A Deductive System based on Froglingo for Natural Language Processing

Kevin H. Xu

kevin@froglingo.com

***Abstract* —** A word, a prepositional phrase, and a sentence can have different meanings in different contexts. Machine learning based Natural Language Processing (NLP) technologies build contexts for disambiguation using datasets from different disciplines such as medicine vs. finance. This approach , however, struggles to eliminate ambiguity at the degree human beings can, as we possess more granular knowledge of the context for a given circumstance. In other words, this approach is inductive (or transductive), i.e., predictive and the predictions based on training data are not guaranteed to be valid. As a result, no NLP system yet developed is able to interact with humans as if it was itself a person.

Before the development of machine learning (statistics-based) approaches, NLP was studied via deductive (symbolic) approaches. However, the tools available at the time were not capable of uniformly representing the subjects of natural language communication. For this reason, deductive approaches during the 1950s through 1980s failed to produce a promising solution for NLP.

In this paper, we introduce a fundamentally deductive system for NLP. It has been implemented with a set of preliminary features using Froglingo, a unified programming language and database management system. It pre-defines a set of representations which approximates things, i.e., entities and actions, in the real world. It collects abstractions embedded in things and in linguistic structures to support general reasoning. It tolerates illegitimate natural language expressions because it references things represented in a database. It interacts with clients to clarify ambiguous sentences because of its deductive approach to parsing. It generates natural language expressions that allow friendly human-machine interactions. It also generates code in natural languages, allowing its database to expand within and across disciplines.

***Keywords*—** Linguistic structure, semantic representation, abstractions, parsing, natural language understanding, natural language & code generation, lambda calculus, (weak) head normal form, reduction, similarity for disambiguation, part of speech, word sense, coreference, discourse, homograph, fixed phrases, idioms.

I. Introduction

Natural language processing (NLP) has reached our daily lives in areas such as information searching, natural language translation, and human-machine interactions. Advances in this field have been driven by machine learning (and deep learning). Machine learning algorithms use training datasets to make predictions or decisions without an explicit programming of linguistic structure and context. However, this limits NLP performance, as machine learning algorithms lack the contextual knowledge that a human would have. Specifically, the machine learning approach is inductive or tranductive, i.e., decisions based on training datasets do not always reflect client needs that are conveyed by natural language expressions. In fact, no machine learning system has been designed to precisely represent (approximate) real-world entities and actions. While one branch of NLP machine learning research promotes pure neural language models without any linguistic structure (and context) [7], the other aims to simulate a better "understanding" of natural languages via embedding linguistic structures [11], [12], [17], [19]. However, these linguistic structures are syntactical forms and do not necessarily represent things in the real world. Further, this branch continues to take neural language models (machine leaning) as its core basis. Because machine learning is a non-deductive approach, it faces stiff challenges to advance NLP to a new level [17], [18], [19].

Another direction toward NLP, called symbolic (deductive) approaches, involves explicitly programming linguistic structure and context (things in the real world). This proved overwhelming during the time period from the 1950s to the early 1990s [16]. In addition to limited computing power in CPU speed and memory space, limited programming language tools and methodologies stalled the development of a robust NLP system. The essential challenge was how to represent all kinds of things uniformly in a computer, i.e., via a programming language, which was the first step in developing a NLP system [13]. One option, called “scruffy” in the field of artificial intelligence, was to use “procedural” programming languages for “fine-tunning” programs. This approach however could not be scaled up, i.e., it was too scruffy to be extended. The other option, called “neat”, was to use mathematically sound methods, i.e., “declarative” or “logic” programming languages like Prolog. It was hoped that this choice, with a single formal paradigm, would be general and extendable. Although a few symbolic NLP systems, including the first one ELIZA in 1966 [20], had been developed, none of them demonstrated a promising solution for a general NLP application. When there were not having a clear direction in declarative approaches, more effective algorithms from statistical and machine learning approaches toward NLP were developed during the early 1990s ([4], [10]). These supplanted earlier, deduction-based approaches toward NLP.

Froglingo is a system that unifies programming language and database management [24], [25]. It has a sub language, called the Enterprise-Participant (EP) data model which doesn't need to have variables which is essential in programming languages. Like traditional database management systems, the EP data model manages business data with finite information. However, it is more expressive than the traditional database management systems because it also manages bounded functions with infinite information (a proper subset of the partial recursive functions). Mathematically, it is as expressive as Turing machines given infinite time and space. Equivalently, we say that the EP data model itself can be a sole tool to develop software applications without a programming language, although in a very inefficient way [21], [22]. For practical efficiency in software development, we add variables on the top of the EP database. The combination of these two is called the Froglingo system, which allows uniformly databasing of both business data and business logic, i.e., the semantics represented by programs and databases. Since database systems (like relational databases) are more productive than programming languages in managing information, as we have observed through practices, Froglingo is more productive than both programming languages and relational database systems. We believe that Froglingo's uniformly higher productivity over business data and logic combines the "single formal paradigm" of the neat approaches with the "fine-tunning" of the scruffy approaches.

In other words, the EP data model is interpreted by an extended lambda calculus, where a database is expressed by a finite set of lambda terms (possibly with assignments), simply abbreviated as 'terms' in this paper, which have unique weak head normal forms and can be effectively reduced to their weak head normal forms [21]. A database approximates a finite set of terms having head normal forms. When a database collects more data, it more closely represents the approximated head normal forms. The union of all such databases together is mathematically the union of all lambda terms with head normal forms, i.e., semantically equivalent to Turing machines.

With the limitations of inductive approaches and with the more productive Froglingo system becoming available, we revisit the feasibility of a deductive approach to NLP. This time, we use terms in Froglingo to uniformly represent a finite set of entities and actions. Because of the high expressiveness of terms, all kinds of syntactical forms and semantic meanings in natural languages, such as homograph, word sense, coreference, and idiom are naturally fit into the expressions in terms. Since a term has a unique (weak head) normal form, we calculate the unique meaning of a sentence by reducing the term expressing the sentence. Because of the deductive nature of the Froglingo, the NLP system can detect ambiguities. Because of the high productivity in Froglingo, the amount of software engineering work, which was one of the major concerns to the machine learning approaches, can be significantly reduced.

Entities and actions are discussed in Section II and III respectively. During the discussion, we show by examples the intended English phrases and sentences that will be mapped to corresponding entity and action representations in the database.

When a thing is uniquely distinguished by itself in the world, it may contain other things semantically. In Section IV, we give a representation of the abstractions embedded within things.

Natural languages use invariant linguistic structure to convey the semantics of things. Therefore, we say that human beings utilize abstractions within linguistic structures to understand more about things without knowing much about them. Multiple sentences or phrases can be equalized no matter what things are conveyed. In Section V, we represent such equivalent abstractions in the NLP system.

In addition to the syntactic expressions introduced in these sections to represent things, there are other syntactic expressions that facilitate and enrich the communications about things using natural languages. These include prepositional phrases (p-phrases), which may not necessarily be part of the representations discussed in sections II to V, and conjunctions. We discuss how the NLP system represents these syntactical forms in Section VI.

With all possible syntactical forms considered, we describe the parser of the NLP system in Section VII, which breaks client input streams into pieces for the representations discussed above. In this section, we also show that the parser is deductive, and therefore the NLP system is deductive. In Section VIII, we define a state of parsing at which the NLP system claims that it "understands" an input stream.

Communications between people are always "friendly", a concept introduced from research in human-machine interactions. People are able to convey a single topic in various ways via different words, phrases, and sentences with different verb moods, auxiliary verbs, and context. People may have completely different responses to a request or an inquiry depending on their personality and their psychological state at a given time. In Section IX, we use extensive examples to demonstrate that the NLP system has been implemented with a few initial features to mimic people's certain behaviors in communication. Code generation is part of natural language generation. To show the difference between the NLP system and the contemporary programming languages, however, we discuss this topic separately in Section X.

II. Entities

A thing, i.e., an entity or an action in the world, cannot be completely represented in a computer. It can, however, be approximated. In this section we discuss the representation of entities; the representation of actions will be covered in the next section. The process of representing an entity starts with a single identifier, a term, in Froglingo to be created in a database. For example, we add the entities *joe, walmart, pnc* to a database via the built-in operator *create*:

create joe;
create walmart;
create pnc;

Noun phrases are intended to be mapped to these terms. A few potential examples include *Joe*, *the old who sites on a*

bench; *walmart*, *the store at the corner*; *the pnc bank*; and *the bank I have an account with*. These phrases can be parsed and identified from client input stream by the parser.

Note that as part of the current design, all text entered by clients will be converted to lower case in the NLP system. This allows the system to use identifiers with capital letters as built-in tokens for the purpose of internal system processes. For example, *Inherit* and *Belong* (to be discussed below) are two such tokens, which will be distinguished from *inherit* and *belong*, converted from client input in the same database.

To provide abstractions, i.e., semantic containment relationships among entities, we introduce types in a database. Similar to a class in object-oriented programming languages, a type is a profile that define a set of common attributes and actions (to be discussed in the next section) that its instances, as entities, share. Here we give a few sample types that will be discussed through the paper. Types are introduced by the built-in operator *schema* (a new operator not discussed in the earlier papers regarding Froglingo):

```
schema thing;
schema entity Inherit thing;
schema organization Inherit entity;
schema bank Inherit organization;
schema store Inherit organization;
schema action Inherit thing;
schema person Inherit organization;
schema male Inherit person;
schema adult Inherit person;
schema commodity Inherit entity;
schema shirt Inherit commodity;
```

The built-in token *Inherit* indicates that the type at the left inherits the properties of the type at the right. (Note that unlike with classes in object-oriented programming languages, the inheritance here is looser, i.e., we allow inherited types to add independently and over-write attributes and actions.)

Now, we can bind entities to types upon a need from use cases. For example, we can bind *pnc* and *walmart* to *bank* and *store* respectively using the built-in token *Belong*:

```
create pnc Belong bank;
create walmart Belong store;
create joe Belong male;
create joe belong adult;
```

We also allow a type to have multiple inheritances and an entity to have multiple belongings, as we have seen with *male*, *adult*, and *joe*.

We can define additional attributes for previously defined types. For example:

```
schema person weight = weight;
schema bank account organization balance = currency;
schema store inventory commodity number = integer;
schema store inventory commodity price = currency;
```

In the definitions above, we introduced additional types *weight*, *number*, and *integer*, where *integer* and *number* are built-in types in Froglingo with the property of *integer Inherit number*. The type *weight* is defined explicitly. Here are the terms to add it along with a few other types, which will facilitate our discussion throughout the paper:

```
schema weight Inherit number;
schema currency Inherit number;
schema attime Inherit number;
schema intime Inherited number;
```

The type *weight* is for the weight of a physical object. For example, the NLP system supports the English phrases *80 lbs* and *40 kgs*. The type *currency* is for money in financial transactions, for example *80 lbs* and *50 dollars*. The type *attime* is for a point of time, e.g., *4:00 pm*, *on Sunday, on 01/21/2021*. The type *intime* is for a period of time, for example *3 days*, *52 seconds*, *23 years*. In the examples, we see that *80 lbs* could refer to either *weight* or *currency*. Given a sentence and its context, the NLP system parser determines if *80 lbs* is for *weight* or *currency*.

To fully support the numeric types discussed here, along with other potential numeric types in the future, the NLP parser was implemented to support expressions having a token *Ow* which allows various numeric types to be defined upon use case needs. A few sample expressions with *Ow* are given in the footnote[1].

With types introduced, we are now ready to add properties for entities by following their types' profiles. Here are a few examples:

```
create joe weight = 180;
create joe inventory shirt number = 4;
create pnc account joe balance = 90000;
create pnc account walmart balance = 1000000;
create walmart inventory shirt number = 4000;
create walmart inventory shirt price = 100;
```

With the sample properties above for *joe*, *pnc*, and *walmart*, the system will use its parser to accept English phrases like *Joe's weight*, *my account balance in PNC*, and *shirt's price at Walmart*. Note that the representation of an entity property, such as *joe inventory*, is also called an entity in this paper.

We have shown abstractions established in the NLP system among types, entities, and entity properties. For example, *joe* is the abstraction of the collection of all Joe's properties, such as *joe weight* and *joe inventory shirt number*. We also have shown that different English phrases may reference a single entity, e.g., *pnc*, *the pnc bank*, and *the bank that I have an account with*.

[1] The expressions below are entered via the built-in operator *create*:

```
Qw $x:[$x isa integer] lbs;
Qw $x:[$x isa integer] kgs;
Qw $x: [$x isa integer] hours;
Qw $x:[$x isa integer] lbs $y:[$y isa integer] ounces;
weight (Qw $x lbs) = $x; /* in lb*/
weight (Qw $x kgs) = ($x * 2.2); /* in lbs*/
currency (Qw $x lbs) = ($x * 1.31); /* in dollars */
currency (Qw $x dollars) = $x;
intime (Qw $x hours) = ($x * 3600); /* secs*/
intime (Qw $x hours $y:[$y isa integer] minutes) = ($x * 60 + $y);
```

Fixed phrases represent entities themselves, such as *Mount Everest* and *New York City*. Therefore, we can have *new york city Belong location* when *location* is defined as a schema. We can make two phrases equivalent via an equation: *create new york city = nyc*.

We count mathematic functions as entities as well. Here is an example which will be further used later:

```
create multiplication $n1: [$n1 isa number]
                      $n2: [$n2 isa number]
                    = ($n * $2);
```

Starting from the next section, we will no longer precede expressions to be added into a database with the operators *create* or *schema*.

III. Actions

Things also include actions (or events) that are taken by entities, which have impacts on other entities. The entities discussed in the earlier section can be inventoried in database before they are categorized with types. The actions to be introduced here, however, have to be categorized into types (here called templates) before they can be expressed in the NLP system. A template is a profile mostly for simple English sentences, i.e., a minimum structure the author of the template believes is sufficient to represent a set of actions. Here are a few sample templates in a database:

```
person (Vw cry);
person (Vw provide) $p: [$p isa person] (Prew with)  thing;
person (Vw talk) (Prew with) $p: [$p isa person];
organization (Vw acquire) $o: [$o isa organization];
thing (Vw surprise) person;
```

In a template, the verb is always in its original form, positive, and active. It is always preceded with a special token *Vw* for the convenience of parsing. Similarly, a preposition is always in its original form and preceded with a special token *Prew*. If necessary, articles, pronouns, adjectives, and adverbs can be part of templates as well. They are preceded with *Thtw*, *Prow*, *Ajw*, and *Avw* respectively. The positions of subjects, objects, and prepositional objects are in general filled with types. They can sometimes be filled with noun phrases, noun clauses, infinitive phrases, or gerund phrases for entities or actions, which will be further discussed later in this section.

Types and variables like *$p: [$p isa person]* are placeholders for noun phrases, action nouns, noun clauses, infinitive phrases, or gerund phrases that represent things. In a parallel to global variables in programming languages, types are globally recognized variables in a database. A template is normally started with a type. We can use either a type or a variable in the place of an object in a template. A type is simpler to present and read when it is at the position of an object. A variable is needed at the position of an object when the object is placed with the same type that has been in the place of the subject, such as the type *person* at the position of the subject in the template: *person (Vw provided) $p: [$p isa person] (Prew with) thing*, where the variable *$p* is used for another set of individuals as type *person*. A variable is also needed when we need to narrow the scope of the entities functioning as the object, which cannot be done by a whole type, such as *$p: [$p isa person and person weight > 180]*.

With templates, the parser tries to break down a sentence and match the sentence to one of the templates. Here are a few sample sentences targeted by the templates above:

```
She didn't cry;
He gave her a bouquet of flowers;
Mary is talking with the person behind the door;
This company has been acquired;
Her appearance surprised me;
```

When we use a sentence to communicate a concept, we try to articulate it as clearly and precisely as we can. Similarly, we use templates to represent groups of sentences as closely as linguistic structure allows. For example, we can make the following templates for the commonly used word *take*:

```
person (Vw take) $s1: [$s1 isa commodity];
person (Vw take) $s2: [$s2 isa idea] (Prew (in spite of)) thing;
person (Vw (take $some: [$some isa wear] off));
person (Vw (take off)) days;
airplane (Vw (take off));
business (Vw (take off));
person (Vw take) vacation (Prew in) location;
```

This exercise may be not exhaustive. We can always add templates or modify existing ones when a new use case arises.

Sometimes, we may find certain entities have very unique actions that are not categorized with a type. We can directly build templates for these specific entities. For example, running is not an action everyone takes daily. Therefore, we may have a template *person (Vw run)* for the type *person*, and additionally add another one specific for *joe*:

```
joe (Vw run) (Avw everday);
```

A template can be assigned with a value. For example [2]:

```
bank (Vw transfer) currency
   (Prew from) $frm: [$frm {+ bank account]
   (Prew to) $tow: [$tow {+ bank account]
 = (update $frm balance = ($frm balance - currency)),
   (update $tow balance = ($tow balance + currency))),
   Botape (bank "transfer" currency "from" $frm "to" $tow
    ";");
```

The following sample sentences are intended to be mapped to the template above by the parser:

[2] See the papers [12, 14] for more information about Froglingo built-in operators {+ and isa that appeared in the paper.

PNC transferred 100 dollars from Joe's bank account in PNC to the PNC bank account of Walmart;

Similarly, we can construct templates for sentences that describe entity properties or states instead of actions. Here are a few examples:

number (Arew be) $n: [$n isa number] = (number == $n);
entity (Arew be) $e: [$e isa entity] = (entity == $e);

With the sample database constructed in Section II and for the expressions: *2 is 2; 2 is 3; and Joe is the person whose bank account balance at PNC is 10000 dollars,* the NLP system will have responses like*: yes, no,* and *yes* respectively.

When the auxiliary verb *have* is the main verb of a sentence, it shows the object is a property of the subject. Here is a sample template having *have*:

entity (Hasw have) $e: [$e isa entity] = there_is $x where
(($x (= entity) and ($x {=- $e)) or
(($x (= entity) and ($x == $e)));[3]

With this template and the database constructed in section II, the system will accept the following as sentences: *Joe has a weight, Joe has an account, Joe has 90,000 dollars; PNC has Joe; PNC has 90,000 dollars.*

There are more expressions using *have*. For example, *I have a boss who is grumpy,* and *I have a park where I take a walk every day*. These would have to be added with new templates like: *entity (Hasw have) boss, entity (Hasw have) area*, where *boss* and *area* would have been defined as types.

When the auxiliary verb *do* is the main verb of a sentence, it functions as the abstraction of the actions that may not be explicitly mentioned in the sentence but can be derived from context. For example, when a person asks: *did you have dinner yet?*, another person may answer: *Yes, I did*. The sentence *what can I do for you?* would bring the following contexts together although all pieces are not be explicitly mentioned together in a single sentence: all the actions *I* can take (which are represented in sentences in the NLP system as we will discuss in the later sections) and the action *you* need *me* to take (which is carried by the sentence *you* will provide in an answer). With that said, the semantics of some sentences, like those with *do* as the main verbs, cannot be explicitly expressed because the sentences don't provide a reference to their contexts. Instead, we allow a template to have an assignment with a built-in operator *Context* to be the assigner. For *do*, we may have a template:

person (Dow do) End = Context Do;

Do may be one of many other tokens the operator *Context* takes to perform special tasks for sentences with *Dow do* as the main verb. These special tasks, along with *Context,* can be coded as part of the implementation and maintenance of the NLP system. They also can be expressed in the database by the authors of template using Froglingo (which hasn't been implemented yet as part of the NLP system).

Similarly, we use *Context* to build templates for the pronoun *it* when it is used to express a time:

(Prew it) (Arew be) intime End = Context It_be;
(Prew it) (Arew be) attime End = Context It_be;

When a client enters: *I bought a shirt from Walmart. It has been 3 days*, the NLP system will be able to correlate the two sentences together. (Note that the information about the verb tense and the time value are available to *Context* when it is processing.)

In the three templates provided above, we used the token *End*. Its inclusion in templates is usually optional. However, we need it if one template is a leftmost subterm of the other. For example, we need at least another template in addition to *person (Dow do)*:

person (Dow do) action;

where *action* is the type for action nouns, which has been introduced in Section II and will be discussed further. For example, *I am doing my homework* is a sentence that will be matched with the template. When the template *person (Dow do) action* is defined as a term, a leftmost subterm such as *person (Dow do)* cannot have an assignment in Froglingo. The token *End* helps assign a value to *person (Dow do)*.

There are some verbs such as *like*, *desire*, *want*, *plan*, and *promise* which take persons as subjects and infinitive or gerund phrases as immediate objectives. For example, *Joe likes to purchase shirts*. These sentences are distinctive in that their secondary verbs express the degree of the subjects' desire in performing their actions. The sentence above, for example, can be understood to imply an action (*Joe purchases a shirt*) which is intended by *Joe* and a degree to which *Joe* has made up his mind to perform that action. To represent the degree of peoples' desires in taking actions, we group the two verbs together to express peoples' thinking, desiring, promising, etc. We will collect these kinds of structures in templates such as:

person (Vw (would like)) (Prew to) Infinitive;
person (Vw like) (Prew to) Infinitive;
person (Vw want) (Prew to) Infinitive;
person (Vw plan) (Prew to) Infinitive;
person (Vw promise) (Prew to) Infinitive;
person (Vw confirm) (Prew to) Infinitive;
organization (Vw keep) Gerund;

where the special token *Infinitive* or *Gerund* indicates that the following sentence structure is an infinitive or gerund phrase. Combining two verbs into a single template simplifies semantic processing after parsing.

[3] The expression at the right is read: 1) if there is a term *$x* such that *$e* is the outer most sub term of *$x* and *entity* is a subterm of *$x*, then *entity* has *$e*. If there is a term *$x* such that *$x* can be reduced to *$e*, then *entity* has *$e*.

In a database, we can accumulate as many templates as we need. We assume that we will model only a finite number of types. There is a finite number of verbs and verb phrases. Each sentence contains a finite number of words. These determine that we need a finite set of templates to model all sentences that can potentially be used in our daily lives.

In this section, we discussed templates with assignments that reflect the impacts of actions conveyed by matched sentences. In the coming sections, we introduce special built-in operators *Botran*, *Botalk*, *Botape*, and *Bothink* that use assignments to mani4pulate English sentences. This rule is not about action impacts, but helps natural language generation.

When a client enters a sentence in past tense, the NLP will record all the break-down information about the sentence including its matched template. For the purposes of this paper, however, we will use sentences exactly as entered by clients. The sentence below is an example which will be further discussed in later sections:

joe purchased a shirt from walmart;

IV. Abstractions I

When multiple sentences can be summarized in brief with one sentence, we call the briefing sentence an abstraction, saying it abstracts (in brief) the actions expressed by multiple sentences. In the section above, we gave an example involving money transfer in a bank using the word (*Vw transfer*), which triggers two operations using the built-in operator *update*. The template containing *Vw transfer* is an abstraction. An abstraction can be a component of another abstraction, i.e., calling another abstraction. Here is an example of an abstraction calling the abstraction containing the verb *Vw transfer*:

organization (Vw donate) currency (Prew to)
$o: [$o isa organization]
=
Botran (organization "transfer" currency
"from pnc account" organization "balance to pnc account"
$o "balance."),
Botape ("Thank you" $o "for your work! Sincerely," organization);

Given a client input such as

I am Joe. I would like to donate 1,000 dollars to the American Red Cross Homeless Shelter,

the NLP system will proceed to make a transaction transferring 1,000 dollars to the shelter organization, which also has an account with *pnc*.

In the template above, the token *Botran*, an abbreviation for "bot transfer", is a built-in operator which re-parses the string generated from its parameter (i.e., the term within the parenthesis following *Botran*) within which the variables are substituted with passing values. In the given sample client input, for example, the resulting string would be *Joe transfer 1000 dollars from pnc account Joe balance to pnc account american red cross homeless shelter balance*. When this string is parsed, the template having the verb *Vw transfer* will be matched, and eventually be executed, which further triggers the execution of the built-in operations of *update*. The operator *Botran* itself doesn't perform tasks for the client but acts as a messenger to invoke another template.

Similar to *Botran*, the token *Botape*, an abbreviation meant for "bot tape", will re-parse the string from the following term with its variables substituted. Instead of involing another template, it simply returns a message to clients. In addition to responding to clients, the operator *Botape* tapes (records) this message as a record for future reference, which will be discussed further in Section IX. In the example where Joe donates 1,000 dollars, the output from *Botape* would be *Thank you the American Red Cross Homeless Shelter for your work! Sincerely, Joe*.

When a template is an abstraction and invokes another template, we say that abstraction is transitive. Such abstractions help to relate sentences. For example, if Joe forgot that he donated 1,000 dollars, he might ask: *Why did my bank account balance at pnc became less?* By tracing the structure within the template for donation and by tracing transaction history, the NLP system will be able to respond with: *because you donated 1,000 dollars to the American Red Cross Homeless Shelter*.

So far we have discussed abstractions that are driven from a given task, i.e., templates, to introduce subtasks, i.e., a sequence of values (called assigners in [21], [25]) at the right hand sides of the template assignments. Subtasks are constrained and shaped by given tasks. For example, a donation must imply a money transfer. There are other kinds of abstractions where subtasks don't have any constraints, or where we are not aware of any constraints. In the example *Joe took a vacation in Chicago*, we are unaware of any constraints, as Joe might have done anything he liked on his vacation. For such abstractions, we don't have template assignments available to connect related actions in a database. However, we can record such abstractions. Given subtasks such as: *During the vacation, Joe visited Willis Tower and visited one of his friends*, the NLP system would record the subtasks as assigners of the given action:

joe took a vacation in chicago =
joe visited willis towner,
joe visited one of his friends.

From this example, we can observe that abstractions can be built along with actions when the corresponding templates lack assignments. With the record above for an abstraction, the system would be able to answer questions like: *What did Joe do during his vacation? Did he visit Willis Towner?*

In the record above, the verbs are in past tense to indicate that the actions were not made by the NLP system. When this record is referenced in the future, the NLP system treats it as gossip, i.e., it could be true but is not validated. When actions are made by the NLP system, the actions are recorded in present tense. For the example where Joe made a donation, the actions (the abstraction) would be recorded as:

joe donate 1000 dollars to american red cross homeless shelter
=

pnc transfer 1000 dollars from pnc account joe to pnc account american red cross homeless shelter,
Thank you the american red cross homeless shelter for your work! Sincerely, Joe.

In the NLP system, sentences in present tense indicate that the corresponding actions are verified facts, as the system itself made the actions.

V. Abstractions II

In the previous section, we constructed templates that support abstractions among actions. In this section, we continue to construct templates for the abstractions embedded in linguistic structure. First, action nouns are often used to brief sentences. For example, the following two sentences are related together by the verb *paint* and the noun *innovation*:

Joe painted his house.
His family is very happy with the innovation.

To support these abstractions, we inventory abstractions embedded among action nouns. Continuing the abstraction constructions using *Inherit*, which has been discussed in section II, we can add more examples for action nouns:

```
move Inherit action;
travel Inherit move;
trip Inherit move;
travel Inherit trip;
improvement Inherit action;
innovation Inherit improvement;
repair Inherit innovation;
conversation Inherit action;
talk Inherit conversation;
trade Inherit action;
purchase Inherit trade;
buy Inherit trade;
```

Continuing from the discussion of *Belong* in Section II, we can construct abstractions between verbs and action nouns. Here are a few examples:

```
Vw travel Belong travel;
Vw purchase Belong purchase;
Vw buy Belong buy;
Vw talk Belong talk;
Vw repair Belong innovation;
```

With the constructions above, the system will be able to relate the following sentences in pairs:

Joe repaired his car. The innovation saved him 2,000 dollars.
Mary has been talking with Joe for 3 hours. The conversation is still going on.

Since many verbs, such as *Vw go, Vw take, Vw paint*, have different meanings in different contexts, certain abstractions cannot be built at a phrase level for these verbs and nouns. However, we can build abstractions at the sentence level. We begin doing so by adding more sample types and templates[4], as demonstrated in Sections II & III, for various contexts:

```
location Inherit entity;
person (Vw travel)  (Prew to) location;
person (Vw go) (Prew to)) location =
      Botran (person "travel to" location);
chicago Belong location;
```

Now, the system is able to relate the pair: *I went to Chicago. I enjoyed the trip.*

To relate the earlier sentence pair containing the words *paint* and *innovation*, we add the following template (provided that a profile like *house Inherit entity* has been constructed):

```
person (Vw paint) house Inherit innovation;
```

The last kind of abstractions we will construct in this section is equivalencies between two templates. By doing so, we can define semantics (via assignment) only once for all equivalent sentences.

Many synonymous verbs can be difficult to equate. But equating sentences with synonymous verb phrases is easier because we have subjects, objects, and prepositional phrases to constrain the semantics of the sentences. Here are a few sample assignments that equate pairs of templates using *Botran*:

```
person (Vw go) (Prew to) location =
   Botran (person "travel to" location ";");
person (Vw (walk away from)) $p: [$p isa person] =
   Botran (person "leave" $p ";");
person (Arew be) weight =
   Botran (person "weighs" weight ";");
person (Vw (travel through)) area =
   Botran (person "travel" area ";");
```

There are many verb phrases which are not semantically equal but which are closely related. We can add additional information using prepositional phrases to equate them. Here are a few sample assignments with additional prepositional phrases:

```
person (Vw drive) home
   =
   Botran (person "come home by car;");
organization (Vw get) commodity (Prew by) (Vngw purchase)
   =
   Botran (organization "purchase" commodity ";");
```

[4] These types and templates are simplified for demonstrative purposes and may not be accurate in modeling the real world.

VI. Prepositional Phrases and Conjunctions

We have already introduced templates to process simple clauses. These templates provide the minimal structure which the author believes necessary to capture the actions the targeted sentences are intended to convey. As modifiers, prepositional phrases (p-phrases) can appear anywhere within sentences, but not necessarily within the templates we construct. This is the case for all p-phrases that modify nouns (or noun phrases). In addition, certain p-phrases, such as those for time and space, can be almost universally applied to verbs in any clauses. Take, for example, the phrase *I walked in a park under the sun at 2:00 pm Saturday*. Adding such p-phrases to all templates would be a clumsy and inefficient way to introduce and maintain the templates. To resolve this issue, we introduce additional structures, called pp-relations in this paper, to capture the relationships between p-phrases and noun phrases or verb phrases.

Given a pp-relation, there are three components: preposition (such as *Prew before*), the type of noun phrase that can follow the preposition, and the type of a noun or verb phrase that the p-phrase can modify. Here is a sample pp-relation:

Prew at attime action;

It says that a p-phrase for the type *attime,* like *at 3:00 pm,* can modify any action. Examples might include *the party starts at 3:00 pm*; *Which party are you attending? It is the one at 3:00 pm*. With this pp-relation, the parser will proceed with the consideration that *at attime* is the time *action* takes place (with either past, present, or future tense).

Prew for organization action;

This pp-relation says that a p-phrase like *for her son* can modify an action, such as *she does this for her son*. With this pp-relation, the parser will proceed with the consideration that the purpose of *action* is for *organization*.

Prew at organization entity = select $x where
(((($x {+ organization) and (entity {- entity)));

This pp-relation says that a p-phrase like *at PNC* can modify another entity, such as *Joe's account*. With this pp-relation having an assignment, the parser will validate and derive the resulting entity of the modified entity such as *Joe's account at pnc*. When *Joe's account* is considered as either *Joe account* or *account Joe*, as the way the NLP currently was designed, *pnc account joe* is determined as the resulting entity based on the *select* operation at the right side of the assignment and the provided the database described in Section II, which means that the parser substitutes the originally noun phrase with the resulting entity.

Collecting pp-relations is sometimes more tedious and complicated than we have demonstrated here. For example, we need to deal with entities modified by multiple p-phrases (as we have done but not presented in this paper). We will eventually need to differentiate all usages, in various contexts, for a few commonly used prepositions such as *for*, *of*, and *with*. However, as long as we continue doing this alongside the modeling of new things, we will eventually exhaust all possible pp-relations.

Before ending this section, we will briefly describe how conjunctions are processed in the NLP system. Conjunctions are important because they join other words and phrases together, allowing the relationships among multiple actions to be expressed explicitly. Without conjunctions, we could only make simple sentences and the relationships among multiple actions would be difficult to convey. Although parsing certain conjunctions, such as *and* and *or*, is tedious, their roles in English are invariant. Since the number of conjunctions is small, we hard-code the logic of parsing and processing conjunction words for now. Note that the actions connected by conjunctions are represented with the relationships determined by the conjunctions, which will be further utilized in natural language processing.

VII. Parsing

Given a Floglingo database defined with expressions representing entities, templates, and actions, the parser receives and parses a stream of words. Before the process starts, the database is collected with all words. In the form originally entered by clients, a word is considered as a noun (a word may not function as a noun, but we give it a chance to be a noun for the time being). When a word can be considered as other syntactical functions, we add a special token in the front of the word. (for verb, we add *Vw*, such as *Vw take*; for adjective, we add *Ajw*, such as *Ajw good*; for adverb, we add *Avw*, such as *Avw well*; for preposition, we add *Prew*, such as *Prew in*; for pronoun, we add *Prow*, such as *Prow it*; and for conjunction, we add *Cjw*, such as *Cjw and*.)

We also collect all idioms and fixed phrases and add their variant forms for their possible syntactical functions. For example, we have *new york city, take advantage of, Vw (take advantage of), responsible for, Ajw (responsible for), in spite of; Prew (in spite of)*.

To differentiate the different forms of a verb for its different tenses, we used special tokens *Vedw*, *Venw*, and *Vngw* for the past tense, the perfect tense, and the present progressive tense. Further we use the special token *Etym* to link them back to the verb original form itself. Here is an example [5]:

Vedw took Etym = Vw take;
Venw taken Etym = Vw take;
Vngw taking Etym = Vw take;

Many words may be preceded with multiple special tokens as they can be used for multiple syntactical functions, such as

[5] The representation of verbs and their various tenses generalizes homograph. For example, the word *saw* can function as a noun such as *a saw*, a verb in present tense *Vw saw*, and a verb in past tense *Vedw saw* for *see*.

Venw known and *Ajw known*. For a few commonly used words, such as *do*, *have*, *what*, *is*, *would*, we use only one special token in the front of each regardless its various syntactical functions (here we will not give the details).

Given a stream of words from clients, the parser remembers what have been parsed, which is partially matched with a most left sub term of a template, e.g., the stream *Joe takes an unforgettable vacation* matches the left most sub term *person (Vw take) vacation* of the template *person (Vw take) vacation (Prew in) location*.

The parser takes the next word from the stream and figures out what a role of the word potentially plays in the template. An essential step of the parser is to identify a phrase from a client input stream that functions as a noun phrase (without a p-phrase or a noun clause as its modifier). Given that the parser has considered a stream segment as the first part of a potential noun-phrase, which is represented by a type or an entity, say E, in the database, the parser will determine if the coming word, say W, can continue to contribute into the noun phrase. If W is a super type of E, or an instance of E, e.g., *bank* is a super type of *pnc,* or *pnc* is an instance of *bank*. If yes, the parser determines that W contributes into the phrase and chooses the instance as the resulting representation of the noun phrase, e.g., *pnc*. If E and W both are within a third term representing an entity, e.g., *pnc* and *joe* are both in *pnc account joe*, the parser determines that W contributes into the noun phrase and chooses the 3rd term (e.g., *pnc account joe*) to represent the resulting noun phrase for the time being. If E and W cannot be connected based on the database by the algorithm above, the parser determines that the W cannot contribute to the noun phrase. The parser continuing its parsing process by taking the noun phrase as it is and taking one of W's variant forms, such as *Prew in* when W is *in*, as the next 'word' right after the noun phrase. If one variant form of W is found unfit, another variant will be tried next.

This algorithm provides a flexibility to accommodate all potential phrases that make practical sense, even not in a good grammar structure. For example, we will allow *pnc bank* and *bank pnc* as valid phrases for *pnc*, and *joe pnc account*, *pnc account joe* as valid phrases for *pnc account joe* which is an entity in the sample database in Section II.

With the algorithm for identifying noun phrases, the other parts of speeches are relatively easier in parsing because all other syntactic forms other than nouns have their prefixed tokens.

If a sentence is written in a good grammar structure, which is modeled in templates and the database inventories sufficient knowledges that the sentence is concerned with, the parser will deterministically break down the sentence and identify a desired template. One exception is that some sentences may not be able to be broken down even they are in good grammar structures. These ambiguities happen when a p-phrase in a sentence follows an object after the verb and we cannot determine if the p-phrase modifies the object or the verb. Here is the well-known example [1], [6], [14]:

He saw the girl with a binocular;

A parser is able to detect the structures like the one above that potentially cause ambiguities. With a database that inventories entities and templates, the parser in this NLP system will further know what the entities exactly are in the positions of the objects and the prepositional objects. By referencing templates, pp-relations, and the entities in database, the parser will be able further diminish some ambiguities if not all.

Nevertheless, ambiguities exist in natural languages, accidently or intentionally. If a sentence exists an ambiguity, the parser will detect it during the parsing process and will report an error or raise a question to clients for a clarification in English during a conversation.

Many sentences don't closely follow legitimate grammar rules but they make practical sense and are frequently used in our daily life. The NLP system accommodate these kinds of sentences by explicitly building templates for them. For example, the sentence *she is 180 lbs*, instead of *she weights 180 lbs*, is often used for celebrities. We can build a template for it:

Person (Arew be) weight = Botran (person "weighs" weight ";");

Replacing pronouns with the entities or actions that the pronouns are referring to is part of the parser's job. It is done closely along with the process of breaking down sentences into pieces. We have given an example of understanding the most difficult pronoun *Prew it* in Section III. We don't discuss further here regarding the rest of the pronouns.

VIII. Understanding

When the parser matches an entire template, we say that the NLP system "understood" the parsed input stream, e.g., the parser found the template *(Vw take) vacation (Prew in) location* for the sentence *I took a vacation in Chicago*.

When the parser consumed all the words in an input stream and didn't reach the end of a template, the parser may be able to determine a template as the final state of parsing depending on specific circumstances. First, if the parsed portion of the template has become unique, i.e., there is no other templates in the database that share the parsed portion of the given template, then the parser takes the whole template as the final state. For example, given the template *organization (Vw buy) commodity (Prew from) store*, if there is no another templates in the database that has the left most sub term *organization (Vw buy) commodity*, and the input stream is *I bought a shirt*, then the parser will consider the entire template is for the client input stream even the input stream didn't have information about the missing portion *(Prew from) store*. Different from programming languages where a function or procedure cannot be executed until all of its arguments are passed with values, a template here will be potentially walked through multiple times when the NLP system starts to evaluate it. Any missing information may be derived from the context, obtained from clients by a real-time interaction, or simply ignored as it may not be needed.

Secondly, even if the parsed portion is a shared left most subterm of multiple templates, but if the parser searched those

sentences that were recently processed in history and identified a template with the same left most subterm, the parser will take the history template as its understanding to the input stream. For example, when the parser had responded the client with a question: *do you confirm to pay 100 dollars?* (which will be further discussed in the coming section), and when the client responded: *I confirm*, where *I confirm* didn't exhaust any one of the templates in the database, then the parser will consider that it understood the client's intention: *I confirm to pay 100 dollars*.

Given that a parsed template has an assignment, the NLP system utilizes the reduction system of Froglingo to reduce the input sentence and the reduction process will always end with a unique state. It means that the system will manipulate the database (with searching and/or updating) and eventually respond the client in a way that will be discussed in the coming section.

When the parser reaches the end of a template and noted that it has no assignment, the NLP system considers that it understood the input stream as it is, i.e., the input stream is in normal form. At the same time, it tries to correlate the input sentence with the parsed sentences in the history, and further respond the client in the way we will further discuss in the coming section.

When the parser cannot find a distinguishing template, i.e, there are multiple templates share a left most sub terms for a given client input stream, the parser consider it as an ambiguity. It would have to respond by generating a question based on the multiple templates with the shared left most sub term. For example, if the client entered a stream: *I took;*, the parser will be able to identify at least two templates that share the same left most sub term *person (Vw take)* which matches the input stream. Such templates include: *person (Vw take) $s1: [$s1 isa commodity];* and *person (Vw take) $s2: [$s2 isa idea] (Prew (in spite of)) thing.* In this case, the parser will generate a question like: *What did you take, a commodity or an idea?*

IX. Natural Language Generation

As soon as the NLP system "understands" a sentence, it is ready to process the parsed sentence before generating a response. Given a set of entities in our real life, people have various interests from different angles. Even for a single topic, the questions can be expressed in many ways. Given an understanding to a sentence, different people may respond differently. The NLP system was designed to mimic (approximate) the friendly interactions people communicate in natural languages. In Section IV (Abstraction I), we have introduced the built-in operators *Botran* and *Botape*. There are two more builtin operators *Bothink,* an abbreviation of "bot think", and *Botalk*, an abbreviation of "bot talk" that will be discussed in this section. They are the key components to utilize the linguistic structures embedded within templates for natural language generation.

In this section, we use a short demo that has been implemented to demonstrate how templates are constructed to support conversations in English. In the demo, the NLP system acts as a sales person for a store. It makes transactions on product purchasing and returning. It answers general questions about products and returning policies. Consequently, the NLP system is a code generator that takes client input streams in English that drive the NLP to collect new entities and new types.

A. Templates for product sales services

Based on the entities built in Section II regarding *joe*, *pnc*, and *walmart*, we construct templates that allow the system and clients to interact to make sales transactions. Here is the top level template to accept requests or inquiries:

```
organization (Vw purchase) commodity (Prew from) store =
    p0 (Bothink ("is" commodity "available in" store "?")) organization commodity store;
```

This template is intended for client requests of purchasing products, such as:

I would like to purchase a shirt from Walmart;

It is also intended for client general inquiries about sales, such as:

Can I buy a shirt?

In this subsection, we focus on the templates for transactions. We will come back to talk about general inquiries in the next subsection by using the templates for product return policies.

The template above has an assignment that calls a function named *p0*. Before we give the definition for *p0*, we discuss the built-in operator *Bothink*, an abbreviation for "bot think". Like the operators *Botran* and *Botape* discussed in Section IV, *Bothink* takes the following term *("is" commodity "available in" store "?"),* substitute the variables *commodity* and *store* with passing values, such as *shirt* and *walmart*, to re-generate a string, e.g., *is shirt available in walmart?*. The string is parsed by the parser to match with the following template:

```
commodity (Arew be) (Ajw available) (Prew in) store =
    (store inventory commodity number > 0);
```

Instead of *Botran*'s calling another template, *Bothink* makes reduction itself to get a *true* or *false* answer back and passes it on to the function *p0*, which is defined as:

```
p0 true organization commodity store =
    p1 (Botalk ("does" organization "confirm to pay" store (store inventory commodity price) "?")) organization commodity store;
p0 false organization commodity store =
    (("Sorry, the product " + commodity) + "is out of stock.");
```

If *Bothink* returns *true*, *p0* will call *p1* for a further reduction. Otherwise, *p0* will return a message back to the client, such as *Sorry, the product shirt is out of stock*. This return message was generated without using *Botape* to indicate that no any

transactions have occurred yet (we will see later that *botape* is used to signal a success of a transaction when the root level template is called).

Similar to *Bothink, Botape*, and *Botran, Botalk* takes the term after it *("does" organization "confirm to pay" store (store inventory commodity price) "?")* and regenerates a string like *does joe confirm to pay walmart 100 dollars?* by substituting variables with passing values and the value from the reduction of *store inventory commodity price* , e.g., *walmart inventory shirt price = 100*, where *dollars* was added because *100* was pre-defined as a *currency dollars*. Instead of trying to find an answer for the question, *Botalk* stops the proceeding and asks this question to the client. It waits until the client responded again such as:

I confirm to pay 100 dollars;

Note that the wordings from clients don't have to exactly follow the wordings in a question, as long as the parser "understood" the response, e.g., by matching the response with a substring of the question the NLP system had asked earlier. (By "understanding", we also meant any other means that work in our daily life will work here. For example, a client response may be: *Yes, I confirm.*) *Botalk* and the three other operators enable a friendly interactions between the NLP system and humans in English.

As soon as *Botalk* got an answer, the reduction process is resumed and eventually *Botalk* passes client's answer, *true* or *false* in this case as a statement is considered as an assertion, to the function *p1*, which is defined as:

```
p1 true $o:[$o isa organization] $c: [$c isa commodity] $s =
    (Botran ($o "pay" $s ($s inventory $c price) "dollar;")),
    (update $s inventory $c number=
        ($s inventory $c number-1)),
    Botape ($o "receive" $c "from" $s "at" timestamp "EST;");
```

where, *Botran* will eventually identify and call the following template:

```
organization (Vw pay) $s: [$s isa organization] currency =
    Botran ("pnc transfers" currency "from" (pnc account
    organization) "to" (pnc account $s));
```

With two *Botran* calls above, the client will receive the following confirmation:

pnc transfers 100 dollars from pnc account joe to pnc account walmart at 2022/03/09 20:23:13 EST. Joe receives a shirt from walmart at 2022/03/09 20:23:13 EST.

The NLP system is implemented to use present tense to represents actions that are deemed as a fact and past tense to represents gossips, i.e., actions that are not considered authentic yet.

The same set of templates are also used to support general inquiries from clients. For example, when a client asks *Can I buy a shirt?,* the NLP system will be able to respond like: *you would have to confirm to pay 100 dollars* if the NLP system notes that the walmart inventory has 1 or more than one shirt. We will talk more about it in the coming section.

B. Templates for customer support services

Things are inter-related. When the sales data from the above subsection is established, it can be referenced for other actions. In this section, we give templates for a "walmart customer support services" that have access to the sales data from the "*walmart* product sales services" as discussed earlier.

Through the discussion, we also extensively demonstrate the flexibilities the NLP system offers to clients when the clients make inquiries with limited information available to the NLP system.

Here is the top-level template to receive requests for product returns:

```
organization (Vw return) commodity =
    r0 (Bothink ("did" organization "buy a" commodity "from
    walmart?")) organization commodity;
```

To simplify our discussion here, we reduce the number of variables by removing the profile of *(Prew from) store* for p-phrases such as *from walmart* from the template. We simply assume that all clients reach out to the NLP for returning *walmart* products. The template is completely defined with the following additional definitions:

```
r0 true $o: [$o isa organization] $c: [$c isa commodity] =
    r1 (Bothink ("how many days has it been?")) $o $c;
r0 false $o: [$o isa organization] $c: [$c isa commodity] =
    (("I was unable to find out a record showing that you bought"
    + $c) + "in our inventory;");
r1 $m:[$m isa number] $o: [$o isa organization] $c: [$c isa
commodity] =
    Bothink ("if" $m "is within 30 days," $o "gives walmart" $c
    "and walmart pays" $o "what" $o "paid; if" $m "is more than
    30 days and" $m "is within 365 days, walmart repairs" $c
    "for free; if" $m "is more than 365 days, walmart doesn't
    offer a service for" $c ";");
number (Arew be) (Prew within) $n: [$n isa number] =
    (number <= $n);
number (Arew be) (Prew (more than)) $n: [$n isa number] =
    (number > $n);
organization (Vw give) $p: [$p isa organization] $c: [$c isa
    commodity] =
    (update organization inventory $c number = (organization
    inventory $c number -1)),
    (update $p inventory $c number = ($p inventory $c number
    + 1)),
    botape (organization "gives" $p $c ";");
organization (Vw repair) commodity =
    Botape (organization "repair" commodity);
organization (Vw offer) service (Prew for) commodity =
    botape (organization "offer service for" commodity ";");
```

With the definitions (note that the last two templates were simplified for this demonstration only), we describe how the NLP system reacts differently to three expressions that are matched with the tope level template:

1. A short inquiry with minimum information available (even without know the client's identity):

   Can I return a shirt?

This expression provides all information that the NLP system needs to parse and identify the template. At this moment, the NLP system only knows the value *shirt* for the variable (argument) *commodity* and who made this inquiry is unknown (the variable *organization* doesn't have a value). In stead of trying to ask a question to get an answer from the client, the NLP system goes ahead to walkthrough [6] the assigner (the function *r0* and its parameters). Since *organization* appears in the term *("did" organization "buy a" commodity "from walmart?")*, which means that *Bothink* will not get an answer from this term, and since the expression is a general inquiry, *Bothink* doesn't stop the proceeding but tells the function *r0* that it has no answer (except for the type it supposed to return). When *r0* isn't able to get a *true* or *false* value from *Bothink* and when *r0* knows that the process is for a general inquiry, it moves forward by trying out both *true* and *false* values. This strategy will allow the NLP system to further investigate to see what will happen by making an assumption that *Bothink* had returned a specific value. This process will continue recursively until it finds all tokens *Botape* reachable from the root template of the walkthrough. Once it finishes the walkthrough, it starts to prepare a response message by regenerating natural languages from the assumptions and the *Botape* clauses that had been walked through. Here is the generated message for the question *Can I return a shirt?* when *I* is unknown:

You must have bought the shirt from walmart. We will see how many days it has been. If the number is within 30 days, you gives walmart the shirt and walmart pays you what you had paid. If the number is more than 30 days and the number is within 365 days, walmart repairs the shirt for free. if the number is more than 365 days, walmart doesn't offer a service for the shirt.

The message was polished to meet the altitude and tendency of the client based on the verb itself, the verb mood, and potentially model auxiliary verbs in a sentence. To respond differently to client requests and inquiries, we collect patterns of sentence structures with certain verbs, verb moods as well as auxiliary verbs that indicate clients' desire rated with a degree. Here is a sample collection in the order of increasing desires [5]:

person think of doing thing.
may person do thing?
could person do thing?
can person do thing?
person plan to do thing.
person promise to do thing.
person (would) like to do thing.
please do thing for person.

2. An inquiry with a little more information (still without knowing the client's identify):

   I bought a shirt from Walmart. It has been 3 days. Can I return it?

With the two past tense sentences, the NLP system records the sentences and tries to correlate them with previously parsed sentences. Assume that there is no any sentence in history that is related to the first sentence, i.e., *I* was not mentioned earlier, the NLP system records this sentence (marking the beginning of a new conversation). When it parses the second sentence, it will match it with the template *Prew it (Arew be) intime*. This leads the system to calculate the time period *3 days* earlier than today (as discussed in Section III) and take the calculated date as the occurring date for the action represented by the first sentence. As a result, the NLP will combine the two sentence and have the following record in the system: *I bought a shirt from Walmart at 2023/03/09 20:23:13 EST*.

When the NLP system parses the third sentence (the question from the client), it will try to determine which template the sentence - *can I return it?* - will be matched with. Along with this decision, the NLP needs to figure out if *it* is the pronoun for *shirt* – a commodity or the action of *I bought a shirt* in the history.

If the database has the only template *organization (Vw return) commodity* that can be potentially matched with the question, the NLP system would simply make the match because *shirt*, a commodity, is the only choice for the pronoun *it*.

The question is: Is it possible to have another template down the road that matches client's question as well? If such a second template existed, it would look like: *organization (Vw return) action*, because *it* in the question can also be the pronoun for the action of *I bought a shirt* in the history.

Checking an English dictionary, we noted that an object in a sentence taking *Vw return* as the main verb is normally an entity. An action can be such an object for *Vw return*, but it is about giving or performing in return to a previous action, for example, *return his call* and *return a compliment*. So the action of a person's purchasing a commodity will not act as an object of a template with *Vw return* as the main verb. Therefore, the template *organization (Vw return) action* can be re-defined with a narrower scope than *action* at the position of the object, such as the resulting template may looks like: *organization (Vw return) $a: [$a isa call or $a isa compliment]*. Such a redefinition would require a greater effort, but make the NLP system smarter as it reduces the degree of ambiguity.

[6] In a programming language, we execute a function (procedure) by providing values for all the arguments of the function. In a parallel, here we allow a template to be executed (reduced) without values for some arguments. In the field of static program analysis, we walkthrough program source code outside of program execution. In a parallel, here we walkthrough templates for partial execution (reduction).

Nevertheless, the NLP system goes back to clients for a clarification by raising a question whenever it meets an ambiguity. For example, the question would be for the example discussed above:

Would you like to return the shirt or your purchase?

With the parsing process accomplished, the NLP system will walkthrough the same assigner that had been walked through in our previous use case. This time, however, the NLP system knows more about the inquiry: the purchase occurred 3 days ago. Knowing that this walkthrough is for a general inquiry which will not have any modification to the database, the NLP system will generate the following message:

You would give walmart the shirt. walmart would pay you what you had paid.

3. A request for transaction without adequate information from client:

   Hi, I bought a shirt from Walmart. I like to return it.

Since the client this time wants to make a transaction to modify the database, the NLP system will need to obtain all necessary information to complete the transaction. Since *I* is still unknow and it is requested in the clause *(Bothink ("did" organization "buy a" commodity "from walmart?"))*, the NLP system will generate the following message based on the template *organization (Vw return) commodity*:

May I know who likes to return a shirt?

As soon as the client identified himself, say *I am Joe* for the sample database discussed in this paper, the NLP system will continue to walkthrough the rest of the assigner, as it has done for the two earlier use cases. Since it has all information it needs this time, it will make a final transaction depending on how many days the product has been purchased. For a product that had been purchased for 3 days, the return message will be:

You give walmart the shirt. walmart pay you 100 dollar.

X. Code Generator

From the earlier subsections, we saw that client's text message in English can drive the NLP to produce outputs in English, which is enabled by templates created by their authors. Rather than directly asking the question of if client's text message can drive the NLP to produce Froglingo code or another programming language code, we ask this question: can English itself be a programming language which takes English sentences 1) as its executables and 2) as a tool to generate source code? We answer these two questions in this section.

We have already seen that the NLP system takes English sentences and produce outcomes in English. So English sentences are executables. Can English sentences as executables be as expressive as Turing machine? We give a template for the factorial function to show that English sentences can be executables equivalent to Turing machine (which itself is not a surprise as a natural language is a super language of any programming languages. But this result claims that we have a tool now, the NLP system, that uses natural language to drive machines as effective as programming languages):

```
fac (Vw take) $n:[$n isa number] (Prew to)
    (Vw produce) $m:[$m isa number]
=
   bothink ("if" $n "is 0," $m "is 1 or" $m is "the multiplication
   of" $n "with what fac takes" ($n -1) "to produce;");
```

Here is a conversation supported by the definition above which has been implemented as part of the NLP system:

Client: *What does fac takes 4 to produce?*
System: *fac takes 4 to produce 24.*
Client: *Does fac takes 4 to produce 24?*
System: *Yes, fac takes 4 to produce 24?*
Client: *Does fac takes 4 to produce 25?*
System: *No, fac takes 4 to produce 24.*

The second question: Can we build some templates such that clients can intentionally create new types and new entities by giving commands in English that eventually drive templates to carry out the execution of the commands? The answer is yes. We can simply add the following templates into the database:

```
newterm Inherit entity;
newterm (Arew be) (Thtw a) type =
   schema newterm Inherit type;
newterm (Arew be) (Thtw an) instance (Prew of) type =
   create newterm Belong type;
```

We can also enhance the NLP system such that its lexer and parser will accept new words or phrases from clients that are not in the database. It will first categorize them as type *newterm*. When the second or third template is called, the NLP system will change *newterm* to be *type* or an instance of *type* respective. Here are a few examples that trigger the creation of new types and entities:

```
virus is an entity;
covid is an instance of virus;
mathematical function is an entity;
fac2 is an instance of mathematical function which takes an
   integer to produce another integer;
```

These will introduce the following into database:

```
virus Inherit entity;
covid Belong virus;
mathematical function Inherit entity;
fac2 Belong (mathematical function);
```

With the new types and instances (entities) created above via English, similarly, we can introduce additional templates that allow clients to add properties for *the types* and *instances*.

To complete the definition of *fac2*, we can add additional sentences right after the first sentence regarding *fac2*:

If the former is 0, the second is 1. Otherwise, the second is the multiplication of the first with what fac takes the first integer minus 1 to produce;

Since *fac2* is defined as a mathematical function and if the type *mathematical function* has been enhanced with rules to define an instance, the NLP would know how to construct a template with assignment that would be identical almost to the definition of *fac* that we discussed earlier.

The last question: Can the NLP system allows other expressions other than "*is a type*" and "*is an instance*" to create new types and entities, such as *covid* and *fac2* respectively? If any equivalent expressions exist, the NLP system should allow various forms of equivalent expressions for the sake of the completeness of the NLP system as a code generator. We believe this completeness will become evident when the NLP system is accumulated with enough things and linguistic structures. On the other hand, the NLP system can always be constructed with equivalent templates (using *Botran*) to allow clients to express their requests in any newly discovered equivalent expressions, as the underneath entities are unchanged.

XI. Related Work

Given a finite set of things (entities and actions) as potential topics of natural language communications, there is a potentially infinite number of expressions (e.g., sentences) in natural languages regarding these things. Machine learning approaches don't intend to represent these things directly. Instead they learn from a finite number of sentences from the past and make predictions about (the future of) these things. This approach cannot guarantee that the predictions are true. Linguistic structure has been recently emphasized as part of the machine learning approach to reduce false conclusions [11], [12], [19]. However, rather than representing things, linguistic structure merely represents the syntactical form of expressions in natural languages. Instead of inventorying sentences, the NLP system inventories the representation of things in terms and templates. This approach avoids the complexity of a statistical approach, which has to consider a potentially infinite number of samples in disciplines such as biology where the functions (things) to be computed are unknown. The NLP system can avoid such complexity because the EP data model uniformly databases all kinds of things conveyable in natural languages. It uses finite (memory) space to expose infinite properties of certain (bounded) partially computable functions.

Although machine learning based approaches don't guarantee correctness, they addressed certain NLP issues and have been widely used in our daily lives. Does a deductive approach work at all as a NLP solution for long-awaited applications such as natural language and code generations? The answer to this question is the most critically dependent on the quality of the parser: Does the parser precisely and correctly parse a sentence that is uniquely meaningful to the author and the readers of the sentence? The answer is yes, because we can simply add a new template or modify an existing template to cover the sentence if there is not an existing one. Can two sentences that are meant differently be reduced to a single value? It is impossible as long as we don't construct the two templates that cover the two sentences respectively with assignments such that the two sentences be reduced to the same value. (In other words, the parsing structure and the semantic meaning of a sentence are fully controlled by the authors who construct the templates in a database.) As soon as a sentence is precisely and correctly parsed, i.e., the NLP system "understood" the sentence, the NLP system simply starts to reduce the sentence, during which nature languages or code are generated.

To generate natural languages, machine learning based approaches do not necessarily "understand" sentences but generate outputs from learning. The result may not be necessarily accurate, but it doesn't need software engineering to represent things (i.e., knowledges) referenced in the sentences. On the other hand, the Froglingo-based NLP system will be able to accurately generate results. However, it needs an effort of databasing knowledges to truly generate natural languages. Although this effort is a software engineering, the effort of doing so using Froglingo, the most productive tool, is significantly reduced from the programming effort of using the contemporary software development tools, such as BigData and the combination of a programming language and a database management system. In addition, the Froglingo based NLP system doesn't require to database all knowledges before processing a sentence. For example, we may simply have a template: *person (Vw take) walk* without an assignment. It would keep the covered sentence *joe takes a walk everyday* as the weak head normal form itself without being further reduced. When the NLP system has limited knowledges, it will be able to answer limited questions. For example, the system will be able to respond: *joe takes a walk everyday* when asked: *how often does joe take a walk?*; the system would respond: *I don't know* when asked: *how much calories does joe burn by taking a walk every day?*.

Rather than predictions in machine learning based NLP approaches, the deductive NLP system gives precise conclusions that are derived from its database. Therefore, it can act as a specialist to perform tasks that don't tolerate errors. For example, it can fill the role of a customer support 'agent' to answer client questions and to perform transactions; a 'tutor' in a classroom; a 'counselor' for personal, social, or psychological issues; and an 'advisor' providing reliable information before decisions are made.

Code generation is another topic in NLP. Generating natural languages using neural language models, e.g., masked language modeling (i.e., predicting next words or masked words in the middle of sentences [11]) is the most challenging, because the targeted natural languages are irregular in syntactical forms and the neural language modeling doesn't have any context as reference (except for experience learned from the past). Generating formal languages (such as SQL and even the more

complex Python [15], [26]) using neural language modeling should be relatively easier, because the targeted languages are simpler than natural languages. With enough information about words and linguistic structures in a database, the Froglingo-based NLP system takes input in natural languages to generate natural languages. As a 'natural' programming language, it takes input in natural languages to generate Froglingo expressions that drive to accumulate more entities and actions from various disciplines in the world. The NLP system works as a natural language and code generator because the generation process is driven by templates (for action profiles), terms (for entities), and sentences (for actions) in a database.

A neural language model that produces natural languages or linguistic structures (e.g., treebanks) normally doesn't detect ambiguity. The neural language models that work as code generators, e.g., semantic parsers [2], [9], [26], can detect and eliminate ambiguities by interactions, because targeted formal languages like SQL are simpler than natural languages. The Froglingo-based NLP system which generate both natural languages and Froglingo expressions, is also able to detect and eliminate ambiguities by interaction, because of the formal language Froglingo, as well as the templates and the context in its database.

When applying a machine learning based NLP system to a discipline, it has to be customized by newly identifying effective training data, which is expensive and only large corporates can afford it. The Froglingo-based NLP system has a common infrastructure embedding the abstractions within linguistic structures. Once it is built, it would become accessible to smaller organizations and individuals. Because this NLP system is a "native" programming language, it would allow everybody even without coding knowledges to database in natural languages for their personal needs (entities and actions). This will bring automation to each corner of our society to serve human beings.

XII. Conclusion

Databasing is an essential strategy in the NLP system, which collect information about (and accommodates exceptions from) things in the world and linguistic structures. Within Froglingo, terms and assignments serve as words and phrases representing real-world entities and properties. We use templates (terms in a special format) and their assignment to model clauses and simple sentences which represent real-world actions. We also accumulate abstractions (relationships) embedded in entities and actions of the world as well as in linguistic structures. The NLP system supports natural language and code generation by embedding linguistic structures into template assignments. The development of the NLP system is an accumulation process. As more things are accumulated, the NLP system will become increasingly robust. As more abstractions are accumulated the NLP system will become smarter and friendlier in interacting with others in natural languages.

This paper focuses on the symbolic approach using Froglingo, where the things in the world are uniformly represented in a database along with linguistic structure. It is an independent approach to NLP in parallel to neural language models. It also can be part of the neural language models to ground natural languages using linguistic structure and contextual knowledges [3].

References

[1] F. Benelhadj, "Prepositional Phrases across Disciplines and Research Genres: A Syntactic and Semantic Approach", A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy, University of Sfax Faculty of Letters and Humanities Department of English, 2015.

[2] J. Berant, A. Chou, R. Frostig, P. Liang, "Semantic Parsing on Freebase from Question-Answer Pairs", Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, pages 1533–1544, Seattle, Washington, USA, 18-21 October 2013.

[3] Y. Bisk, A. Holtzman, J. Thomason, J. Andreas, Y. Bengio, J. Chai, M. Lapata, A. Lazarido, J. May, Al. Nisnevish, N. Pinto, J. Turian, "Experience Grounds Language", Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, pages 8718–8735.

[4] E. Black, R. Garside, G. Leech, "Statistically-Driven Computer Grammars of English: The IBM/Lancaster Approach", Computational Linguistics, P. 498, Volume 20, Number 3, 1993.

[5] A. S. Cowena, D. Keltner, "Self-report captures 27 distinct categories of emotion bridged by continuous gradients", E7900–E7909, PNAS, www.pnas.org/cgi/doi/10.1073/pnas.1702247114, September 5, 2017.

[6] M. H. Hamdan, I. H. Khan, "An Analysis of Prepositional-Phrase Attachment Disambiguation", International Journal of Computational Linguistics Research Volume 9 Number 2 June 2018.

[7] W. Han, H. T. Ng, "Diversity-Driven Combination for Grammatical Error Correction", 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI), November 2021.

[8] M. A. Hedderich, L. Lange, H. Adel, J. Strötgen, D. Klakow, "A Survey on Recent Approaches for Natural Language Processing in Low-Resource Scenarios", Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2545–2568. June 6–11, 2021. ©2021 Association for Computational Linguistic.

[9] F. Li and HV Jagadish, "Constructing an interactive natural language interface for relational database", Proceedings of the VLDB Endowment, 2014, 8(1):73–84.

[10] D. M. Magerman, "Statistical Decision-Tree Models for Parsing", ACL'95: Proceedings of the 33rd annual meeting on Association for Computational Linguistics, June 1995, Page 276 – 283.

[11] C. D. Manning, K. Clark, J. Hewitt, U. Khandelwal, O. Levey. "Emergent linguistic structure in artificial neural networks trained by self-supervision, 30046-30054, PNAS, December 1, 2020, Vol. 117 No. 48.

[12] M. C. de Marneffe, C. D. Manning, J. Nivre, D. Zeman, "Universal Dependencies", Computational Linguistics (CL), 25 February 2021.

[13] M. Minsky, ""Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy", in Artificial Intelligence at MIT., Expanding Frontiers, Patrick H. Winston (Ed.), Vol 1, MIT Press, 1990. Reprinted in AI Magazine, 1991.

[14] K. Nadh, "Prepositional phrase attachment ambiguity resolution using word sense hierarchies", Undergraduate thesis project, Middlesex University, London, U.K, February - April 2008.

[15] L. Perez, L. Ottens, S. Viswanathan, "Automatic Code Generation using Pre-Trained Language Models",

cs230.stanford.edu/projects_spring_2020/reports/38907662.pdf, arXiv:2102.10535v1 [cs.CL] 21 Feb 2021.

[16] J. R. Pierce, J. B. Carroll, E. P. Hamp, D. G, Hays, C, F, Hockett, A. G. Oettinger, A. Perlis, "Language and Machines, Computers in Translation and Linguistics", A Report by the Automatic Language Processing Advisory Committee, Division of Behavioral Science, National Academy of Sciences, National Research Council, National Academy of Science, National Research Councile, Publication 1416, 1966.

[17] S. Ruder, "The 4 Biggest Open Problems in NLP", https://ruder.io/4-biggest-open-problems-in-nlp/.

[18] A. See, "Neural Generation of Open-Ended Text and Dialogue", A Dissertation Submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philsophy, August 2021.

[19] A. See, "Deep Learning, Structure and Innate Priors, A Discussion between Yann LeCun and Christopher Manning", https://www.abigailsee.com/2018/02/21/deep-learning-structure-and-innate-priors.html, February 21, 2018.

[20] J. Weizenbaum, "Computational Linguistics, ELIZA – A Computer Program for the Study ofNatural Language Communication Between Man and Machine", Communications of the ACM, Volume. 9, Number. 1, January 1966.

[21] K. Xu, "A Class of Bounded Functions, a Database Language and an Extended Lambda Calculus", Journal of Theoretical Computer Science, Vol. 691, August 2017, Page 81 - 106.

[22] K. Xu, "The Enterprise-Participant Data Model, an Untyped Recursive Language Semantically Approximating the Lambda Calculus", 2014 Computability in Europe Conference (CiE 2014), Budapest, Hungary, June 2014.

[23] K. Xu, J. Zhang, S. Gao, "Froglingo, a Programming Language empowered by a Total-Recursive-Equivalent Data Model", Journal of Digital Information Management (JDIM), Volume 9, Number 4, August 2011, Page 135 - 146.

[24] K. Xu, J. Zhang, S. Gao, "Approximating Knowledge of Cooking in Higher-order Functions, a Case Study of Froglingo", Workshop Proceedings of the Eighteenth International Conference on Case-Based Reasoning (ICCBR 2010), pp. 219 – 228.

[25] K. Xu, "User's Guide to Froglingo, An Alternative to DBMS, Programming Language, Web Server, and File System", http://www.froglingo.com/FrogUserGuide20.pdf, Release 2.0, March 14th, 2013.

[26] Z. Yao, Y. Su, H. Sun, W. Yih, "Model-based Interactive Semantic Parsing: A Unified Framework and A Text-to-SQL Case Study", EMNLP-2019.