Appendices – Supplementary Material

A. Related work

While LLMs have achieved remarkable performance in NLP that has changed our daily life, the statistical based machine learning approaches have been continuously adopting various technologies, such as Transformers' chain of thought, to increase the capacity in reasoning and aim to approach human's intelligence, termed as Artificial General Intelligence (AGI) in literature [Merrill and Sabharwal(2024)]. In parallel, Froglingo is a symbolic and PAC learnable approach toward the same direction. Though not in practice but theoretically, Froglingo has a clear advantage over LLMs that it has the controllable precision toward NLP because it is a symbolic and PAC learning approach.

Another relevant effort from statistical machine learning is source code generation using LLM [Brown et al.(2020), Jiang et al.(2023)]. On the other hand, Froglingo doesn't generate programming language code but execute it's own Turing-complete expressions (see Sample #30 in Appendix for an example). When the former is dedicated to code generation, the latter, as a single system, is also aimed to generate NL.

The computing power of a PAC learnable in terms of computability is the primary factor to determine how much a machine can learn toward NLP while reasoning from non-PAC learnable programming language may help on certain targeted tasks. We know the class of conjunctions of boolean literals is PAC learnable [Kearns and Vazirani(1994)] and a class of bounded functions is PAC learnable [Xu(2025), Xu(2024)]. The practice of statistical machine learning like LLMs has demonstrated a massive portion of the computing power of Turing machines, as we have observed. But it is still an open question on how much the computing power is exactly, especially when the convergence in statistical approaches is not guaranteed. The bottom line is that a class of partial recursive functions represented by a Turing-complete machine cannot be effecitively PAC learnable [Gold(1967)]. There are many different notions of learning beside the PAC learnability, such as online learning [Littlestone(1987)] and learning through enumerating a sequence of languages in a known language class [Kleinberg and Mullainathan(2024), Li et al.(2025)]. We focus on the PAC learnability because it is the most relevant to NLP.

B - Froglingo language specification

In Section 2, we introduced EP terms, denoted as the set $\mathbf{E}$:

Definition 1 *The set of EP terms* $\mathbf{E}$ *containing identifiers only are defined as following:*

$$m \in \mathbf{F} \implies m \in \mathbf{E}$$

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

where $\mathbf{F}$ is a set of identifiers aimed to represent bounded functions that are definable through an EP database. (Note there are an arbitrary number of identifiers and only a finite number of them can appear in an EP database.) In [Xu(2017)], EP was also introduced with a set of constants $\mathbf{C}$, aimed to represent partial recursive functions definable by the Turing machine including integers and mathematical functions like the multiplication and a factorial function. The terms allowed to be in an EP database is the set $\mathbf{E}^1$:

$$m \in \mathbf{F} \implies m \in \mathbf{E}^1$$

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

where the behavior of applying a constant to another term was not considere, i.e., $c\ m$ for $c \in \mathbf{C}$ and $m \in \mathbf{E^1}$, was not allowed in an EP database.

In this section, we extend EP to Froglingo that allows constants to be defined with their behaviors. The notion of constants in Froglingo is meant more than ground values like an integer 23 and Turing complete expressions constructed on the top of built-in mathematical operations like plus and multiplication. We use $\mathbf{C^1}$ to denote the constants based on built-in mathematical operations. It is also meant Froglingo expressions constructed on the top of variables that demonstrate the computation behaviors of the Froglingo reductions. We use $\mathbf{C^2}$ to denote this set of constants. We will note later that $\mathbf{C^2}$ has nothing more than bounded functions.

To support constants, we need to introduce variables, placeholders for values constants can be applied to. There are two kinds of variables: one is local variable such as $\$x$ for $fac\ \$x$ and the other is global variables, also serving as types, such as the built-in *integer* type for the entire integers and a user defined *person* type.

Because identifiers, types, and constants share a single set of tokens when they are named in a Froglingo database, we rename the symbol $\mathbf{F}$ in Section 2 to be $\mathbf{G}$, abbreviating the phrase "ground tokens", and keep the symbol $\mathbf{F}$ for its members, again called identifiers, to represent bounded functions definable in an EP database. An EP database is now called a Froglingo database that is extended with Turing-complete expressions ($\mathbf{C}^1$) and Froglingo expressions representing bounded functions ($\mathbf{C}^2$).

When a ground token is defined into a database by the built-in operator *schema*, e.g., *schema person*;, the token becomes a type. We use $\mathbf{T}$ to denote all such types, e.g., $person \in \mathbf{T}$. Given a type, we can further define a subtype using *schema*, e.g., *boy isA person*.

A type's instance is introduced to database by the built-in command *create* and the built-in operator *isA*, e.g., *create Jacob isA boy* to create *Jacob* tagged with the type *boy*. When an instance is created into database, it becomes an identifier, a member of the set $\mathbf{F}$, e.g., $Jacob \in \mathbf{F}$.

When a token is to be admitted to database at the first time and it is not intended to be declared as a constant, a type, or an instance, e.g., *create PNC account* where *PNC* has already been admitted to database as an instance of type *bank* and *account* has not been in database yet, it is automatically categorized as an identifier.

We further use $\mathbf{V}$ to denote a set of variables. We use $\mathbf{C}$ to denote the union of the 2 classes of constants we discuss earlier, i.e., $\mathbf{C} = \mathbf{C}^1 \cup \mathbf{C}^2$. Now, we are ready to define Froglingo by starting from the most general form: a set $\mathbf{E}^2$, to serve as potential Froglingo assignees in a database:

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

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

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

where each variable is a ground token preceded with \$ and the ground token can be repeatedly used across a database because a variable is local to a preceding term, i.e., when $m\ n \in \mathbf{E}^2$, m cannot be a variable but n can be. When an application $m\ n \in \mathbf{E}^2$, so are its sub terms m and n. We also call m a left most sub term.

When a variable $\$x$ appears in a term $m \in \mathbf{E}^2$, it can 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*, and each binary expression has mathematical binary operators like '+' as well as EP's transitive relational operators like {=+ and operands from $\mathbf{E}^3$ to be defined next. For example, we may define $tax\ \$x : [\$x \leq 200000] := (0.3 * \$x)$; $tax\ \$y : [\$y > 200000] := (0.4 * \$x)$;

When we allow types to be in $\mathbf{E}^2$, it is not only to constrain variables with the types, but more to allow the types to reference entities, such as "this boy" is actually reference a specific instance of the *boy* type. The set $\mathbf{C}^2$ is empty when a database is initiated but it keeps growing when the database grows (see the database definition later in detail). The set $\mathbf{C}^1$ is a fixed set of objects, e.g., *integer* and one of its members 23, no matter they are in a database or not.

Only a term $m \in \mathbf{E}^2$ can appear in a Froglingo database as an assignee. When a term $m \in \mathbf{E}^2$ and later $m \in D$, a subterm $n \in m$ is also said in m and D, i.e., $n \in m$ and $n \in D$ respectively.

To introduce a Fraglingo assignment, we need to define another set $\mathbf{E}^3$ to serve as assigners:

Definition 3 *The Froglingo terms* $\mathbf{E}^3$ *to be assigners in a database:*

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

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

where a variable can be a leftmost subterm of a $\mathbf{E}^3$ term, which is not allowed in $\mathbf{E}^2$.

$\mathbf{E}^3$ includes all possible terms in Froglingo and are potential assigners in a Froglingo database. To simplify the discussion, an assigner with a sequence of expressions (as an example was given in Section 3) is not considered in this paper.

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 and variables in m, i.e., $FV(m) = \{v : v \in m \text{ or } v \in \mathbf{T} \cup \mathbf{V}\}$, where $v \in m$ denotes that v is a subterm in m.

Definition 4 *A Froglingo assignment is in the form of $m := n$, where $m \in \mathbf{E}^2$, $n \in \mathbf{E}^3$, $FV(n) \subseteq FV(m)$. We denote all such assignments as $\mathbf{A}$, i.e., $m := n \in \mathbf{A}$.*

Definition 5 *Froglingo database: A database, denoted as D, consists of a finite set of terms $m \in \mathbf{E}^2$ and a finite set of Froglingo assignments $m := n \in \mathbf{A}$, such that each element is entered in the following methods:*

1. *given $m \in \mathbf{G}$ and the command:*
   - (a) *schema m; or*
   - (b) *schema m isA n; where $n \in \mathbf{T}$,*

   then $m \in D$ and $m \in \mathbf{T}$.

2. *given $m \in \mathbf{E}^2$ and the command:*
   - (a) *create m;, then $m \in D$*
   - (b) *create m isA n; where $n \in \mathbf{T}$, then $m \in D$ and m has the type n.*

   Further,

   - (a) *if $c\ v \in m$ ($c\ v$ is a subterm of m), where $v \in \mathbf{V}$, then we categorize $c \in \mathbf{C}$ if c was not in D before, or we recategorize $c \in \mathbf{C}$ when it had been categorized as an identifier before.*
   - (b) *if $n \in m$ and $n \in \mathbf{T}$, then n remains to have its own type.*
   - (c) *otherwise, for all rest $n \in m$ and $n \in \mathbf{G}$, we categorize n to be an identifier, i.e., $n \in m$ and $n \in \mathbf{F}$*

3. *given an assignment $m := n \in \mathbf{A}$:*
   - (a) *inventory m according the steps 1 and 2 above.*
   - (b) *simply inventory each subterm e sequentially in the order it appears in n and keep all necessary parentheses (and) as they are.*
   - (c) *ensure $FV(n) \subseteq FV(m)$. Otherwise, abort the data entering process.*
   - (d) *for any sub term $q \in n$ and $q \in \mathbf{G}$, ensure q has already been in D, i.e., $q \in D$, by the time q is to be inventoried as part of the assigner in D. Otherwise, abort the data entering process.*
   - (e) *finally, committee the request to have $m := n \in D$.*

In Definition 5.2.(a), we introduced user defined constants. A ground token can be turned to be a constant such as fac and tax that we discussed earlier. An application also can be a constant, such as *bank account* is a constant when we have *bank account* $\$x$: [$\$x$ *isA person*]. We allow a constant to have two variables such as *multi* has two variables when *multi integer* $\$y$: [$\$y$ *isA integer*] := (*integer* $*$ $\$y$), which is equivalent to *multi* $\$x$ $\$y$:= ($\$x * \$y$). We also allow a non-variable term to follow a constant, e.g., *person* (*bE is*) $\$p$: [$\$p$ *isA person*] S *mother* := *person* (*verB mother*) $\$p$, where S is a non-variable identifier following a variable. (*mother* in the assignee is not a variable thought it is a type. In this context, *mother* is just a token and doesn't play any role in the meaning of the sentence because the assigner doesn't reference the *mother* in the assignee while it has *verB mother* which is meant differently.)

To simply our discussion, we assume that the NLP system to be built only receives queries about entity instances after completing the learning period, where NL querying utterance often use types referencing instances such as "the man" in addition to "John". Now, we need to define a set of terms allowed to be inputs in corresponding to NL querying utterance:

Definition 6 *Terms in a correspondence to NL querying utterance, $\mathbf{E}^4$:*

$$m \in \mathbf{F} \cup \mathbf{T} \cup \mathbf{C} \implies m \in \mathbf{E}^4$$

$$m, n \in \mathbf{E}^4 \implies (m\ n) \in \mathbf{E}^4$$

Definition 7 *Terms without types, $\mathbf{E}^5$:*

$$m \in \mathbf{F} \cup \mathbf{C} \implies m \in \mathbf{E}^5$$

$$m, n \in \mathbf{E}^5 \implies (m\ n) \in \mathbf{E}^5$$

Definition 8 *Froglingo Normal Form - terms $NF(D)$ to be reduced to: A term n is an normal form if and only if*

1. *n is null, i.e, $n \equiv null$, where null is a special identifier in $\mathbf{F}$, or*
2. *$n \in \mathbf{E}^5$, $n \in \mathbf{E}^2$, $n \in D$, and for any q such that $n := q \notin D$.*

It's clear that the set $NF(D)$ is finite provided D consists of a finite number of terms in $\mathbf{E}^2$ and a finite number of assignments.

Given any term $m \in \mathbf{E}^4$ under a database D, 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 $c\ v \in \mathbf{E}^2$ where c is either a ground token or an application without a variable in it and v is a variable, we say a term $c\ a \in \mathbf{E}^4$ matches $c\ v$ if a meets v's constraint when v is defined with a boolean expression as the constraint. A term $c\ a \in \mathbf{E}^4$ always matches $c\ v$ if v is type free, i.e., not defined with any constraint. When $c\ a$ matches $c\ v$, we use $(c\ v)\ \{a/v\}$ to denote the state of the match and the new term having the instances 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 $c\ a\ b \in \mathbf{E}^4$ 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 refering to the matched term and ENV.subs referring 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 $m \in \mathbf{E}^4$ is identical to a term $q \in D$ and $q \in \mathbf{E}^2$, i.e., $n \equiv q$ and there are no variables in n, we still need ENV to memorize q about where the reduction process has gone so far, i.e., $ENV = q\{\}$.

Definition 9 *(Froglingo reduction) Given a database D and a term $m \in \mathbf{E}^4$ with a context (discourse) where coreference information such as an entity vs. its type (and its type's ancestors) is available, we have the following Froglingo reduction rules:*

1. *$m \in \mathbf{C} \implies m \Rightarrow m$.*
2. *$m \in D, m \in F \implies m \Rightarrow m$.*
3. *$m \notin D, m \in F \implies m \Rightarrow null$.*
4. *when $m \in \mathbf{T}$, search the context to find an entity e that has m as its immediate type or an ancestor type. If there is not such entity identified, concludes the given m query is disconnected from the existing context and return $m \Rightarrow null$. Otherwise, $m \Rightarrow e$.*
5. *In the case of $m \equiv a\ b$. By induction, we assume a has been matched to a term $a' \in D$ and $a' \in \mathbf{E}^2$, 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 $b' \in \mathbf{E}^2$, where b' is memorized in another environment ENV2. If ENV2 $\notin NF(D)$, we conclude $m \Rightarrow null$. Otherwise,*
   - *(a) $\exists\ p \in NF(D)$ such that $p \equiv ENV2.term$ and $a'\ p \in D$ and $a'\ p \in \mathbf{E}^2$, then we successfully found a match and memorize ENV3 = $(a'\ p)\ \{ENV1.subs\}$, and return $m \Rightarrow ENV3$.*
   - *(b) if there is a variable v such that $a'\ v \in D$ and $a'\ v \in \mathbf{E}^2$, and further ENV2 meets v's constraint, then we successfully found a match and memorize ENV3 = $(a'\ v)\ \{ENV1.subs \cup ENV2.subs\}$. Return $m \Rightarrow ENV3$.*
   - *(c) if there is no variable v such that ENV2 meets v's constraint, then we assign ENV3 = null and return $m \Rightarrow null$.*
6. *When $m := n \in D$, we evaluate m by following steps above and obtain a return environment ENV. Then we substitute the local variables and the global types appearing in n with the values memorized in ENV, i.e., we obtain $n\{ENV.subs\}$. Then we initiate $ENV = null$ for $n\{ENV.subs\}$, evaluate it by following the steps 1 to 5 above and obtain another environment ENV2. If $ENV2.term \in NF(D)$ (then $ENV2.subs$ should be null), then return $m \Rightarrow ENV2.term$. Otherwise, $m \Rightarrow null$.*

We use the symbol $\rightarrow_F$ to denote a multi-step reduction based on $\Rightarrow$, e.g.,$m \rightarrow_F m_n$ when we have $m \Rightarrow m_0$, $m_0 \Rightarrow m_1$, ..., and $m_{n-1} \Rightarrow m_n$.

The reduction defined above forces a term $m \in \mathbf{E}^4$ under a database D to be mapped to a normal form in NF(D). Therefore a database represent a (partial) bounded function though not all terms $m \in \mathbf{E}^4$ are guaranteed to have a normal form.

Lemma 1 *A database D in Froglingo, when the built-in constants $\mathbf{C^1}$ are not considered, represents bounded functions (while not all terms in $\mathbf{E}^4$ are guaranteed to have a terminated reduction for a normal form).*

Proof Definition 9 defines a reduction process that reduces a term in $\mathbf{E}^4$ to a normal form in $NF(D)$ for a given database D, where the size of $NF(D)$ is finite. $\square$

B.1 Type-free Froglingo

When all the variables $v \in D$ are not constrained by a boolean conditional expression, which are also meant the elements in $\mathbf{C}^1$ are type free, we call such a Froglingo system defined earlier is type free, denoted as $\mathcal{F}^0$.

$\mathcal{F}^0$ can fully express the closed lambda terms Λ^0 [Barendregt(1984)] syntactically. For each closed lambda terms $M \in \Lambda^0$ in the form of $\lambda\ x.N$, there is a mapping function $\mathbf{g}$ such that

$\mathbf{g}(\lambda\ x_1.M) = m_1$, where $m_1\ \$x_1 := \mathbf{g}(N\{\$x_1/x_1\}) \in \mathbf{A}$

$\mathbf{g}(\$x) = \$x$

$\mathbf{g}(M\ N) = \mathbf{g}(M)\ \mathbf{g}(M)$

During the conversion, each λ is replaced with a unique identifier in a correspondence to the variable that is converted to a unique variable in Frogling with a \$ preceded. For example the lambda term $\lambda\ x.(x\ x)$ is converted to the Froglingo expression: $m\ \$x := \$x\ \$x$. The β-reduction in the lambda calculus is equivalent to the assignment := and a reduction step that is only in a correspondence to := in Definition 9 in Froglingo. For example when $\Omega \equiv (\lambda\ x.(x\ x))\ (\lambda\ x.(x\ x)) \rightarrow_\beta (\lambda\ x.(x\ x))\ (\lambda\ x.(x\ x))$, $m\ m\ \Rightarrow\ m\ m$.

Therefore, Froglingo $\mathcal{F}^0$ (excluding EP) and the lambda calculus would be isomorphic if Froglingo didn't take the normal form definition in Definition 8 and the reduction strategy in Definition 9, but took the lambda calculus normal form, head normal form, and weak head normal form definitions.

Nevertheless, not all terms in $\mathbf{E}^4$ can be effectively reduced to a normal form in NF(D):

Lemma 2 *The reduction strategy of $\mathcal{F}^0$, as defined in Definition 9, is not strongly normalizing, i.e., there is some $m \in \mathbf{E}^4$ such that m cannot be $F-reduced$ to a normal form $n \in NF(D)$.*

Proof The reduction on $m\ m$ doesn't terminate, where $m\ \$x := \$x\ \$x$. $\square$

The syntactical form $\mathbf{E}^2$ of $\mathcal{F}^0$ is not fully in the form of EP terms $\mathbf{E}$ (the one given in Section 2, and alternatively given in Definition 1) of EP. To declare that EP can carry the syntactical form of Turing-complete expressions, we simply introduce the additional symbols from constants $\mathbf{C}^1$, types $\mathbf{T}$, and variables $\mathbf{V}$ to EP such that EP has the syntactical forms of $\mathbf{E}^2$ and $\mathbf{E}^3$. At the same time, EP is added with additional reduction rules:

Definition 10 *Additional reduction rules for EP when we say EP carries Turing complete expressions:*

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

The additional reduction rules in conjuction with the $F-reduction$ in Definition 9 is applicable and equivalent to EP's $D-$reduction given in Section 2 (and also given in [Xu(2017)]). Then we say:

Proposition 1 *Froglingo expressions are in the form of EP expressions.*

Proof Taking the database definition (Definition 5), the normal form definition (Definition 8 which is exactly the same normal form definition for EP) of Froglingo, and taking an EP expressions ($\mathbf{E}^4$ without types), the reduction rules (Definition 9) for Froglingo and the further reduction rules (Definition 10) are equivalent to EP's $D-reduction$ rules (given in Section 2 and formally in [Xu(2017)]). Note that the Froglingo assigner definition $\mathbf{E}^3$ allows more expressions than the assigner definition for EP does. However, the reduction results from the two assigner definitions are the same. $\square$

No matter what constraints are applied to variables, the Froglingo database definition (Definition 5) contains the definition for EP (as given in Section 2, and formally in [Xu(2017)]):

Proposition 2 *EP is included within Froglingo regardless Frogling is typed or type-free.*

Proof It is clear from Definition 2 to 9 as $\mathbf{E}^1$ is contained in $\mathbf{E}^2$. $\square$

B.2 Typed Froglingo

When all or some variables defined in $\mathbf{E}^2$ are assigned with boolean conditional expressions as constraints, we call Froglingo typed, denoted as $\mathcal{F}^1$ (note we can say: $\mathcal{F}^1 - EP - \mathbf{C}^1 = \mathbf{C}^2$, where $\mathcal{F}^0$ can be in the place of $\mathcal{F}^1$ as well). The typed Froglingo may not have to guarantee every term in $\mathbf{E}^4$ to be effectively reduced to a normal form, but it provides a tool to facilitate the development of Froglingo expressions that will always halt in execution. In this sense, Froglingo is a programming language with a type system. Note that when we call $\mathcal{F}^1$ typed, it should be different from the typed lambda calculus [Barendregt(1984)] that guarantees a lambda expression in the typed lambda calculus to be effectively reduced to a normal form.

In Lemma 1, we said that Froglingo $\mathcal{F}^1$ without $\mathbf{C^1}$ represents bounded functions. It doesn't meant we have a learning algorithm to construct Froglingo expressions. Instead, we say that Froglingo expressions are target concepts for a learning algorithm to approximate and converge to and what the algorithm constructs are EP expressions in an EP database.

Theorem 1 *A Froglingo database is PAC learnable provided it doesn't including constants that take built-in types as variables that have infinite domain, i.e.,* $\mathcal{F}^1 - \mathbf{C}^1$ *is PAC learnable.*

Proof Without the constants $\mathbf{C}^1$, $\mathcal{F}^1$ only produces bounded functions according to Lemma 1. A class of bounded functions is PAC learnable according to the result from [Xu(2025)]. Therefore, we say a class of Frogingo databases, or informally saying a Froglingo database, without $\mathbf{C}^1$ is PAC learnable. □

Appendix C - Parsing and Mapping templates

When we says a Froglingo database without $\mathbf{C}^1$ is learnable in Theorem 1, we implies that it may not be efficiently PAC learnable. This is because the size of a Froglingo expression can be as long as a user like to. For examle, we may have an expression like f $\$x_1$: [...] $\$x_2$: [...] ... $\$x_k$: [...], where $k \geq 1$. Even we have a finite number of entities in a database, say n entities, the constant f would have a instance space of n^k, which grows exponentially in the size of the expression (translated to an exponential logarithm of the cardinality of the class of such Froglingo expressions), causing a database not efficiently PAC learnable [Xu(2025)]. An analogy would be a vector of words in statistical machine learning, where each element of the vector can be a word selected from a vocabulary.

In Froglingo, we break down utterance into smaller pieces: a complex sentence is broken down to individual simple sentences. A modifier such as adjective, adverb, and preposition phrase is converted to a simple sentence that is related to an object that is modified. For example, "John helped a old man who was sick" would be parsed to a structure: *John* (*verB helped*) (((*articlE a*) (*adJ old*) *man*) (*whO who* (*bE was*) (*adJ sick*))). It would be initially saved in database but eventually broken down with *John* (*verB helped*) *p*0010, *p*0010 (*bE was*) *old*, *p*0010 (*bE was*) *sick*, where *p*0010 is an identifier for the man who was sick. Although a longer structure than the standard simple sentence of the "subject-verb-object" structure may be necessary sometimes, the majority of knowledge stored in database are stored in the standard simple sentence structure form. The shortened sentence structure form is applied to parsing templates too.

Instances of parsed sentences and parsing templates with the shortened sentence structure take less database space for the purpose of NLP. In terms of the PAC learning theory, the learned EP database and the parsing templates that are inventoried with less memory demonstrate more powerful learnability. An analogy is that an application embedded into an Euclidean space with a less dimensionality is easier to learn than an application that needs more dimensionality when being embedded to an Euclidean space.

Formally, we assume a parsing template has a fixed size in average, e.g., two: one for subject and the other for object while verb is always filled with a verb instance. In other words, a parsing template has a typical structure like *person* (*verB help*) $\$x_2$: [...], and the instance space of a template is n^2.

Lemma 3 *Parsing templates (and the corresponding mapping teamepltes, i.e., parsing templates are assignees and mapping templates are assigners) are efficiently PAC learnable.*

Proof Parsing and mapping templages are Froglingo assignments which are PAC learnable according to Theorem 1. Because the term size of a parsing template is constant, the size of its instance space, translated to the logarithm of the cardinality of the class of such parsing templates, is in polynomial. This determines that parsing and mapping templates are efficiently PAC learnable [Natarajan(1989), Kearns and Vazirani(1994), Xu(2025)] □

Theorem 2 *Provided the assumption that knowledge is isomorphic to a bounded function, NL utterance are efficiently PAC learnable.*

Proof EP databases are PAC learnable and some EP database are efficiently PAC learnable [Xu(2025)]. Lemma 3 says parsing templates are efficient PAC learnable, therefore, the EP expressions in an EP databases, or simply say EP databases, that are guided by and converging to the parsing templates must be efficient PAC learnable. Because of the assumption that knowledge is isomorphic to a bounded function, we conclude that knowledge (or equivalently NL utterance) is efficiently PAC learnable. □

Appendix D - Organizing knowledge into the space of a bounded function

Theoretically, we have the following conclusion immediately:

Theorem 3 *Provided that knowledge is isomorphic to a bounded function, all the objects in knowledge can be inventoried in an EP database and the relationships among the objects can be calculated.*

Proof Because of the assumption that knowledge is isomorphic to a bounded function and because an EP database represents a bounded function, we immediately have: knowledge can be organized into the space of a bounded function, i.e., the objects in knowledge can be inventoried in an EP database and the relationships among the objects can be calculated by the EP reduction rules. □

This theorem says that there is a mapping from NL utterance to an EP database. In the reality, however, this mapping process will be tedious because each word, particularly those frequently used words, behaves differently and needs to be modeled differently. In Section 7, we discussed, as an example, how the word "sport", defined as a type *sport*, penetrates into relevant utterance while it means a significant portion of our daily life but its counterpart in Froglingo has to play a "low profile" role. When we say "sport" plays a "low profile" role in the machine readable form of Froglingo, we meant that it appears to have very few sub types and almost "no instances", which is contrasting to the type *person* that is rich in the numbers of its sub types and instances. When the type *person* is defined, it also has a lot of pre-existing information such as a person can talk, has a body, and etc.. When the type *sport* is defined, can we think of any pre-existing information we can attach to it in Froglingo (but not in NL) and is there any instances of sport? The answer appears to be no. Thinking of it from a different angle, however, it really doesn't matter because one can learn the meaning of "sport", though itself not being defined with any information, through *soccer* that is defined as a subtype of *sport*, through "John played soccer yesterday", "Jenny run in a New York Marathon", and etc.. As a matter of fact, *sport* is rich too, though itself not being defined with any information, because *soccer* is its sub type, John and Jen's actions are its instances, and etc.. We perceive *sport* differently from *person* most likely because we take the world entities as the information ground and abstract words like "sport" are the information not on the ground.

Appendix E - A user interface supporting both Froglingo expressions and natural language

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.

In the discussion below, the operator *isA* is used indiscriminately between a type (created by *schema* and an instance (created by *create*), but the differences are implied by English words, such as car (a type) vs. Joe (an instance).

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 *p*0002, 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 ≤ t0002 start ≤ 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 *f*0001, 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 *f*0001 and created *c*0001 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) No-time));*	`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: [îsA 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	

References

[Barendregt(1984)] H. P. Barendregt. 1984. *The Lambda Calculus - its Syntax and Semantics.* North-Holland.

[Brown et al.(2020)] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. *Language Models are Few-Shot Learners.* NeurIPS

2020.

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

[Jiang et al.(2023)] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao. 2023. *Self-planning Code Generation with Large Language Models*. ACMTrans. Softw. Eng. Methodol., Vol. 1, No. 1, Article . Publication date: October 2023.

[Kearns and Vazirani(1994)] M.J. Kearns and U.V. Vazirani. 1994. *An introduction to computational learning theory*. The MIT Press.

[Kleinberg and Mullainathan(2024)] J. Kleinberg and S. Mullainathan. 2024. *Language generation in the limit*. NeurIPS 2024, arXiv:2404.06757.

[Li et al.(2025)] J. Li, V. Raman, and A. Tewari. 2025. *Generation through the lens of learning theory*. 38th Annual Conference on Learning Theory (COLT 2025).

[Littlestone(1987)] N. Littlestone. 1987. *Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm*. Machine Learning, 2:285–318, 1987.

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

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

[Xu(2017)] 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.

[Xu(2024)] K. Xu. 2024. *Outline of a PAC Learnable Class of Bounded Functions Including Graphs*. The 7th International Conference on Machine Learning and Intelligent Systems (MLIS 2014).

[Xu(2025)] 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.