretation from human experience.	The word `widow` must be involved with a group of people, $family$. The *f0001* family is created to have *p0001* and later $jack$ and $c0001(a\ cow)$ as members.
4	*adJ poor $p: [$p isA person] := there_is $f:[$f isA family] where ($f $p != null and $f (bE be) (ajD poor));*	*f0001 (bE be) (adJ poor);*	The inquiry `who is poor?` would have an answer by matching expressions like *f0001 (bE be) (adJ poor).*	Adjectives like $poor$ is semantically more complex, involving statistics. No additional adjectives are discussed in this table. Additionally, we manage to say Jack's family is poor, instead of saying Jack's mom is poor.
5	*her son := (coreF her) S son;*	No data is created, but a interim syntax transformation from *her son* to *p0001 S son*.	No effect, only a syntax transformation	$coreF$ abbreviates coreference. $coreF\ her$ retrieves the `widow` in the story. S abbreviates the `'s` symbol after a person's name. Therefore `her son` is transformed as `the widow's son`
6	*son isA person;* *mother isA person;* *person S son : = there_is $f: [$f isA family], $s: [$s isA person] where $f person isA mother and $f $s isA son;*	*p0002 isA person;* *f0001 p0002 isA son;* *f0001 p0001 isA mother;*	*p0002* is the new data added in database, referring to Jack. Also specify that the widow is a mother.	The phrases like `her son` have a fixed syntactical form. They are defined once and reused for later to parse other text. In the 3rd assignment of the "Template" column, $person$ and son are global types and acting as variables.
7	*name isA thing;* *person name : = (person namE == name);*	*p0002 namE := jack;*	$jack$ from now on is a coreference to $p0002$, i.e., *coreF jack := p0002*.	The text `son Jack` matches the 2nd template. `Jack` is categorized as a name while others are possible, e.g., a machine. All text from users are converted to small cases while capital cases are memorized separately.
8	*there (verB live) person := person (verB live);*	*p0001 (verB live) (preP in) t0001;* *p0002 (verB live) (preP in) t0001;*	The entire English sentence is now not mapped to two EP terms, each represents the state of being living without detailed semantics for now	The parser splits the text into two sentences because of the conjunction word `and`. Therefore, there are two corresponding EP terms.
9	*husband isA person;* *widow (dO do) (noT not) (verB live) (preP with) husband := there_is $f: [$f isA family] where widow {+ $f and ! (there_is $h : [$h isA husband] where $h {+ $f);*	no new data is created because there is no text in corresponding this template. This template is optional to give a constraint that a widow doesn't not have or live with a husband.	This constraint may not necessarily be enforced. But It can be invoked for validation in an I mode.	We could add more semantics by adding more templates like this one to enrich the understanding of this sentence.
`One day, Jack's mother told him to sell their only cow.`				

10	*adverB (one day) := there_is $t: [$t isA day] where (coreF (one day)) start < $t and $t < (coreF (one day)) end;*	*t0002* (where an internal structure is created for *t0001 end* $\leq$ *t0002 start* $\leq$ *t0001 end, and t0002 end - t0002 start = 24 hours)*	$coreF(one\ day)$ is mapped to be "Once upon a time" that was recorded earlier. *t0002* is created for a period of time within once upon a time.	The phrase `One day` can be a future day or a past day but unsure exactly which day it would be when it serves as an adverb in a sentence. In the given sentence, it is a past tense and within the time period *t0001* set by `Once up on a time`. *coreF (one day)* finds out the tense of the sentence first, e.g., the past tense in this case, and then searches a previously defined time period constraining `one day`.
11	*coreF name := there_is $p: [$p isA person] where $p namE == name;*	No data is created	*coreF jack* returns *p0002*.	`Jack` may not only refers to a person but also others such as a tool to lift a car. Therefore `Jack` was tried to be parsed in different categories before confirming it is the name for p0002.
12	*person S mother : = there_is $f: [$f isA family], $p: [$p isA person] where ($f person isA son or $f person isA daughter) and $f $p isA mother;*	No data is created	*f0001 p0001* is returned as Jack's mother, where Jack is an instance of $person$.	Since the L mode process found the instance *f0001 p0001*, it doesn't create a new one but retrieve the existing one.
13	*animal isA thing;* *livestock isA animal;* *cow isA livestock;*	No data is created.	No immediate impact yet	However, any instances including *c0001* in Row #15 with the type *cow* can be predicted as a *livestock, animal*, and $thing$ through the lifecycle of the NLP process.
14	*coreF their;*	No data is created	*f0001* is returned. Like other coreferences, *coreF their* is determined by a built-in process	$coreF\ \ they$ returns Jack and his mother, but $coreF\ \ their$ returns $f0001$, something shared by both Jack and his mother.
15	*family S cow := family cow isA livestock;*	*c0001 isA cow* *f0001 c0001 isA livestock*	*c0001, f0001 c0001* reflect the basic information from the original text `their cow`	In the L mode, the process tried to find a cow instance in $f0001$ and created $c0001$ because no one was found.
16	*person (verB tell) $p: [$p isA person] Infinitive;* *person (verB sell) $t : [$t isA thing];*	*p0001 tell jack (jack sell c0001 ((PreP at) Notime));* Note the action represented by *tell* is implicitly modified by a time within the given `One day`. Therefore the sentence is actually in past tense because the system can tell.	The constructed data is a predictor, which represents an action taken by Jack's mother. Subsequent queries can be answered such as `what did Jack's mother tell Jack?`, `Who told Jack to sell their cow?`, etc..	*Infinitive* is a built-in operator indicating the following text is an infinitive phrase, i.e., `to do ....` The built-in node $Notime$ indicates that *jack sell c1000* is not a fact yet as it may or may not happen. This sentence with the verb $sell$, converted from the infinitive clause, will be further updated to make the $sell$ a fact in Row #28.
17	*desirE 1 := desire; desirE 2 := like; desirE 2.5 := want, desirE 3 := hint; desirE 4 := encourage; desirE 5 := tell; desirE 6: = ask; desirE 7 := command; desirE 8 := enforce};*	No data was generated		Facing the phrase "Jack's mother told him to ...", we optionally construct a template $desirE$ that places all verbs related to "desire" in a sequence to reflect the degree of desires. This template helps to correlate similar sentences together to find paraphrase sentences.
`Jack went to the market and on the way he met a man who wanted to buy his cow.`				
18	*coreF name := there_is $p: [$p isA person] where $p namE == name;*	No data is created	*coreF jack* returns *p0002*.	The same process as discussed in #11.
19	*location isA thing;* *market isA location*	No data was created	No immediate impact yet	However any instances including *m0001* in Row # 20 with the type *market* can be predicted as a *market, location*, and $thing$ through the lifecycle of the NLP process.
20	*the market := there_is $m: [$m isA market];* Note `a market` would be given the same definition as our process doesn't rely on `a` or `the` vigorously	*m0001 isA market*	*m0001* reflects the basic information of the original text `the market`, an interpretation from human experience.	While this template is defined based on human experience, it can also be derived by the sentence `Jack went to the market`, where `the market` can be reasoned as a location, where there is a template like the one in #21.

21	*person (verB go) (preP from) $l1: [$l1 isA location] (preP to) $l2: [l2 isA location] := (update person geoLoc := $l2 geoLoc);*	*l0002 isA location; f0001 geoLoc := l0002; p0002 geoLoc := m0001;*	*l0002* refers to Jack's home location, a dump node without information	There should be a standard geographical (and time) data calculation to construct the first and second assignments on the Predictor column. The update command in the template enforces an update for both L and O modes
22	*adverB (on the way (preP from) $l1: [$l1 isA location] (preP to) $l2: [$l2 isA location]) := there_is $l: [$l isA location] where $l* `is between` *$l1* `and` *$l2;*	*l0003 isA location;* where *l0003* `is between` *l0002* `and` *m0001*;	the geographical distance should be implemented in a standard geographical calculation package	
23	*articlE a man = there_is $p: [$p isA person];* Note a man can be defined with more attributes but we simply define it as a person just for demonstration purpose.	*p0003 isA person;*	A template for "the man" would be the same one for "a man", as our process doesn't differentiate "a" from "the" to tolerate human errors.	However, the parser would prefer to create a new person because of "a man" is given. Otherwise, considering "the man" being Jack himself would make the sentence "Jack met the man", where "the man" is Jack, not meaningful.
24	*person (verB meet) $p: [$p isA person] (preP at) location := ((person geoLoc == location) and ($p geoLoc == location));*	*p0002 geoLoc := l0003; p0003 geoLoc := l0003;*	The template enforces the geoLoc of the two persons to be changed	The template can be enriched with more attributes, but the same location the two persons met is the highlight of this template.
25	*person (verB want) infinitive; person (verB buy) $t: [$t isA thing];*	*p0003 (verB want) (p0003 (verB buy) c0001 ((preP at) Notime));*	`his cow` is mapped to *c0001* based on the similar process we discussed earlier	*person (verB want) infinitive* is very similar to *person (verB tell) $p: [$p isA person] infinitive* and they can be correlated using the $desirE$ template in #17.

`Jack took the magic beans and gave the man the cow.` Note: in our discussion, we skipped the conversations between Jack and the man, where the man's 5 magic beans would be traded to Jack for Jack's cow. To simplify our discussion, we assume only one bean and the following data have been generated: *f0002 isA family; f0002 p0003; bean isA thing; b0001 isA bean; b0001 (bE be) (adJ magic); f0002 bean;* where we consistently set up an organization, such as a family, a person belongs to.

26	*person (verB take) thing from $p: [$p isA person] := Botran ($p (verB give) person thing);*	No data is generated.	The text `Jack took the magic beans` is to be converted to *p0003 (verB give) p0002 b0001* in #27.	It defines that a person takes a thing from another is equivalent to that the second person gives the first person the thing. The original text doesn't have the phrase `from the man` but the template gives the parser a hind to find it.
27	*person (verB give) $p1 : [$p1 isA person] $t: [$\widehat{isA}$ thing] := (person geoLoc == $p1 geoLoc), delete coreF (family person) $t, create coreF (family $p1) $t;* *coreF (family person) := select $f: [$f isA family] where $f person != null;*	*f0001 b0001* and *f0002 c0001* were added into database, and $f0002$ $b0001$ and $f0001$ $c0001$ are removed from database. The results come from the intermediate expressions: *p0003 (verB give) p0002 b0001; p0002 (verB give) p0003 c0001;*	The template is aimed to first validate that the persons and the good to be exchanged are next to each other, e.g., the geographical coordinates are the same. Then it update the belongings of both Jack and the man in their family accounts.	When a sentence implies actions like "give", we use the Froglingo update commands in template explicitly to enforce the action for both L and O modes. Also the built-in term $coreF$ can be user-defined this time.
28	*person (verB sell) $g: [$g isA thing] := there_is $buyer: [$buyer isA person] where person (verB give) $buyer $g;*	No data was generated as there is no text to trigger an execution on it	This template would help to validate `Did Jack sell his cow`?, which is related to the word "sell" in the sentence "Jack's mother told him to sell their only cow" at an I mode	An extra step to demonstrate that new information can be derived from the sample text by using the template.
29	*person (verB buy) $g: [$g isA thing] := there_is $seller: [$seller isA person] where $seller (verB give) person $g;*	No data was generated	Not used in this demonstration. This template would help to answer the question: `Did the man buy his cow?`	An extra step to demonstrate that new information can be derived from the sample text by using the template.

`What does the function fac take 5 to produce?` Note: though natural language itself is ambiguous, there are some text that can precisely express vigorous mathematical expressions.

30	*multiplication $n1: [$n1 isa number] $n2: [$n2 isa number] = ($n1 multi $n2);* *fac (verB take) $n:[$n isA number] (preP to) (verB produce) $m:[$m isA number] = Botran (* "if" $\$n$ "is 0," $\$m$ "is 1 or" $\$m$" $is\ the\ multiplication\ of$" $\$n$ "$with\ what\ fac\ takes$" $(\$n - 1)$ "$to\ produce;$");	No data is generated in database, but respond with an answer of 120.	The answer to the given text is 120. The template acts precisely as a factorial function	