Automatically Solving Number Word Problems by Semantic Parsing and Reasoning

Shuming Shi[1], Yuehui Wang[2*], Chin-Yew Lin[1], Xiaojiang Liu[1] and Yong Rui[1]
[1] Microsoft Research
{shumings, cyl, xiaojl, yongrui}@microsoft.com
[2] University of Science and Technology of China
wyh9346@mail.ustc.edu.cn

Abstract

This paper presents a semantic parsing and reasoning approach to automatically solving math word problems. A new meaning representation language is designed to bridge natural language text and math expressions. A CFG parser is implemented based on 9,600 semi-automatically created grammar rules. We conduct experiments on a test set of over 1,500 number word problems (i.e., verbally expressed number problems) and yield 95.4% precision and 60.2% recall.

1 Introduction

Computers, since their creation, have exceeded human beings in (speed and accuracy of) mathematical calculation. However, it is still a big challenge nowadays to design algorithms to automatically solve even primary-school-level math word problems (i.e., math problems described in natural language).

Efforts to automatically solve math word problems date back to the 1960s (Bobrow, 1964a, b). Previous work on this topic falls into two categories: symbolic approaches and statistical learning methods. In symbolic approaches (Bobrow, 1964a, b; Charniak, 1968; Bakman, 2007; Liguda & Pfeiffer, 2012), math problem sentences are transformed to certain structures by pattern matching or verb categorization. Equations are then derived from the structures. Statistical learning methods are employed in two recent papers (Kushman et al., 2014; Hosseini et al., 2014).

Most (if not all) previous symbolic approaches suffer from two major shortcomings. First, natural language (NL) sentences are processed by simply applying pattern matching and/or transformation rules in an ad-hoc manner (refer to the related work section for more details). Second, surprisingly, they seldom report evaluation results about the effectiveness of the methods (except for some examples for demonstration purposes). For the small percentage of work with evaluation results available, it is unclear whether the patterns and rules are specially designed for specific sentences in a test set.

1). One number is 16 more than another. If the smaller number is subtracted from 2/3 of the larger, the result is 1/4 of the sum of the two numbers. Find the numbers.

2). Nine plus the sum of an even integer and its square is 3 raised to the power of 4. What is the number?

3). The tens digit of a two-digit number is 3 more than the units digit. If the number is 8 more than 6 times the sum of the digits, find the number.

4). If the first and third of three consecutive even integers are added, the result is 12 less than three times the second integer. Find the integers.

Figure 1: Number word problem examples

In this paper, we present a computer system called SigmaDolphin which automatically solves math word problems by semantic parsing and reasoning. We design a meaning representation language called DOL (abbreviation of dolphin language) as the structured semantic representation of NL text. A semantic parser is implemented to transform math problem text into DOL trees. A reasoning module is included to derive math expressions from DOL trees and to calculate final answers. Our approach falls into the symbolic category, but makes improvements over previous symbolic methods in the following ways,

* Work done while this author was an intern at Microsoft Research

1) We introduce a systematic way of parsing NL text, based on context-free grammar (CFG).

2) Evaluation is enhanced in terms of both data set construction and evaluation mechanisms. We split the problem set into a development set (called dev set) and a test set. Only the dev set is accessible during our algorithm design (especially in designing CFG rules and in implementing the parsing algorithm), which avoids over-tuning towards the test set. Three metrics (precision, recall, and F1) are employed to measure system performance from multiple perspectives, in contrast to all previous work (including the statistical ones) which only measures accuracy.

We target, in experiments, a subtype of word problems: number word problems (i.e., verbally expressed number problems, as shown in Figure 1). We hope to extend our techniques to handle general math word problems in the future.

We build a test set of over 1,500 problems and make a quantitative comparison with state-of-the-art statistical methods. Evaluation results show that our approach significantly outperforms baseline methods on our test set. Our system yields an extremely high precision of 95.4% and a reasonable recall of 60.2%, which shows promising application of our system in precision-critical situations.

2 Related Work

2.1 Math word problem solving

Most previous work on automatic word problem solving is symbolic. STUDENT (Bobrow, 1964a, b) handles algebraic problems by first transforming NL sentences into *kernel sentences* using a small set of transformation patterns. The kernel sentences are then transformed to math expressions by recursive use of pattern matching. CARPS (Charniak, 1968, 1969) uses a similar approach to solve English rate problems. The major difference is the introduction of a tree structure as the internal representation of the information gathered for one object. Liguda & Pfeiffer (2012) propose modeling math word problems with augmented semantic networks. Addition/subtraction problems are studied most in early research (Briars & Larkin, 1984; Fletcher, 1985; Dellarosa, 1986; Bakman, 2007; Ma et al., 2010). Please refer to Mukherjee & Garain (2008) for a review of symbolic approaches before 2008.

No empirical evaluation results are reported in most of the above work. Almost all of these approaches parse NL text by simply applying pattern matching rules in an ad-hoc manner. For example, as mentioned in Bobrow (1964b), due to the pattern "($, AND $)", the system would incorrectly divide "Tom has 2 apples, 3 bananas, and 4 pears." into two "sentences": "Tom has 2 apples, 3 bananas." and "4 pears."

WolframAlpha[1] shows some examples[2] of automatically solving elementary math word problems, with technique details unknown to the general public. Other examples on the web site demonstrate a large coverage of short phrase queries on math and other domains. By randomly selecting problems from our dataset and manually testing on their web site, we find that it fails to handle most problems in our problem collection.

Statistical learning methods have been proposed recently in two papers: Hosseini et al. (2014) solve single step or multi-step homogenous addition and subtraction problems by learning verb categories from the training data. Kushman et al. (2014) can solve a wide range of word problems, given that the equation systems and solutions are attached to problems in the training set. The method of the latter paper (referred to as KAZB henceforth) is used as one of our baselines.

2.2 Semantic parsing

There has been much work on analyzing the semantic structure of NL strings. In semantic role labeling and frame-semantic parsing (Gildea & Jurafsky, 2002; Carreras & Marquez, 2004; Marquez et al., 2008; Baker et al., 2007; Das et al., 2014), predicate-argument structures are discovered from text as their shallow semantic representation. In math problem solving, we need a deeper and richer semantic representation from which to facilitate the deriving of math expressions.

Another type of semantic parsing work (Zelle & Mooney, 1996; Zettlemoyer & Collins, 2005; Zettlemoyer & Collins, 2007; Wong & Mooney, 2007; Cai & Yates, 2013; Berant et al., 2013; Kwiatkowski et al., 2013; Berant & Liang, 2014) maps NL text into logical forms by supervised or semi-supervised learning. Some of them are based on or related to combinatory categorial grammar (CCG) (Steedman, 2000). Abstract Meaning Representation (AMR) (Banarescu et al., 2013) keeps richer semantic information than CCG and logical

[1] http://www.wolframalpha.com

[2] https://www.wolframalpha.com/examples/Elementary-Math.html (bottom-right part)

forms. In Section 3.1.4, we discuss the differences between DOL, AMR, and CCG, and explain why we choose DOL as the meaning representation language for math problem solving.

3 Approach

Consider the first problem in Figure 1 (written below for convenience),

One number is 16 more than another. If the smaller number is subtracted from 2/3 of the larger, the result is 1/4 of the sum of the two numbers. Find the numbers.

To automatically solve this problem, the computer system needs to figure out, somehow, that 1) two numbers x, y are demanded, and 2) they satisfy the equations below,

$$x = 16 + y \qquad (1)$$

$$(2/3)x - y = (x + y) / 4 \qquad (2)$$

To achieve this, *reasoning* must be performed based on common sense knowledge and the information provided by the source problem. Given the difficulty of performing reasoning directly on unstructured and ambiguous natural language text, it is reasonable to transform the source text into a structured, less ambiguous representation.

Our approach contains three modules:

1) A *meaning representation language* called DOL newly designed by us as the semantic representation of natural language text.
2) A *semantic parser* which transforms natural language sentences of a math problem into DOL representation.
3) A *reasoning* module to derive math expressions from DOL representation.

English: Nine plus the sum of an even integer and its square is 3 raised to the power of 4. What is the number?

DOL trees:

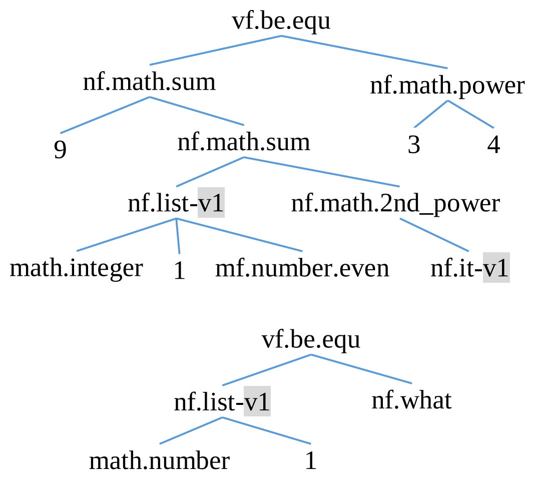

Figure 2: DOL example

3.1 DOL: Meaning representation language

Every meaningful piece of NL text is represented in DOL as a semantic tree of various node types. Figure 2 shows the DOL representation of the second problem of Figure 1. It contains two semantic trees, corresponding to the two sentences.

3.1.1 Node types

Node types of a DOL tree include *constants*, *classes*, and *functions*. Each interim node of a tree is always a function; and each leaf node can be a constant, a class, or a zero-argument function.

Constants in DOL refer to specific objects in the world. A constant can be a number (e.g., 3.57), a lexical string (like "New York"), or an entity.

Classes: An entity class refers to a category of entities sharing common semantic properties. For example, all cities are represented by the class *location.city*; and *math.number* is a class for all numbers. It is clear that,

3.14159 ∈ math.number
city.new_york ∈ location.city

A class C_1 is a *sub-class* (denoted by ⊆) of another class C_2 if and only if every instance of C_1 are in C_2. The following holds according to common sense knowledge,

math.number ⊆ math.expression
person.pianist ⊆ person.performer

Template classes are classes with one or more parameters, just like template classes in C++. The most important template class in DOL is

t.list<*c*,*m*,*n*>

where *c* is a class; *m* and *n* are integers. Each instance of this class is a list containing at least *m* and at most *n* elements of type *c*. For example, each instance of t.list<math.number,2,+∞> is a list containing at least 2 numbers.

Functions are used in DOL as the major way to form larger language units from smaller ones. A function is comprised of a *name*, a list of *core arguments*, and a *return type*. DOL enables function overloading (again borrowing ideas from programming languages). That is, one function name can have multiple core-argument specifications. Below are two specifications for fn.math.sum (which appears in the example of Figure 2).

nf.math.sum!1:
$1: math.expression; $2: math.expression
return type: math.expression
return value: The sum of its arguments

nf.math.sum!2:
$1: t.list<math.expression,2,+∞>
return type: math.expression
return value: The sum of the elements in $1

Here "$1: math.expression" means the first argument has type math.expression.

DOL supports three kinds of functions: noun functions, verb functions, and modifier functions.

Noun functions map entities to their properties or to other entities having specific relations with the argument(s). For example, *nf.math.sum* maps math expressions to their sum. Noun functions are used to represent noun phrases in natural language text. More noun functions are shown in Table 1.

Among all noun functions, *nf.list* has a special important position due to its high frequency in DOL trees. The function is specified below,

```
nf.list
    $1: class;  $2: math.number
    return type: t.list<$1>
    return value: An entity list with cardinality $2
and element type $1
```

For example nf.list(math.number,5) returns a list containing 5 elements of type math.number. It is the semantic representation of "five numbers".

Pronoun functions are special zero-argument noun functions. Examples are nf.it (representing an already-mentioned entity or event) and nf.what (denoting an unknown entity or entity list).

Verb functions act as sentences or sub-sentences in DOL. As an example, vf.be.equ (in Figure 2) is a verb function that has two arguments of the quantity type.

```
vf.be.equ
    $1: quantity.generic;  $2: quantity.generic
    return type: t.vf
    Meaning: Two quantities $1 and $2 have the
same value
```

In addition to core arguments ($1, $2, etc.), many functions can take additional *extended arguments* as their modifiers. Our last function type called *modifier functions* often take the role of extended arguments, to modify noun functions, verb functions, or other modifier functions. Modifier functions are used in DOL as the semantic representation of adjectives, adverb phrases (including conjunctive adverb phrases), and prepositional phrases in natural languages. In the example of Figure 2, the function mf.number.even modifies the noun function nf.list as its extended argument.

3.1.2 Entity variables

Variables are assigned to DOL sub-trees for indicating the co-reference of sub-trees to entities and for facilitating the construction of logical forms and math expressions from DOL. In Figure 2, the same variable v1 (meaning a variable with ID 1) is assigned to two sub-trees in the first sentence and one sub-tree in the second sentence. Thus the three sub-trees refer to the same entity.

Function	**Remarks**
nf.math.numerator $1: math.fraction ret: math.number	Get the numerator of fraction $1
nf.math.gcd $1: t.list<math.integer,2,+∞> ret: math.integer	Get the greatest common divisor of the elements of $1
nf.e.height $1: e.concrete ret: quantity.length	Get the height of $1 which is a concrete entity
vf.believe $1: e.agent; $2: t.vf.std ret: t.vf	Agent $1 believes that $2 is true as a predicate
mf.number.even ret: t.mf.adj	Indicating the property of being an even number

Table 1: Example DOL functions

3.1.3 Key features of DOL

DOL has some nice characteristics that are critical to building a high-precision math problem solving system. That is why we invent DOL as our meaning representation language instead of employing an existing one.

First, DOL is a strongly typed language. Every function has clearly defined argument types and a return type. A valid DOL tree must satisfy the *type-compatibility* property:

Type-compatibility: The type of each child of a function node should match the corresponding argument type of the function.

For example, in Figure 2, the return type of nf.math.power is math.expression, which matches the second argument of vf.be.equ. However, the following two trees (yielded from the corresponding pieces of text) are invalid because they do not satisfy type-compatibility.

sum of 100 [unreasonable text]
~~nf.math.sum!2(100)~~ [invalid DOL tree]

sum of 3 and Jordan [unreasonable text]
~~nf.math.sum!2({3, "Jordan"})~~ [invalid tree]

Second, we maintain in DOL an open-domain type system. The type system contains over 1000 manually verified classes and more automatically generated ones (refer to Section 3.2.1 for more details). Such a comprehensive type system makes it possible to define various kinds of functions and to perform type-compatibility checking. In contrast, most previous semantic languages have at most 100+ types at the grammar level. In addition, by introducing template classes, we avoid maintaining a lot of potentially duplicate types and reduce the type system management efforts. To the best of our knowledge, template classes are not available in other semantic representation languages.

Third, DOL has built-in data structures like t.list and nf.list which greatly facilitate both function declaration and text representation (especially math text representation). For example, the two variants of nf.math.sum (refer to Section 3.1.1 for their specifications) are enough to represent the following English phrases:

3 plus 5 → nf.math.sum!1(3, 5)
sum of 3, 5, 7, and 9 → nf.math.sum!2(nf.list(3, 5, 7, 9))
sum of ten thousand numbers → nf.math.sum!2(nf.list(math.number,10000))

Without t.list or nf.list, we would have to define a lot of overloaded functions for nf.math.sum to deal with different numbers of addends.

3.1.4 Comparing with other languages

Among all meaning representation languages, AMR (Banarescu et al., 2013) is most similar to DOL. Their major differences are: First, they use very different mechanisms to represent noun phrases. In AMR, a sentence (e.g., "the boy destroyed the room") and a noun phrase (e.g., "the boy's destruction of the room") can have the same representation. While in DOL, a sentence is always represented by a verb function; and a noun phrase is always a noun function or a constant. Second, DOL has a larger type system and is stricter in type compatibility checking. Third, DOL has template classes and built-in data structures like t.list and nf.list to facilitate the representation of math concepts.

CCG (Steedman, 2000) provides a transparent interface between syntax and semantics. In CCG, semantic information is defined on words (e.g., "λx.odd(x)" for "odd" and "λx.number(x)" for "number"). In contrast, DOL explicitly connects NL text patterns to semantic elements. For example, as shown in Table 2 (Section 3.2.1), one CFG grammar rule connects pattern "{$1} raised to the power of {$2}" to function nf.math.power.

Logical forms are another way of meaning representation. We choose not to transform NL text directly to logical forms for two reasons: On one hand, state-of-the-art methods for mapping NL text into logical forms typically target short, one-sentence queries in restricted domains. However, many math word problems are long and contain multiple sentences. On the other hand, variable-id assignment is a big issue in direct logical form construction for many math problems. Let's use the following problem (i.e., the first problem of Figure 1) to illustrate,

One number is 16 more than another. If the smaller number is subtracted from 2/3 of the larger, the result is 1/4 of the sum of the two numbers. Find the numbers.

For this problem, it is difficult to determine whether "the smaller number" refers to "one number" or "another" in directly constructing logical forms. It is therefore a challenge to construct a correct logical form for such kinds of problems.

Our solution to the above challenge is assigning a new variable ID (which is different from the IDs of "one number" and "another") and to delay the final variable-ID assignment to the reasoning stage. To enable this mechanism, the meaning representation language should support a lazy variable ID assignment and keep as much information (e.g., determiners, plurals, modifiers) from the noun phrases as possible. DOL is a language that always keeps the structure information of phrases, whether or not it has been assigned a variable ID.

In summary, compared with other languages, DOL has some unique features which make it more suitable for our math problem solving scenario.

3.2 Semantic Parsing

Our parsing algorithm is based on context-free grammar (CFG) (Chomsky, 1956; Backus, 1959; Jurafsky & Martin, 2000), a commonly used mathematical system for modeling constituent structure in natural languages.

3.2.1 CFG for connecting DOL and NL

The core part of a CFG is the set of grammar rules. Example English grammar rules for building syntactic parsers include "S → NP VP", "NP → CD | DT NN | NP PP", etc. Table 2 shows some example CFG rules in our system for mapping DOL nodes to natural language word sequences. The left side of each rule is a DOL element (a function, class, or constant); and the right side is a sequence of words and arguments. The grammar rules are consumed by our parser for building DOL trees from NL text.

So far there are 9,600 grammar rules in our system. For every DOL node type, the lexicon and grammar rules are constructed together in a semi-automatic way. Math-related classes, functions, and constants and their grammar rules are manually built by referring to text books and online tutorials. About 35 classes and 200 functions are obtained in this way. Additional instances of each element type are constructed in the ways below.

Classes: Additional classes and grammar rules are obtained from two data sources: Freebase[3] types, and automatically extracted lexical semantic data. By treating Freebase types as DOL classes and the mapping from types to lexical names as grammar rules, we get the first version of grammar for classes. To improve coverage, we run a term peer similarity and hypernym extraction algorithm (Hearst, 1992; Shi et al., 2010; Zhang et al., 2011) on a web snapshot of 3 billion pages, and get a peer-similarity graph and a collection of is-a pairs. An is-a pair example is (Megan Fox, actress), where "Megan Fox" and "actress" are instance and type names respectively. In our peer similarity graph, "Megan Fox" and "Britney Spears" have a high similarity score. The peer similarity graph is used to clean the is-a data collection (with the idea that peer terms often share some common type names). Given the cleaned is-a data, we sort the type names by weight and manually create classes for top-1000 type names. For example, create a class person.actress and add a grammar rule "person.actress → actress". For the other 2000 type names in the top 3000, we create classes and rules automatically, in the form of "class.TN → TN", where TN is a type name. For example, create rule "class.succulent → succulent" for name "succulent".

vf.be.equ($1,$2) →	{$1} be equal to {$2}
	| {$1} equal {$2}
	| {$1} be {$2}
vf.give($1,$2,$3) →	{$1} give {$2} to {$3}
	| {$1} give {$3} {$2}
nf.math.sum!1($1,$2) →	{$1} plus {$2}
	| {$2} added to {$1}
nf.math.sum!2($1) →	sum of {$1}
	| addition of {$1}
nf.math.power($1,$2) →	{$1} raised to the {power|exponent} of {$2}
nf.list($1,$2) →	{$2} {$1}
mf.number.even →	even
mf.condition.if($1) →	if {$1}
mf.approximately →	approximately
	| roughly
education.university →	university
math.number →	number
math.integer →	integer

Table 2: Example grammar for connecting DOL and NL

Functions: Additional noun functions are automatically created from Freebase properties and attribute extraction results (Pasca et al., 2006; Durme et al., 2008), using a similar procedure with creating classes from Freebase types and is-a extraction results. We have over 50 manually defined math-related verb functions. Our future plan is automatically generating verb functions from databases like PropBank (Kingsbury & Palmer, 2002), FrameNet (Fillmore et al., 2003), and VerbNet[4] (Schuler, 2005). Additional modifier functions are automatically created from an English adjective and adverb list, in the form of "mf.adj.TN → TN" and "mf.adv.TN → TN" where TN is the name of an adjective or adverb.

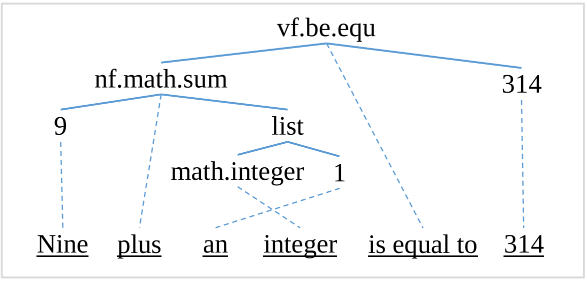

Figure 3: The DOL semantic parse tree for "Nine plus an integer is equal to 314"

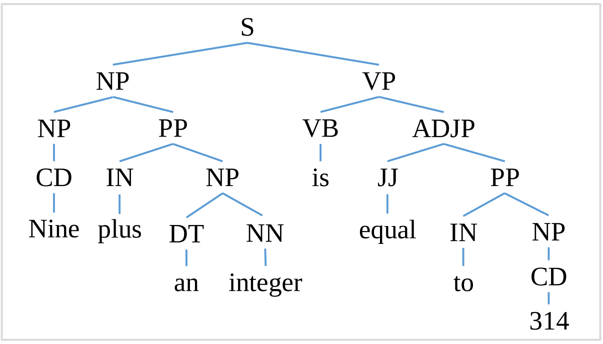

Figure 4: A syntactic parse tree

3.2.2 Parsing

Parsing for CFG is a well-studied topic with lots of algorithms invented (Kasami, 1965; Earley, 1970). The core idea behind almost all the algorithms is exploiting dynamic programming to achieve efficient search through the space of possible parse trees. For syntactic parsing, a well-known serious problem is ambiguity: the appearance of many syntactically correct but semantically unreasonable parse trees. Modern syntactic parsers reply on statistical information to reduce

[3] Freebase: http://www.freebase.com/

[4] VerbNet: http://verbs.colorado.edu/~mpalmer/projects/verbnet.html

ambiguity. They are often based on probabilistic CFGs (PCFGs) or probabilistic lexicalized CFGs trained on hand-labeled TreeBanks.

With the new set of DOL-NL grammar rules (examples in Table 2) and the type-compatibility property (Section 3.1.3), ambiguity can hopefully be greatly reduced, because semantically unreasonable parsing often results in invalid DOL trees.

We implement a top-down parser for our new CFG of Section 3.2.1, following the Earley algorithm (Earley, 1970). No probabilistic information is attached in the grammar rules because no Treebanks are available for learning statistical probabilities for the new CFG. Figure 3 shows the parse tree returned by our parser when processing a simple sentence. The DOL tree can be obtained by removing the dotted lines (corresponding to the non-argument part in the right side of the grammar rules). A traditional syntactic parse tree is shown in Figure 4 for reference.

During parsing, a score is calculated for each DOL node. The score of a tree T is the weighted average of the scores of its sub-trees,

$$\boldsymbol{S}(\boldsymbol{T}) = \frac{\sum_{i=1}^{k} \boldsymbol{L}(\boldsymbol{T_i}) \cdot \boldsymbol{S}(\boldsymbol{T_i})}{\sum_{i=1}^{k} \boldsymbol{L}(\boldsymbol{T_i})} \cdot \boldsymbol{p}(\boldsymbol{T}) \qquad (3)$$

where T_i is a sub-tree, and $L(T_i)$ is the number of words to which the sub-tree corresponds in the original text. If the type-compatibility property for T is satisfied, $p(T)$=1; otherwise $p(T)$=0.

All leaf nodes are assigned a score of 1.0, except for pure lexical string nodes (which are used as named entity names). The score of a lexical string node is set to $1/(1+\mu n)$, where n is the number of words in the node, and μ (=0.2 in experiments) is a parameter whose value does not have much impact on parsing results. Such a score function encourages interpreting a word sequence with our grammar than treating it as an entity name.

Among all candidate DOL trees yielded during parsing, we return the one with the highest score as the final parsing result. A null tree is returned if the highest score is zero.

3.3 Reasoning

The reasoning module is responsible for deriving *math expressions* from DOL trees and calculating problem answers by solving equation systems. Math expressions have different definitions in different contexts. In some definitions, equations and inequations are excluded from math expressions. In this paper, equations and inequations (like "a=b" and "ax+b>0") are called s-expressions because they represent mathematical sentences, while other math expressions (like "x+5") are named n-expressions since they are essentially noun phrases. Our definition of "*math expressions*" therefore includes both n-expressions and s-expressions.

Different types of nodes may generate different types of math expressions. In most cases, s-expressions are derived from verb function nodes and modifier function nodes, while n-expressions are generated from constants and noun function nodes. For example, the s-expression "9+x=314" can be derived from the DOL tree of Figure 3, if variable x represents the integer. In the same Figure, The n-expression "9+x" is derived from the left sub-tree.

The pseudo-codes of our math expression derivation algorithm are shown in Figure 5. The algorithm generates the math expression for a DOL tree T by first calling the expression derivation procedure of sub-trees, and then applying the semantic interpretation of T. All the s-expressions derived so far are stored in an expression list named XL.

```
Algorithm MathExpDerivation
  Input: DOL tree T
  Output: Math expression X(T)
  Global data structure: Expression list XL
1: For each child $C_i$ of T
2:     $X(C_i)$ = MathExpDerivation($C_i$)
3:     If X(Ci) is an s-expression
4:         Add $X(C_i)$ to XL
5: X(T) ← Applying the semantic interpretation
of T
6: Return X(T)
```

Figure 5: Math expression derivation algorithm

vf.be.equ($1,$2) → X($1) = X($2)	(1)
nf.math.sum!1($1,$2) → X($1) + X($2)	(2)
nf.math.sum!2($1) → $\sum_{\mathbf{e}\in\$\mathbf{1}} \mathbf{X}(\mathbf{e})$	(3)
nf.math.gcd($1) → gcd({X(e) | **e** ∈ $**1**})	(4)
nf.list($1,$2) → V = (v_1, v_2…, v_n), n=X($2)	(5)
mf.number.even → X($↑) % 2 = 0	(6)

Table 3: Example semantic interpretations

The semantic interpretation of DOL nodes plays a critical role in the algorithm. Table 3 shows some example interpretations of some representative DOL functions. In the table, $1, $2 etc. are function arguments, and $↑ for a modifier node denotes the node which the modifier modifies. So far the semantic interpretations are built manually. Please note that it is not necessary to make semantic interpretations for every DOL node in solving number word problems. For example, most class nodes and many adverb nodes can have null interpretations at the moment.

4 Experiments

4.1 Experimental setup

Datasets: Our problem collection[5] contains 1,878 math number word problems, collected from two web sites: algebra.com[6] (a web site for users to post math problems and get help from tutors) and answers.yahoo.com[7]. Problems on both sites are organized into categories. For algebra.com, problems are randomly sampled from the number word problems category; for answers.yahoo.com, we first randomly sample an initial set of problems from the math category and then ask human annotators to manually choose number word problems from them. Math equations[8] and answers to the problems are manually added by human annotators.

We randomly split the dataset into a dev set (for algorithm design and debugging) and a test set. More subsets are extracted to meet the requirements of the baseline methods (see below). Table 4 shows the statistics of the datasets.

Baseline methods: We compare our approach with two baselines: KAZB (Kushman et al., 2014) and BasicSim.

KAZB is a learning-based statistical method which solves a problem by mapping it to one of the *equation templates* determined by the annotated equations in the training data. We run the ALLEQ version of their algorithm since it performs much better than the other two (i.e., 5EQ and 5EQ+ANS). Their codes support only linear equations and require that there are at least two problems for each equation template (otherwise an exception will be thrown). By choosing problems from the collection that meet these requirements, we build a sub-dataset called LinearT2. In the dataset of KAZB, each equation template corresponds to at least 6 problems. So we form another sub-dataset called LinearT6 by removing from the test set the problems for which the associated equation template appears less than 6 times.

BasicSim is a simple statistical method which works by computing the similarities between a testing problem and those in the training set, and then applying the equations of the most similar problem. This method has similar performance with KAZB on their dataset, but does not have the two limitations mentioned above. Therefore we adopt it as the second baseline.

For both baselines, experiments are conducted using 5-fold cross-validation with the dev set always included in the training data. In other words, we always use the dev set and 4/5 of the test set as training data for each fold.

Evaluation metrics: Evaluation is performed in the setting that a system can choose NOT to answer all problems in the test set. In other words, one has the flexibility of generating answers only when she knows how to solve it or she is confident about her answer. In this setting, the following three metrics are adopted in reporting evaluation results (assuming, in a test set of size n, a system generates answers for m problems, where k of them are correct):

Precision: k/m

Recall (or coverage): k/n

F1: $2PR/(P+R) = 2k/(m+n)$

Dataset		#problems	#sentences (average)	#words (average)
All	dev	374	1.79	20.3
	test	1,504	1.75	22.5
Linear	dev	247	1.78	19.6
	test	986	1.72	19.0
LinearT2	dev	172	1.85	18.8
	test	669	1.71	17.4
LinearT6	dev	71	1.96	16.8
	test	348	1.80	16.1

Table 4: Dataset statistics (Linear: problems with linear equations; T2: problems corresponding to template size ≥ 2)

4.2 Experimental results

The Overall evaluation results are summarized in Table 5, where "Dolphin" represents our approach. The results show that our approach significantly outperforms (with $p<<0.01$ according to two-tailed t-test) the two baselines on every test set, in terms of precision, recall, and F-measure. Our approach achieves a particularly high precision of 95%. That means once an answer is provided by our approach, it has a very high probability of being correct.

Please note that our grammar rules and parsing algorithm are NOT tuned for the evaluation data. Only the dev set is referred to in system building.

[5] Available from http://research.microsoft.com/en-us/projects/dolphin/

[6] http://www.algebra.com

[7] https://answers.yahoo.com/

[8] Math equations are used in the baseline approaches as part of training data.

Since the baselines generate results for *all* problems, the precision, recall, and F1 are all the same for each dataset.

Dataset	Method	Precision (%)	Recall (%)	F1 (%)
LinearT6	KAZB	49.1	49.1	49.1
	BasicSim	59.7	59.7	59.7
	Dolphin	**98.1**	**72.9**	**83.6**
LinearT2	KAZB	37.5	37.5	37.5
	BasicSim	46.3	46.3	46.3
	Dolphin	**97.3**	**68.0**	**80.0**
Linear	BasicSim	32.3	32.3	32.3
	Dolphin	**95.7**	**63.6**	**76.4**
Test set all	BasicSim	29.0	29.0	29.0
	Dolphin	**95.4**	**60.2**	**73.8**

Table 5: Evaluation results

The reason for such a high precision is that, by transforming NL text to DOL trees, the system "*understands*" the problem (or has structured and accurate information about quantity relations). Therefore it is more likely to generate correct results than statistical methods who simply "guess" according to features. By examining the problems in the dev set that we cannot generate answers, we find that most of them are due to empty parsing results.

On the other hand, statistical approaches have the advantage of generating answers without understanding the semantic meaning of problems (as long as there are similar problems in the training data). So they are able to handle (with probably low precision) problems that are complex in terms of language and logic.

Please pay attention that our experimental results reported here are on *number word problems*. General math word problems are much harder to our approach because the entity types, properties, relations, and actions contained in general word problems are much larger in quantity and more complex in quality. We are working on extending our approach to general math word problems. Now our DOL language and CFG grammar already have a good coverage on common entity types, but the coverage on properties, relations, and actions is quite limited. As a result, our parser fails to parse many sentences in general math word problems because they contain properties, relations or actions that are unknown to our system. We also observe that sometimes we are able to parse a problem successfully, but cannot derive math expressions in the reasoning stage. This is often because some relations or actions in the problem are not modeled appropriately. As future work, we plan to extend our DOL lexicon and grammar to improve the coverage of properties, relations, and actions. We also plan to study the mechanism of modeling relations and actions.

5 Conclusion

We proposed a semantic parsing and reasoning approach to automatically solve math number word problems. We have designed a new meaning representation language DOL to bridge NL text and math expressions. A CFG parser is implemented to parse NL text to DOL trees. A reasoning module is implemented to derive math expressions from DOL trees, by applying the semantic interpretation of DOL nodes. We achieve a high precision and a reasonable recall on our test set of over 1,500 problems. We hope to extend our techniques to handling general math word problems and to other domains (like physics and chemistry) in the future.

Acknowledgments

We would like to thank the annotators for their efforts in assigning math equations and answers to the problems in our dataset. Thanks to the anonymous reviewers for their helpful comments and suggestions.

Reference

J.W. Backus. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. Proceedings of the International Conference on Information Processing, 1959.

Y. Bakman. 2007. Robust understanding of word problems with extraneous information. http://arxiv.org/abs/math/0701393. Accessed Feb. 2nd, 2015.

C. Baker, M. Ellsworth, and K. Erk. 2007. SemEval-2007 Task 19: Frame semantic structure extraction. In Proceedings of SemEval.

L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider. 2013. Abstract meaning representation for sembanking. In Proc. of the Linguistic Annotation Workshop and Interoperability with Discourse.

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

J. Berant and P. Liang. 2014. Semantic Parsing via Paraphrasing. In ACL'2014.

D.G. Bobrow. 1964a. Natural language input for a computer problem solving system. Report MAC-TR-1, Project MAC, MIT, Cambridge, June

D.G. Bobrow. 1964b. Natural language input for a computer problem solving system. Ph.D. Thesis, Department of Mathematics, MIT, Cambridge

D.L. Briars, J.H. Larkin. 1984. An integrated model of skill in solving elementary word problems. Cognition and Instruction, 1984, 1 (3) 245-296.

Q. Cai and A. Yates. 2013. Large-scale semantic parsing via schema matching and lexicon extension. In Association for Computational Linguistics (ACL).

X. Carreras. and L. Marquez. 2004. Introduction to the CoNLL-2004 shared task: Semantic role labeling. In Proceedings of CoNLL.

E. Charniak. 1968. CARPS: a program which solves calculus word problems. Report MAC-TR-51, Project MAC, MIT, Cambridge, July

E. Charniak. 1969. Computer solution of calculus word problems. In Proceedings of international joint conference on artificial intelligence. Washington, DC, pp 303–316

N. Chomsky. 1956. Three models for the description of language. Information Theory, IRE Transactions on, 2(3), 113-124.

S. Clark, and J. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. Computational Linguistics, 33(4):493-552.

D. Das, D. Chen, A.F.T. Martins, N. Schneider and N.A. Smith. 2014. Frame-Semantic Parsing. Computational Linguistics 40:1, pages 9-56

D. Dellarosa. 1986. A computer simulation of children's arithmetic word problem solving. Behavior Research Methods, Instruments, & Computers, 18:147–154

V. Durme, T. Qian, and L. Schubert. 2008. Class-driven attribute extraction. In Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1, pp. 921-928. Association for Computational Linguistics, 2008.

J. Earley. 1970. An efficient context-free parsing algorithm. Communications of the ACM, 13(2), 94-102.

C.J. Fillmore, C.R. Johnson, and M.R. Petruck. 2003. Background to FrameNet. International Journal of Lexicography, 16(3).

C.R. Fletcher. 1985. Understanding and solving arithmetic word problems: a computer simulation. Behavior Research Methods, Instruments, & Computers, 17:565–571

D. Gildea, and D. Jurafsky. 2002. Automatic labeling of semantic roles. Computational Linguistics, 28(3).

M. Hearst. 1992. Automatic Acquisition of Hyponyms from Large Text Corpora. In Fourteenth International Conference on Computational Linguistics, Nantes, France.

M.J. Hosseini, H. Hajishirzi, O. Etzioni, and N. Kushman. 2014. Learning to Solve Arithmetic Word Problems with Verb Categorization. In EMNLP'2014.

D. Jurafsky, and J.H. Martin. 2000. Speech & language processing. Pearson Education India.

T. Kasami. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages (Technical report). AFCRL. 65-758.

P. Kingsbury, and M. Palmer. 2002. From TreeBank to PropBank. In Proceedings of LREC.

N. Kushman, Y. Artzi, L. Zettlemoyer, and R. Barzilay. 2014. Learning to automatically solve algebra word problems. In Proc. of the Annual Meeting of the Association for Computational Linguistics (ACL).

T. Kwiatkowski, E. Choi, Y. Artzi, and L. Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In Empirical Methods in Natural Language Processing (EMNLP).

I. Lev, B. MacCartney, C. Manning, and R. Levy. 2004. Solving logic puzzles: From robust processing to precise semantics. In Proceedings of the Workshop on Text Meaning and Interpretation. Association for Computational Linguistics.

C. Liguda, T. Pfeiffer. 2012. Modeling Math Word Problems with Augmented Semantic Networks. NLDB'2012, pp. 247-252.

Y. Ma, Y. Zhou, G. Cui, R. Yun, R. Huang. 2010. Frame-based calculus of solving arithmetic multistep addition and subtraction word problems. In International Workshop on Education Technology and Computer Science, vol. 2, pp. 476–479.

L. Marquez, X. Carreras, K.C. Litkowski, and S. Stevenson. 2008. Semantic role labeling: an introduction to the special issue. Computational Linguistics, 34(2).

A. Mukherjee and U. Garain. 2008. A review of methods for automatic understanding of natural language mathematical problems. Artificial Intelligence Review, 29(2).

M. Pasca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. 2006. Organizing and searching the world wide web of facts-step one: the one-million fact extraction challenge. In AAAI (Vol. 6, pp. 1400-1405).

K.K. Schuler. 2005. VerbNet: A broad-coverage, comprehensive verb lexicon. Dissertation. http://repository.upenn.edu/dissertations/AAI3179808

S. Shi, H. Zhang, X. Yuan, and J.-R. Wen. 2010. Corpus-based semantic class mining: distributional vs. pattern-based approaches. In Proceedings of the 23rd International Conference on Computational Linguistics, pages 993–1001. Association for Computational Linguistics.

M. Steedman. 2000. The Syntactic Process. The MIT Press.

Y. W. Wong and R. J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In Association for Computational Linguistics (ACL), pages 960–967.

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

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

L.S. Zettlemoyer and M. Collins. 2007. Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL).

F. Zhang, S. Shi, J. Liu, S. Sun, and C.-Y. Lin. 2011. Nonlinear evidence fusion and propagation for hyponymy relation mining. In ACL, volume 11, pages 1159–1168.