Chemistry Studio : An Intelligent Tutoring System (Natural Language Component)

Ankit Kumar
Y8088
ankitkr@iitk.ac.in

Abhishek Kar
Y8021
akar@iitk.ac.in

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Ashish Tiwari
SRI International
tiwari@csl.sri.com

Amey Karkare
IIT Kanpur
karkare@cse.iitk.ac.in

Abstract— **We present a prototype of an expert system that can aid students in learning Chemistry, specifically Periodic Table, by automatically solving problems posed in natural language. Our work involves natural language processing of the posed questions and automatic conversion into an intermediate logical representation. We employ a novel tree based parsing mechanism to build a type-safe logical formula out of recognized tokens. Our system can handle both existentially and universally quantified queries. We also demonstrate parsing of assertion based questions spanning multiple sentences. Finally, we come up with a number of heuristics to make our system more robust and report the results on a dataset of Periodic Table problems collected from various sources. The entire system is developed in C#.**

I. Introduction

In this work, we present an intelligent tutoring system aimed at instructing students by simulating the reasoning processes of a student aiming to solve a problem. The current state of the art systems are static in nature and answers given by them are based on a simple lookup in their databases. We aim to develop a system that will take into account the knowledge base of the student and try to emulate the inference mechanisms used by him/her to arrive at a solution. Hence, our system aims at solving a problem by a systematic reasoning process similar to that employed by students while attempting the problem.

We have chosen the domain of Periodic Table problems in Chemistry meant to be tackled by students from grades 9 through 12 for our tutoring system. The simplicity and structured nature of the domain makes it ideal for translation from natural language to a logical formulation and subsequent problem solving. Our work involves natural language processing of the posed problems and their conversion into a logical representation.

Our system encapsulates a number of novel techniques for processing the questions posed in natural language and employs heuristics to make it robust in the domain of Periodic Table chemistry. We organize the report as follows. In the following section, we describe the previous work in this field and how our approach is different. Section III provides a brief overview of the system followed by the description of our dataset in Section IV. Next we introduce our intermediate logic and give details of our system design in Section VI. Section VII through XII describe special techniques used by us to make our algoruthm robust Section XIII deals with real world examples and how our algorithm deals with them. Section XIV presents the results on our dataset and finally we conclude with some ideas and directions for future research on this topic.

II. Related Work

Current similar work includes the Geoquery project at University of Texas, Austin [1] [2] [3] [4] wherein various machine learning techniques and formal grammars are used for machine translation. A Question-Answering System for Advanced Placement chemistry dealing with problems in stoichiometry and equillibrium reactions was developed as part of the Halo Pilot project by Vulcan Inc. The system uses a combination of several modern Knowledge Representation and Reasoning technologies, in particular semantically well-defined frame systems, automatic classification methods, reusable ontologies, and a methodology for knowledge base construction [5]. The system provides the reasoning in reaching its final answer in English as the templates are hard coded into it. It is to be noted that this system used problems manually encoded into the domain language of the system and Natural Language Processing was out of scope of this project.

A number of Intelligent Tutoring Systems occur in the domain of physics. Isaac (Novak et al) [6] and Mecho (Bundy et al) [7] are examples of systems that attempt to parse sentences posed in natural language and convert them into semantic frames and logical formulae respectively. Pyrenees is an example of a model tracing tutoring system for equation based problems where problems were manually translated into the domain language. A recent work by Jung et al [8] proposes using natural language to represent knowledge in an intelligent tutoring system. This enables instructors to easily add to the knowledge base and even debug it. The system solves problems in the domain of kinematics and mechanics though it can be extended to different domains by appropriately encoding the mental models corresponding to that domain (e.g. concepts of group, period etc in periodic table chemistry). It uses hypernyms from WordNet to find broad concepts and does the parsing according to it.

We employ a much simpler approach that utilises the well-structured properties of the problems in this domain by inferring the terms in the logic from certain cues in the question. We then propose algorithms that arrange these terms to form a well formed logical representation of the question. We find that with the help of appropriate ranking mechanisms and heuristics, we can successfully extract the correct logical representation from multiple possible arrangements of the terms.

III. System Overview

The entire system is comprised of two basic components - Natural Language Processing and Problem Solving. Given a problem posed in the natural language, it is first processed using our proposed algorithm to convert it into an intermediate logical representation. An intermediate language has been formulated to encapsulate the breadth of the domain of Periodic Table problems. The intermediate representation generated is then subsequently processed by a theorem proving system that has a database of the rules and facts that govern the domain of periodic table. The goal of the problem solving system is not just to solve the problem using raw facts, but to emulate the reasoning process of a student studying the subject. Ultimately we would like to generate problems with graded difficulties and automatically provide hints to students stuck at a point.

Our contribution to the project is the conversion of questions posed in natural language to the intermediate logical form.

IV. Dataset

As a preliminary to our work, we collected a number of benchmark problems in the domain of Periodic Table comprising of MCQ's, true and false questions, fill in the blank and interrogative questions meant for students between grade 9 and 12. The sources include course textbooks, standardized tests like SAT Chemistry, California Star Chemistry and GRE Chemistry, and various other educational websites. The current corpus size is around 200 questions.

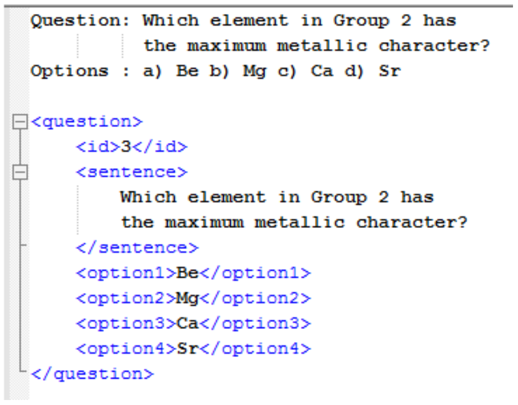
These sample questions were used to identify the keywords of the domain corresponding to which *terms* in the logic were created in the form of *predicates*, *functions* and *simple terms*. It also helped us to identify certain cues whose presence in a sentence indicate the occurrence of some token in the corresponding logical representation. We compile such a mapping of cue phrases to tokens.

We encode certain rules in the system based on the problem corpus. The evaluation metric for our translation scheme will be the ratio of the size of the encoded rules to the size of the problem set that the system is able to translate successfully.

All these problems were then manually converted into their intermediate logical form to match our result against and provide a starting base for the problem solving team. The problem set is organised in XML form (Fig. 1) which is the input format for our translation system.

V. Intermediate Logic

We interpret the logical representation of a question as the choice of the domain variable that makes the formula true.



```
Question: Which element in Group 2 has
          the maximum metallic character?
Options : a) Be b) Mg c) Ca d) Sr

<question>
    <id>3</id>
    <sentence>
        Which element in Group 2 has
        the maximum metallic character?
    </sentence>
    <option1>Be</option1>
    <option2>Mg</option2>
    <option3>Ca</option3>
    <option4>Sr</option4>
</question>
```

Fig. 1: Input XML format of the question

The domain variables are the free variables appearing in the formula. The set of entities over which the domain variable varies depends on the type of question (e.g. the options in an MCQ).

For example, consider the question : "What is the atomic number of calcium ?". At first glance, one might be tempted to formulate this as *AtomicNumber(Ca)*. However, according to our interpretation of logical representations, the correct translation would be to *Same(AtomicNumber(Ca), $1)* where $1 is the domain variable varying over the set of possible numbers. We have decided to have such a formulation keeping in mind the nature of the system involved in problem solving - a theorem proving system. Here we make use of the fact that with such an interpretations of logical representations, the problem solving component will easily utilize a theorem prover to test the values of the domain variables that make the logical representation *true*.

The intermediate logic is comprised of terms which can be classified into the following:

- Predicates : They can be unary, binary or ternary. These terms take as input other terms and return a boolean value. Some of them are enlisted in table I.
- Functions : These terms take as input other terms and return some value which is a function of the input terms. Some of them are enlisted in table I
- Simple terms : These are comprised of terms corresponding to the elements, numbers, variables etc. which do not have any nested term inside them. Some of the terms with their types are enlisted in table I

We assign an input and output type to all of the above terms. These terms can appear nested within each other such that the output type of a nested sub-formula satisfies the input type of the enclosing term. For example, *AtomicNumber(Ca)* represents the atomic number of Calcium, *AtomicNumber*

Predicates	Functions	Simple terms (Type)
AlkaliMetal(element)	AtomicNumber(element)	Ca (Element)
Halogen(element)	AtomicRadius(element)	Li (Element)
AlkalineEarthMetal(element)	IE(element)	15 (Numeric)
RareEarthElement(element)	Group(element)	AtomicRadiusProperty (NumericFunction)
Max(numericfunc,bool)	Period(element)	Increase (change)
And(bool,bool)	OxidationState(element)	Stays_Same(change)
Or(bool,bool)	IonicRadius(element)	Up (movement)
Same(num,num)	ElectronAffinity(element)	Left_down (movement)
Order(numericfunc,change,set)	Conductance(element)	\$0 (variable)
Trend(movement,numericfunc,change)	Reactivity(element)	\$1 (variable)

TABLE I: Terms in the intermediate logical representation

being a unary function that takes a simple term of type element (here Ca) and returns the Atomic number which is of type *numeric*. *Order(AtomicNumberProperty, increase, {Li,Be, H})* represents a boolean formula which returns true if the set of elements Li, Be, H are in increasing order of their atomic numbers. Here, *Order* is a ternary predicate that takes 3 elements - a simple term of type *numeric function*, a simple term of type *change* and a simple term of type *set* which is an ordered set of simple terms of type *element*.

The logical formulation of predicates *Max()* and *Min()* are worth discussion. Max and Min require a function that they maximize/minimize and a domain over which the input function needs to be maximized/minimized. This *domain* type is modeled by any sub-formula of type boolean. This has been made possible because of the fact that according to our interpretation of formulae, a sub-formula is interpreted as all the values for the domain variable that make this sub-formula true, hence any sub-formulae of return type boolean can model a domain type. For example, *Max(IEProperty,Same(Group(\$1),12))* restricts the maximization of the ionization energy property to the domain of all elements satisfying the formula *Same(Group(\$1),12)* i.e. elements belonging to Group 12 of the periodic table.

VI. System Design

The process of translation of the question to its intermediate logical representation is done in three steps as illustrated by Fig. 2. First comes the lexer that chunks the input problem and identifies the various terms in the logic appearing in the question. Then comes the parsing of the options provided in Multiple Choice Questions that provides additional insight into the problem as we will describe shortly. The final phase is the parser that tries to arrange the identified set of terms into a type-safe logical formula.

A. Lexer

The algorithm employed by the lexer is elucidated in Algorithm 1. At any point of time, we loop through the list of cue phrases and try to find the best matching cue with some prefix of the input sentence. If we are successful, we chop off the corresponding portion and loop again. Otherwise, we remove

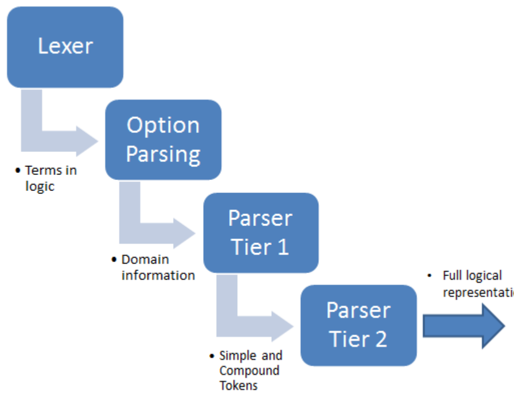


Fig. 2: Stages involved in translation

Algorithm 1 Lexer Algorithm

```
Require: $S$ : Problem Sentence, $M$ : Directory mapping cue phrases
    to tokens, Returns $L$ : list of identified tokens
1: while S.Length != 0 do
2:     for cue phrase C in M do
3:         Find if some prefix of S matches with C
4:     end for
5:     if some match is found then
6:         Choose the match with highest confidence, add corresponding token to $L$ and chop off the prefix from $S$
7:     else
8:         Chop off 1 word from beginning of S
9:     end if
10: end while
11: return L
```

the first word from the sentence, deeming it extraneous. This process is continued till the whole sentence is consumed.

Note that these cue phrases can appear in the sentence in any of their derivative forms like *-ing*, *-es*, *-ed* etc. This problem can be dealt with using stemming. However, currently we are using a variation of the Levenshtein edit distance [9] to compute a normalized similarity index between the two strings. This also helps us in dealing with the misspelt words. We define the similarity score between two strings S_1 and S_2 to be equal to $1 - LD(S_1, S_2)/Max(S_1.length, S_2.length)$. Where $LD(S_1, S_2)$ represents the Levenshtein edit distance between the two strings.

The lexer also collects certain metadata corresponding to each identified token. This includes the position information of the corresponding cue phrase in the sentence. The Levenshtein edit distance similarity score is also stored as a confidence measure (to be used in further phases).

B. Parsing options in MCQ

In this module, we try to extract information about the final result required by the question by parsing the options of a Multiple Choice Question. For example, if all the options are elements, we can infer that the result variable in the formula should be of type *element*. Similarly, if we find that an option contain an ordering of elements (e.g. Be $<$ B $<$ Li), we can infer that the question requires the *Order* predicate with an *increases* parameter and an ordered set consisting of {Be ,B ,Li} as per specifications of the *Order* predicate. Effectively, this step not only helps us in inferring the type of the final result variable in the logical formula but also insert certain implicit tokens in the question (*Order* in the above case). It also aids in deciding the number of domain variables to be introduced.

Example: Which set of elements contains a metalloid?
a)Ca, H, Li b)N, O, P c)As, Mo, U d)Xe, Ar, Kr
Formula: Or(Metalloid($1), Or(Metalloid($2), Metalloid ($3)))

Here, the parsing of the options gave us the insight of adding three domain variables to the list of tokens which get arranged in type-safe manner in parsing phase.

An option of *numeric* type in an MCQ helps us in tier 2 of the parser as described in the next section. In case of True/False questions, we allow for the introduction of the *Implies(Hole)* token as we describe in the section dealing with *Forall* queries.

C. Parser

As described previously, the logical representation of an input sentence is comprised of terms appearing inside one another in a nested form where the output type of the nested term (or the nested sub - formula) satisfies the input type of the enclosing term. Such an arrangement of terms can be viewed as a tree structure where each node represents a term with its children subtrees representing the nested sub-formulae. We call such a tree the **representation tree** of the logical formula. The internal nodes represent the predicates/functions that accept other terms as their input (children). The leaves correspond to the simple terms that take no further input tokens.

The parser takes the list of identified terms from the lexer and creates nodes corresponding to them. For each identified term, it creates a node whose children are unassigned at the moment and we call these unassigned nodes as **holes**. In totality, the job of the parser is to fill these holes with other subtrees in a type safe manner such that the final tree generated has no hole. Such a generated tree with no hole represents a logical formula. Please note that corresponding to a set of input terms, there might be multiple ways of type-safe arrangement of the corresponding nodes. This calls for a ranking scheme that will be used to find the most likely representation tree for a problem sentence.

A partial representation tree is one in which all nodes are not assigned i.e. containing some holes. Simple tokens are partial representation trees wherein only one (the root) is assigned. For example, simple token corresponding to the *AtomicRadius* predicate will be *AtomicRadius(Hole)*. A compound token is formed by arranging some partial tokens in type-safe manner to form a partial representation tree.

The parser is divided into two tiers. The first tier is used to construct compound tokens from simple tokens in order to exploit the local structure of the input sentence. The second tier takes the compound tokens from first tier and assembles them in type-safe manner to generate all possible representation trees. It then uses a ranking scheme to identify the most likely representation for the input problem sentence.

1) Tier 1: In our initial design, we only had the type-safe assembling of the simple tokens. However, we found that such a scheme resulted in construction of number of extraneous formulae.

Example: Which element is in Group 3 and period 2?
Formulae if only tier 2:
And(Same(Group($1) , 3), Same(Period($1), 2))
And(Same(Period($1) , 3), Same(Group($1), 2))

We noticed that this could be avoided by utilizing the local structure in the sentence to from some compound tokens from the simple ones before arranging them in tier 2. The position metadata collected by the lexer was used in formation of compound tokens.

Some of the ways in which the local structure of the problem has been exploited are listed below :

- Association of numbers with numeric predicates based on proximity : For every number encountered in the input sentence, we associate it with the different numeric functions discovered in the sentence with confidences proportional to their proximity scores. For example, in the previous case, the most likely predicate associated with 3 would be *Group* and the 2 would be *Period*. Thus we generate the compound tokens *Same(Group(Hole),3)* and *Same(Period(Hole),2)* in tier 1 to reflect this structure.
- Association of equality predicate with a numeric function based on proximity: If we encounter a phrase that calls for checking equality of two numeric functions, we generate a compound token to reflect this sub-structure.
  Example: Which of the following are in the *same group* as Oxygen?
  Formula: Same(Group($1), Group(O))
  In the above example, we generate the compound token *Same(Group(Hole),Group(Hole))* based on the phrase *same group*.

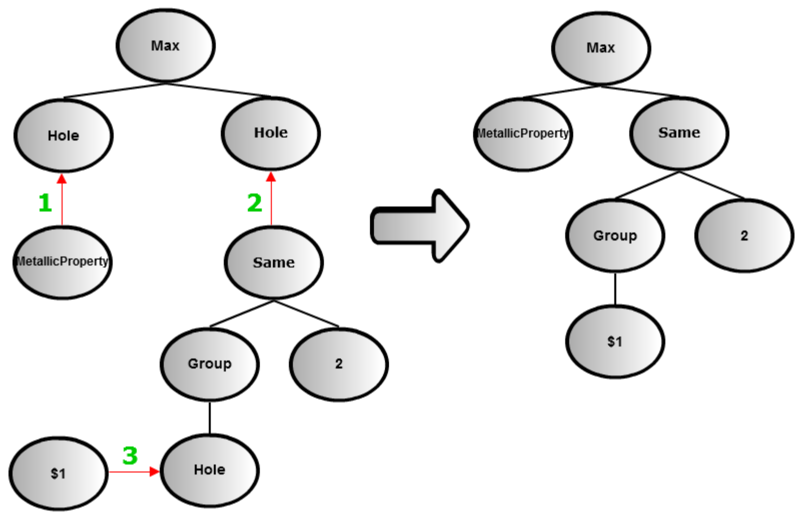


Fig. 3: Steps involved during construction of representation tree

- Identification of certain terms which generally occur coupled with other terms: In some cases, appearance of a term in the problem sentence calls for the introduction of certain other terms that frequently occur together and its occurrence in the sentence may not be explicit.
  Example: As atoms of elements in Group 16 are considered in order from top to bottom, the electronegativity of each successive element:
  1) Increases 2) Decreases 3) Remains the same 4) Can't say
  Formula : Trend(down, Electronegativity, $1)
  In this example, we infer the presence of the *Trend* predicate from the occurence of the term *down* as they usually appear together in sentences.

As evident from the last example, the tier not only formulates compound tokens, but also inserts certain tokens implicit in the question.

2) Tier 2: We develop an algorithm that takes as input a set of sub-expressions in the logic, and constructs another expression that ideally uses all of those sub-expressions. The number of such expressions can be more than one. For developing this algorithm, we exploit the types associated with various sub-expressions.

This tier takes the compound as well as simple tokens from tier 1 and tries to assemble them in a type- safe manner to form a well-formed representation tree. There could be two approaches to do this - the *top down* approach and the *bottom up* approach. In the top down approach, at any step we have a partial representation tree with holes to be filled and we take a decision of which hole to fill and which token to fill it with. After taking each such decision, we branch out a recursion path with that decision at that point. Effectively, this is a systematic exploration of the search space. In the bottom up approach, we start with the terminal tokens (leaves) and try to group them at each step to form bigger partial representation trees, effectively providing us with a list of group of partial representation trees. The top down approach provides better control over each partial representation tree as in a recursion path, we have only a single representation tree to deal with which makes it easier to employ various heuristics and ranking schemes in our algorithm. This is why our algorithm is in a top down manner in which at each execution step, we fill the leftmost hole in the partial tree.

Algorithm: We use a top down recursive algorithm to fit all tokens in a type-safe manner. The algorithm is elucidated in Figure 2

The algorithm starts with unused tokens and all tokens set to all compound and simple tokens obtained from tier 1. The initial partial tree with which the algorithm starts up is **Hole** of the type boolean (in agreement with our interpretation of logical representations). Every call of the function Type-Safe takes as input a list of unused tokens, a global list of all tokens and the partial tree formed till now by a particular decision path. The *decision path* refers to the sequence of hole-filling decisions taken which has resulted in the current partial tree. If the confidence of the partial tree becomes less than a threshold value we discard this decision pathway and return. Otherwise, there can be two situations.

- The current partial tree has no holes left. This can have two implications : either we have successfully completed the construction of the representation tree or we have assembled a complete subtree of the full representation tree. To handle the first case, we add this completed tree to the global list of full representation trees. We penalize the confidence of this tree in proportion to the count of unused tokens. To handle the second case, we make use of an important observation. As explained previously, we started the construction of the tree with a hole of

Algorithm 2 Type-Safe Arrangement Algorithm

```
Require: unused tokens, partial tree, all tokens
 1: if partial tree.confidence < threshold then return
 2: end if
 3: if partial tree has no holes then
 4:     Add this tree to the global list of completed trees with
        confidence decreased in proportion to number of tokens unused.
 5:     if there are unused tokens then
 6:         Create a new tree with "And" as the root and the current
            partial tree as its left child with a hole as the right child
 7:         typeSafe(unused tokens, new tree, all tokens)
 8:     end if
 9: else
10:     if no unused tokens left then
11:         for token t in all tokens do
12:             if t can fill hole of partial tree then
13:                 Newtree = holefill(partial tree, t)
14:                 Newtree.confidence = oldtree.confidence * penal-
                    ization factor
15:                 Type-safe(unused tokens, new tree, all tokens)
16:             end if
17:         end for
18:     end if
19:     if there are unused tokens left then
20:         for token t in unusedTokens do
21:             if t can fill hole of partial tree then
22:                 new tree = holefill(partial tree, t)
23:                 Type-safe(unused tokens \ {t}, new tree, all to-
                    kens)
24:             end if
25:         end for
26:         if none of the tokens are able to satisfy the hole then
27:             for token t in all tokens \ unused tokens do
28:                 if t can fill hole of partial tree then
29:                     new tree = holefill(partial tree, t)
30:                     new tree.confidence = old tree.confidence *
                        penalization factor
31:                     Type-safe(unused tokens, new tree, all tokens)
32:                 end if
33:             end for
34:         end if
35:     end if
36: end if
```

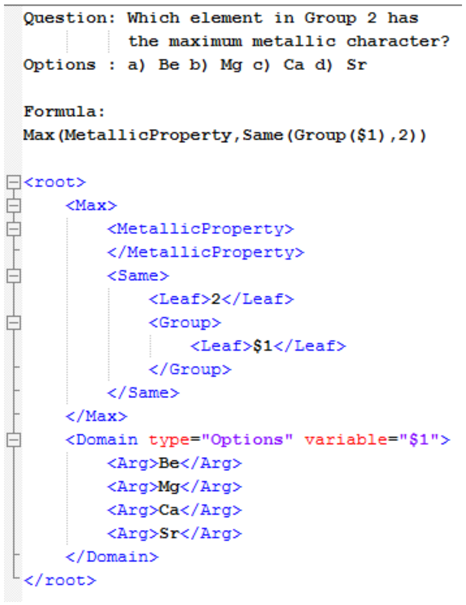
return type boolean. This implies that the current tree formed would have appeared in the full representation tree in conjunction with some other subtree which could have been assembled from the currently unused tokens. We found that happens in cases in which there is an implicit conjunction/disjunction in the problem sentence. We handle this case by creating a new tree with its root as *And() / Or()* predicates with the current completed tree as its left child.
Example : Which element in period 3 is a gas at STP?
Formula : And(Same(Period($1), 3), IsGasAtSTP($1))
Explanation : Note that there is no explicit cue for an And() predicate in the problem sentence. As a result, the lexer fails to guess its presence. During the course of the algorithm, at some stage we form a complete tree as *Same(Period($1), 3)* with {IsGasAtSTP, $1} as unused tokens. As explained above, we add an And() predicate thus forming a partial representation tree of the form *And(Same(Period($1), 3), Hole)*. This hole is subsequently filled by the remaining unused tokens in type-safe format resulting in the final formula given above.

- The current partial tree has holes. In this we might encounter one of the two scenarios.
  - We have exhausted the list of unused tokens. This might happen when a token appears more than once in the full representation tree but is detected by the lexer just once due to occurence of a single cue. We handle this by reproducing tokens from the list of all tokens one by one. If a token is reproduced that can fill the hole, we reduce the confidence of the new tree generated by a penalization factor and then recurse.
    Example : Which if the following elements have the maximum atomic radius and metallic character ?
    Formula : And(Max(AtomicRadiusProperty, $1), Max(MetallicCharacter, $1))
    Explanation : The lexer recognizes the 'maximum' cue just once in the question and produces the *Max()* predicate just once. On the other hand, our logical formula requires two *Max()* predicates to form a valid logical formula. The above heuristic of ours solves this problem.
  - We have a list of unused tokens from which we have to find a token that correctly fills the hole in our partial tree. We do this by iterating over all the tokens in the unused tokens list and trying to fit them into the hole. For every such token found, we fill the hole with the token and recurse. In case no token was able to fill the hole in the tree we try to fill the hole with one of the previously used tokens in the tree. This list is obtained by taking the set difference of all tokens and unused tokens. If such a token is found, we fill the hole and recurse with the confidence of the new tree decreased by a penalization factor.

After all recursion branches return, the global list of completed trees will contain all the trees generated with their confidences. We pick the tree with the highest confidence and report is as the correct representation tree of the problem sentence. The output of our system is represented in XML format which is shown in Figure 4. We define tags corresponding to each of the terms in the logic and the XML output is produced by outputting appropriate tags during pre-order traversal of the completed tree.

Special Techniques: We employ certain techniques and heuristics during the course of the algorithm to remove extraneous results. We describe two of them below.

- **Permutation Removal**: Our logical representation contains many symmetrical terms i.e. terms with multiple inputs of the same type. For example, *Same(..., ...)* takes as input two numeric type sub-formula.



```
Question: Which element in Group 2 has
          the maximum metallic character?
Options : a) Be b) Mg c) Ca d) Sr

Formula:
Max(MetallicProperty,Same(Group($1),2))

<root>
    <Max>
        <MetallicProperty>
        </MetallicProperty>
        <Same>
            <Leaf>2</Leaf>
            <Group>
                <Leaf>$1</Leaf>
            </Group>
        </Same>
    </Max>
    <Domain type="Options" variable="$1">
        <Arg>Be</Arg>
        <Arg>Mg</Arg>
        <Arg>Ca</Arg>
        <Arg>Sr</Arg>
    </Domain>
</root>
```

Fig. 4: Output of our algorithm in XML form

In such cases, we observed that the algorithm produced semantically equivalent completed trees whose textual representations were permutations of each other. For example *Same(Group($1),Group(Li))* and *Same(Group(Li),Group($1))* are permutations of each other and are semantically equivalent. Let's see how both of these were generated by the algorithm from the same query.
The second tier of the parser received the following tokens from the previous stages - i) *Same(Group(Hole),Group(Hole))* ii) *$1* iii) *Li*. Consider the execution step where the current partial tree is *Same(Group(Hole),Group(Hole))* and the unused tokens are {*$1,Li*}. As both of these are of type *element* and thus capable of filling the first hole, there will be two recursive calls - one where the hole is filled with *$1* and the other where the hole is filled with *Li*. The former return the first formula and the latter returns the second formula.
In order to avoid this phenomenon, we use a permutation removal algorithm. We define the value of a subtree to be its textual representation (its pre-order traversal). We enforce the following constraint at each internal node of the tree created:
Constraint: At each internal node of a representation tree, all its children with the same type should be ordered in the lexicographic order of their *values*.
We maintain this invariant at every execution step of the algorithm by checking for satisfaction of the constraint along the path from the parent of the hole being filled to the root. If any node in this path violates this constraint, we discard this recursion path. Note that we define the value of a hole to be a character whose lexicographic value is infinity.

- **Variable Branch Removal**: We employ a heuristic that helps us in pruning some generated trees that, though well-formed, are semantically meaningless from the point of view of a question.
Example: Which element is in the same group as Lithium and same period as Barium?
Formula: And(Same(Group($1),Group(Li)), Same(Period($1),Period(Ba)))
Extraneous formula: And(Same(Group(Ba),Group(Li)), Same(Period($1),Period(Ba)))
Explanation: Note that the extraneous formula is type consistent. However, the subtree Same(Group(Ba),Group(Li)) does not contain any variable terms and hence has a constant value irrespective of the value taken by the domain variable. Hence, this formula is meaningless from the point of view of the question and is hence, discarded.
Such examples call for the following heuristic:
Heuristic: At least one of the children subtree of every *Same()* node in a tree should have at variable in it. All children subtrees of every *And()* and *Implies()* node in a tree should have a variable in it.

VII. Automatic Introduction of Free Variables

We observed that our algorithm suffered in questions where implicit introduction of free variables was needed in order to formulate a valid logical formula. Common examples include the questions involving quantifiers as well as assertion based questions.

Example: Alkali metals belong to group 1 of the periodic table.

In the above example, we need to infer that there is an implicit variable (say $1)that needs to be checked whether it belongs to group 1 and then be quantified over the domain of all alkali metals. Note that there is no cue phrase in the question to suggest the introduction of such a variable and our algorithm would, in this case, fail to fill up the hole corresponding to this token. Note that this kind of a situation arises in True/False based questions as even the options dont provide us with a cue that a variable token of type *elem* is required. We use an intelligent guessing technique to introduce such implicit variables during the parsing phase of our algorithm.

Note that In case of the absence of this variable token, we run into a roadblock during the course of our parsing algorithm. The situation can be either of the following:

- There are holes (requiring a token of type *elem*) in the representation tree and tokens available in the unused tokens list but none of the tokens is able to satisfy this hole even after replication.
- There are holes (requiring a token of type *elem*) in the

representation tree and no tokens are available in the unused tokens list and none of the original tokens when replicated satisfy this hole.

In both these cases, we introduce a new handcrafted variable token of type *elem* and fill the hole with it. This enables the algorithm to proceed forward and build the required represention tree. The introduction of variables in case of assertion based questions is discussed in section IX-B

VIII. Handling Quantified statements

We encounter many problems that involve quantifiers in their logical representations. As expected, we encounter two types of quantification - universal quantification over all members of a domain and the existential quantification with some property satisfaction.

- Universal quantification : Usually the problems of interest involve checking the satisfiability of a predicate/property for all elements of a domain. The domain is often modelled as the set of elements satisfying some other predicate/property. Consequently, this gives us the general template for a basic *Forall* quantification scheme : $\forall x A(x) \rightarrow B(x)$ where $A(x)$ and $B(x)$ are sub-formulae of type boolean that according to our previous discussion in section V, successfully capture the notion of domain. This allows us to focus on this template only and ignore the other templates like $\forall x, A(x) \wedge B(x)$ which are not of interest.
- Existential quantification : In this case, the problems of interest involve the checking the existence of an element of the domain under consideration which satisfies certain property. As said previously, both the domain as well as the property under consideration is captured well by sub expressions of boolean type. This also gives us the template for existentially quantified statements : $\exists x, A(x) \wedge B(x)$ where $A(x)$ corresponds to the sub-formula capturing the domain while $B(x)$ captures the property under consideration.

Our handling of quantifications has the following assumptions:

- All quantification based queries quantify over only one variable
- There is no nesting of universal and existential quantifiers

These assumptions have been made after studying the examples in the dataset and extend well to real world problems in this domain.

An important problem encountered was the guessing of the quantification type involved in such questions. In certain cases, the occurences of direct cue-phrases like "all", "exists" helped in determining the nature of quantification involved. However, in some cases, such direct cues were not present. To handle this, we first make the following observation :

Observation : The truth value $\forall x, A(x) \rightarrow B(x)$ and $\exists x, A(x) \wedge B(x)$ *can* differ only if the cardinality of the set of elements satisfying the domain predicate $A(x)$ is not unity.

We found from our dataset that im most questions of this case, the domain gets satisfied by a single element and in that case, any of the quantifications will yield the same truth value. For questions with diferrent number of elements in the domain, we favoured the formation of Forall quantification over existential as we observed that existential queries usually had an occurence of a cue phrase in them.

We now discuss handling of forall quantification. The discussion for existential quantification is similar.

A. Universal Quantification

As mentioned, our template for handling universal quantification is $\forall x, A(x) \rightarrow B(x)$. Corresponding to this template, we define the logical representation in our logic as *Forall(quantified variable, Boolean sub-formula)*

Example: Alkali elements show metallic character.
a) True b) False

The Forall predicate takes as its arguments the variable being quantified and a boolean formula that has to be satisfied by that variable. The correct translation of the above example would be:

Forall($1,AlkaliMetal($1) $\Longrightarrow$ *Metallic($1))*

Handling Forall requires answering of two questions :

- Position of implication.
- Identification of domain (antecedent) and property (consequent) predicates.

We observed that the position of the implication sign normally occurs at the position corresponding to the **verb** in the sentence. In the above example the verb "*show*" demarcates the antecedent and the consequent in the implication. In order to find the verb in the sentence we used the Part of Speech tagger in the Stanford CoreNLP [10] package. The details of the integration of the package into our system is given in a Section XI.

Once we have the position of the implication, we need to decide on its direction. For example, *Forall($1,Metallic($1)* $\Longrightarrow$ *AlkaliMetal($1))* would be a wrong translation for the given question. We are also faced with the problem of reporting the same questions in active as well as passive voice which though, look different to the parser, have the same logical representation. The question

Example: Metallic character is shown by alkali metals.

has the same translation as the first example but is in passive voice. Hence, to resolve the antecedent and consequent we use the following strategy which is based on the observations made from the dataset:

The voice of the question was again determined by the Stanford CoreNLP package. However, note that this algorithm

Algorithm 3 Forall resolution Algorithm

```
1: A(x) = Subtree with predicates having cover to the left of the verb
2: B(x) = Subtree with predicates having cover to the right of the verb
3: if Sentence is in Active Voice then
4:     Construct implication in direction $A(x) \implies B(x)$
5: else
6:     Construct implication in direction $B(x) \implies A(x)$
7: end if
```

is still not bulletproof as shown in the following example :

Example: Group 1 elements are called alkali metals.

Logical representation :
ForAll($1, Implies(Same(1,Group($1)), AlkaliMetal($1)))

In this question, our algorithm incorrectly reports the logical representation as *ForAll($1,Implies(AlkaliMetal($1),Same(1,Group($1))))*. However, we find that such examples are rare.

IX. Assertion based questions

In the domain of periodic table chemistry, we come across a number of questions that assert facts about a particular entity in a couple of statements statements and then finally ask a question about some other fact/property of the same. Such questions normally span multiple sentences and involve quantification over the variables introduced. This category of problems make use of our automatic introduction of variables. However, the variables introduced over the different sentences, mostly, refer to each other and hence these questions typically present us with the problem of anaphora/coreference resolution.

A. Anaphora Resolution

Anaphora resolution refers to the problem of resolving what a pronoun or a noun phrase refers to in a particular sentence. Consider the following example:

Example: An element A forms covalent bond with oxygen. It has high electronegativity and belongs to group 13. What is its atomic number?

In the above example, the pronouns "it" in sentence 2 and "its" in sentence 3 refer to "element A" introduced in the first sentence. This is a well studied problem in Natural Language Processing and different methods appear in literature for solving this problem. We use the anaphora resolution system bundled with the Stanford CoreNLP package that implements the deterministic multi-pass sieve coreference resolution system [11] [12]. The tool returns a coreference graph giving the dependencies between the pronouns and the entities they refer to. The output for the above example would be:

Output: Coreference Resolution Graph

- sentence 1, word 2
- sentence 2, word 1
- sentence 3, word 3

B. Handling Multiple Sentences

In questions spanning multiple sentences, we first hypothesize that each statement is complete in itself, i.e. each statement corresponds to a logical formula of its own. These questions typically assert facts about some element(s) or property(s) in the initial statements and pose a question in the last statement. The last example illustrates this fact where facts are asserted about the "element A" and we are finally asked to find its atomic number. Our way of dealing with multiple sentences is the following:

1) Parse each sentence independently to form logical formula corresponding to tokens in that particular sentence while introducing free variables as discussed in section **??**.
2) Use the coreference resolution system to find coreference chains and "tie up" the free variables in different sentences referring to the same entity
3) Construct the final formula of the form $A_1(x) \wedge A_2(x) \cdots \wedge A_n(x)$, where $A_i(x)$ refers to the logical formula constructed from the i^{th} sentence.
4) Properly quantify over the free variables left. We make use of the observation discussed in Section VIII noting that these questions typically talk about a single element/entity and hence, an existential quantification over the *conjuncted* formula above will suffice

X. Implementation of Negations

In our dataset, we encountered negations in primarity the following formats :

- Appearance of *non-*
- Appearance of *not*
- Appearance of *no*

These three *templates* of appearance of negation corresponds to different strategies in our algorithm as explained next.

- To handle negation in form of *non-*, we observe that such occurences try to negate the immediate predicate occurring next to it and hence couple it directly to that token in tier-1 of the parsing phase.

  Example : Which of the non-metal is a gas at STP?
  Formula : And(IsGasAtSTP($1), Not(Metallic($1)))

- To handle negation in the form of *not*, we make use of the fact that in natural language, the human perception of *not* is just a token that has got some scope in the sentence which it negates. We observe that usually, the scope of such an appearance of *not* covers the entirety of the right of the poition of the token *not* in the sentence and we use this observation and verify the placement of the "Not()' predicate during the parsing.

Example : Alkali metals are not bad conductors of electricity
Formula : ForAll($1, Implies(AlkaliMetal($1), Not(Low(Conductivity($1)))

Example : Not all alkali metals form basic oxides
Formula : Not(ForAll($1, Implies(AlkaliMetal($1), BasicOxide(x)))

- To handle negation in the form of *no*, we use the natural interpretation of "no" as "there does not exist". Afterwards we form the logical representation based on rules for "not" and existential quantification.
  Example : No alkali metal is a gas at STP *Formula :* Not(Exists($1, And(AlkaliMetal($1), IsGasAtSTP($1)))

XI. Integration of Stanford CoreNLP tools

The need for Part of Speech tagging, corefence analysis and active/passive classification called for the use of external NLP packages. The Stanford CoreNLP package is one of the most widely used tools in the NLP community for basic tasks like POS tagging, deep parsing, Named entity recognition, corefence analysis etc. The tool is written in Java and is available for free.

A big difficulty in integrating the CoreNLP package with our code was the interoperability of C# and Java. Owing to the discontinuation of Microsoft's support to the development of Visual J#, we were forced to use the package as a command line black box tool. However, such intercommunication using files had an added disadvantage of availability as at a time only a single user could then use the tool. A further problem arose that on every run, the Stanford NLP tool had to load 2GB of trained models into memory for the tasks and thus took a lot of time (around 17 seconds per question). This method was highly inefficient and we had to come up with a new solution for using this tool. We finally ended up using the online demo of the tool by querying it from within our code and reading the XML response. This reduced the time required to the tune of 2 sec per question as the online demo has a running service with all models preloaded.

XII. Ranking Algorithm

The basic idea behind our algorithm is to arrange the discovered predicates in a type-safe manner. This might result in multiple representation trees. Thus, we need a method to rank various representation trees on the basis of their features and steps followed during their construction. Following are some of the heuristics we use to rank the trees:

- An important heuristic used for ranking the completed trees involves the *cover* of the logical formula on the question. We define the *cover* of a leaf node as the span of its cue word in the question. The *cover* of an internal node is the union of the cover of all its children. The greater the cover of the tree, the more likely it is to be the correct representation.
- We assign higher confidence to filling a hole with a token that is closer to the parent of the hole being filled in the question.
- We penalize a tree when it involves replication of tokens during its construction.
- We penalize a tree when we introduce a hand-crafted token (e.g. And, Or, Implies) during its construction.
- While reproducing tokens in the algorithm, the penalization factor is proportional to the size of the token (compound or simple) being reproduced as it makes less sense to reproduce larger trees.
- We penalize a complete tree when there are unused tokens still remaining at the termination of the algorithm. The penalization factor is proportional to the proportion of unused tokens remaining. This ensures that parses that utilize more tokens appearing in the question are ranked higher.

XIII. Examples

The following examples illustrate the diversity of problems currently handled by our system.

- **Question**: Which element is in Group 15 and has the strongest metallic character?
  a) N b) P c) Sb d) Bi
  Formula: Max(MetallicProperty, Same(Group($1),15))

- **Question** : Which element in period 3 has highest atomic radius and minimum metallic character?
  a) Na b) Mg c) Al d) Si
  Formula :
  And(Max(AtomicRadiusProperty, Same(Period($1), 3)), Min(MetallicProperty, Same(Period($1), 3)))

- **Question**: As we go right in a period, atomic radius _______:
  a) Decreases b) Increases c) Stay the same d) Can't say
  Formula: Trend(Right, AtomicRadiusProperty, $1)

- **Question**: What is the atomic number of Calcium?
  a) 10 b) 20 c) 30 d) 40
  Formula: Same($1, AtomicNumber(Ca))

- **Question**: Arrange the following in increasing order of Atomic radius.
  a) Be $<$ B $<$ C b) B $<$ C $<$ Be c) C $<$ Be $<$ B
  d) Be $<$ C $<$ B
  Formula: Order(AtomicRadiusProperty, Increase, $1)

- **Question**: Group 17 elements are gases at STP.
  a) True b) B False

Formula: ForAll($1,Implies(Same(17,Group($1)), GasAtSTP($1)))

- **Question**: An element X belongs to group 3. It has high ionisation energy. What is its atomic number?
  a) 5 b) 13 Be c) 31 d) 49
  Formula:
  And(And(High(FirstIonisationEnergy($1)), Same($2,AtomicNumber($1))),Same(3,Group($1)))

- **Question**: Noble gas elements form ionic bonds with oxygen.
  a) True b) B False
  Formula: ForAll($1,Implies(NobleGas($1), IonicBond($1,O)))

We demonstrate a run of algorithm on the question in Figure 3. The lexer processes the sentence and produces the following terms - *Group(), 2, Max()* and *MetallicProperty*. These terms are stored in a data structure alongwith their position and confidence data. Next, we pass the options and infer that the domain variable in this problem is of type *element*. This results in the addition of *$1* to the term list. The first tier of the parser takes as input these terms and tries to assemble some of them into compound tokens based on their metadata. Following our heuristic of associating numbers with numeric predicates based on proximity, we get the following compound token: *Same(Group(Hole),2)*. Note that *Same()* takes two inputs of numeric type and the output type of *Group()* is numeric. Thus, this compound token is type-consistent. This stage also converts the terms that couldn't be assembled into simple tokens. We finally come to the second tier of the parser which takes as input the simple and compound tokens from the previous stage. As our goal is to produce a formula that we evaluate to be true or false after substituting every option, we start with a hole of type boolean. The simple token *Max(Hole,Hole)* satisfies this hole and we recurse with the partial tree as *Max(Hole,Hole)* and unused tokens list as {*MetallicProperty,Same(Group(Hole),2)*}, *$1*. We then try to fill the first hole of the partial tree which takes a token of type *Numeric*. Thus, *MetallicProperty* satisfies this hole and we recurse again with partial tree as *Max(MetallicProperty,Hole)* and unused tokens list as {*Same(Group(Hole),2), $1*}. The algorithm now takes the decision of filling the hole with *Same(Group(Hole),2)* resulting in the partial tree as *Max(MetallicProperty,Same(Group(Hole),2))* and the only unused token as *$1*. In the final step, *$1* is filled in the hole of the tree resulting in the tree *Max(MetallicProperty,Same(Group($1),2))*. This is a representation tree with no holes and is returned by the algorithm as one of the possible candidates for the logical representation corresponding to the question.

XIV. Statistical Evaluation

One of the evaluation metrics could be the ratio of the number of rules encoded to the corpus size of problems solved. We encode 173 predicates/entities/functions in our algorithm out of which 118 are names of elements. We have currently evaluated our algorithm on a set of 126 problems obtained from the Tata McGraw Hill textbook for Grade XI. Our algorithm is currently able to solve 70 out of the 126 problems. The major reason for not being able to solve some of the problems is the absence of modelling of certain chemistry-specific predicates. This just corresponds to adding domain knowledge to our system. As we can see, by adding a minimal number of predicates (around 55, most of them being chemistry-specific), we have been able to solve 55% of the problems in our dataset. It is to be noted that this dataset is quite small and contains a wide variety of questions spanning various domains in periodic table chemistry and a number of concepts too. We believe that our algorithm would perform reasonably well on a larger dataset representative of questions faced by students in the real world scenario.

XV. Future Work

There are several challenges remaining to be tackled in the further stages of development of this system. We need to disambiguate certain cue words that might correspond to different tokens. For example, 'At' might correspond to the element Astatine, 'In' might correspond to the element Indium or its English meaning. We also need to infer various representations of numbers like 'First', '1^{st}' and '1' as the same. Similar disambiguation of 's' and 'p' with respect to element, block and orbital is also required. One way to tackle the above problems might be to use the Named Entity Recognition tool in the CoreNLP package.

Our current permutation removal algorithm fails to deal with nested conjunctions, i.e. *And(And(x,y),z)* and *And(And(x,z),y)*. One way to remove this would be to model the commutative property of *And* and *Or* in our algorithm. Furthermore, Certain properties like electronic configuration and reactions remain to be modelled. We need a better modelling of conjunctions to be able to solve questions of the form "Alkali metals belong to group 1 and are metallic in nature". Finally, the ultimate goal of this project would be to generate explanations for the solutions in natural language and paraphrasing of explanations would be an interesting problem to explore.

XVI. Conclusion

Natural language is very rich and at the same time often ambiguous. Subtle difference which are invisible/inconsequential to the machine can have dramatic effects on the meaning. For a long time linguists have believed in the understanding of natural language in the type-theoretic framework where portions of sentences have types and scopes. In this project, we tried to develop a system that analyzed the natural language using the type-theoretic model and tried to parse the sentences to a logical representation. While contemporary works focus on analyzing languages by learning, we hypothesize that for a simpler structured domain like Chemistry, a much simpler type-theoretic approach armed with some heuristics observed from the domain can achieve similar, if not better, success. During the later phase of the project, we tried to use some techniques of learning to improve upon our system and were successful in doing so. In conclusion, we feel that a combination of such a type-theoretic approach and the standard machine learning techniques can achieve good success for a well structured domain like Chemistry.

References

[1] L. R. Tang and R. J. Mooney, "Using multiple clause constructors in inductive logic programming for semantic parsing," in *In Proceedings of the 12th European Conference on Machine Learning*, 2001, pp. 466–477.

[2] R. Ge and R. J. Mooney, "A statistical semantic parser that integrates syntax and semantics," in *Proceedings of the Ninth Conference on Computational Natural Language Learning*, ser. CONLL '05. Stroudsburg, PA, USA: Association for Computational Linguistics, 2005, pp. 9–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1706543.1706546

[3] Y. W. Wong and R. J. Mooney, "Learning for semantic parsing with statistical machine translation," 2006.

[4] R. J. Kate, "Using string-kernels for learning semantic parsers," in *In Proc. of COLING/ACL-06*, 2006, pp. 913–920.

[5] K. Barker, V. K. Chaudhri, S. Y. Chaw, P. Clark, J. Fan, D. J. Israel, S. Mishra, B. W. Porter, P. Romero, D. Tecuci, and P. Z. Yeh, "A question-answering system for ap chemistry: Assessing kr&r technologies," in *KR*, 2004, pp. 488–497.

[6] G. S. Novak and A. A. Araya, "Research on expert problem solving in physics," in *AAAI*, 1980, pp. 178–180.

[7] A. Bundy, "Mecho: A program to solve mechanics problems," 1979.

[8] S.-Y. Jung and K. VanLehn, "Developing an intelligent tutoring system using natural language for knowledge representation," in *Intelligent Tutoring Systems (2)*, 2010, pp. 355–358.

[9] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, pp. 31–88, March 2001. [Online]. Available: http://doi.acm.org/10.1145/375360.375365

[10] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, ser. NAACL '03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 173–180. [Online]. Available: http://dx.doi.org/10.3115/1073445.1073478

[11] H. Lee, Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky, "Stanford's multi-pass sieve coreference resolution system at the conll-2011 shared task," in *Proceedings of the CoNLL-2011 Shared Task*, 2011.

[12] K. Raghunathan, H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. Manning, "A multi-pass sieve for coreference resolution," 2010.