# The NL2KR system

Chitta Baral, Juraj Dzifcak, Kanchan Kumbhare, and Nguyen H. Vo

School of Computing, Informatics, and Decision Systems Engineering,
Arizona State University, Tempe, Arizona, USA
chitta@asu.edu, Juraj.Dzifcak@asu.edu, krkumbha@asu.edu,
Nguyen.H.Vo@asu.edu

**Abstract.** In this paper we will describe the NL2KR system that translates natural language sentences to a targeted knowledge representation formalism. The system starts with an initial lexicon and learns meaning of new words from a given set of examples of sentences and their translations. We will describe the first release of our system with several examples.

**Keywords:** Natural Language Understanding, Lambda Calculus, Knowledge Representation

## 1 Introduction and Motivation

Our approach to understanding natural language involves translating natural language text to formal statements in an appropriate knowledge representation language so that a reasoning engine can reason with the translated knowledge and give a response, be it an answer, a clarifying question or an action. To translate natural language text to a formal statement we propose to use the compositional method of Montague [1] where the translation (or meaning) of words are given as lambda calculus formulas and the meaning of phrases and sentences are obtained by composing the meaning of the constituent words. The challenge in doing this is in coming up with appropriate lambda calculus expressions for each word. The challenging aspects in this are: (a) the number of words may be huge, (b) the lambda calculus expression (or meaning) of some words are too complicated for humans to come up with it, and (c) the lambda calculus expressions for the words are target language specific; so it is not a one time affair like compiling traditional dictionaries. To address these challenges we use an inverse lambda algorithm [2] that computes the meaning of a word/phrase G when the meaning of the word/phrase H and the phrase GH (or HG) is known.

The NL2KR system uses an initial lexicon containing some words and their meanings and a set of training corpus containing sentences in natural language and their translations to learn new meanings of words. The system then uses the new learned lexicon to translate new sentences. In this paper, we would like to give an overview of the NL2KR system and examples of using it.

## 2 Overview

Shown below in Fig. 1 is the architecture of the NL2KR system. It has two sub-parts which depend on each other (1) NL2KR-L for learning and (2) NL2KR-T for translating.

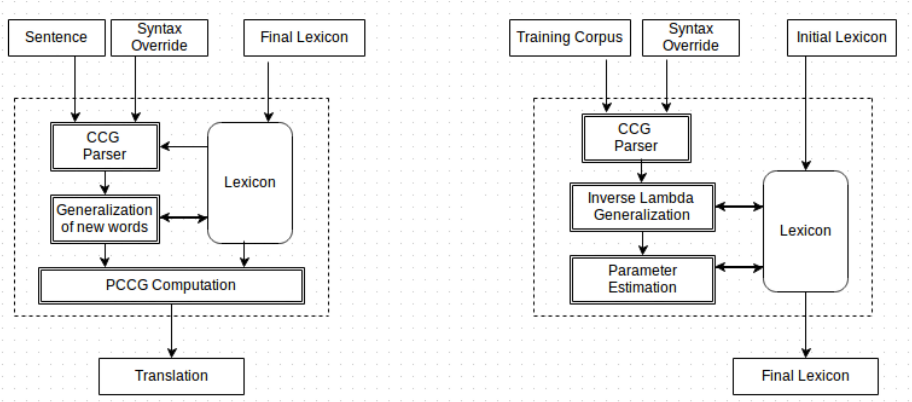


**Fig. 1.** Architecture of the NL2KR system: NL2KR-T on the left and NL2KR-L on the right

The NL2KR-L sub-part takes an initial lexicon consisting of some words and their meanings in terms of $\lambda$-calculus expressions & a set of training sentences and their target formal representations as input. It then uses a Combinatorial Categorical Grammar (CCG) parser to construct the parse trees. Next, the learning sub-part of the system uses Inverse-$\lambda$ and Generalization algorithms to learn meanings of newly encountered words, which are not present in the initial lexicon, and adds them to the lexicon. A parameter learning method is then used to estimate a weight for each lexicon entry (word, its syntactic category and meaning) such that the joint probability of the sentences in the training set getting translated to their given formal representation is maximized. The result of NL2KR-L is the final lexicon, which contains a larger set of words, their meanings and their weights.

Once the training component finishes its job, the translation sub-part (NL2KR-T) uses this updated lexicon and translates sentences using the CCG parser. Since words can have multiple meanings and their associated $\lambda$-calculus expressions, weights assigned to each lexical entry in the lexicon helps in deciding the more likely meaning of a word in the context of a sentence.

## 3 Using NL2KR

The latest version of NL2KR system can be downloaded from http://nl2kr.engineering.asu.edu. It can be run on Linux (64 bit) and OS-X.

### 3.1 Third Party Software Used by NL2KR

The current version of NL2KR uses the following libraries and tools developed by others:

- Scripting language Python (version $\geq$ 2.6)
- AspCcgTk version 0.3 [1]
- Stanford Log-linear Part-Of-Speech Tagger [2] (version 3.1.5)
- Oracle Java (version $\geq$ 1.6)
- ASP grounder and solver: gringo (version 3.x), clasp (version 2.x) and clingo (version 3.x) [3]

### 3.2 Installation guide

The NL2KR package contains a readme file and a zipped file which contains

- AspCcgTk
- Jar file and models of Stanford Log-linear Part-Of-Speech Tagger
- Jar file and configurations for NL2KR
- Gringo, clasp and clingo

After unzipping the package, the instruction in the readme file directs how to install NL2KR.

### 3.3 Files/Folders in the package

Following is a brief list of important files/folders in the package:

- **README**: Installation instruction and examples of how to use NL2KR.
- **NL2KR.jar**: NL2KR's classes packed in a jar file.
- **config.properties**: Default configuration of the NL2KR system.
- **install.sh**: Script to install NL2KR.
- **ccgParser.sh**: Script to get a CCG parse tree of a given sentence.
- **Generalization.sh**: Script that gives generalized meanings of a word.
- **Inverse.sh**: Script to compute the inverse using the inverse lambda algorithms.
- **Lambda.sh**: Script to do application operation, given a function and an argument in $\lambda$-calculus.
- **NL2KR-L.sh**: Script to run the NL2KR-L sub-part.
- **NL2KR-T.sh**: Script to run the NL2KR-T sub-part.
- **RunConfiguration**: Example inputs of the preceding scripts.
- **resources**: Folder containing AspCcgTk package, gringo, clasp and clingo.
- **examples**: Folder containing examples in various domains for NL2KR-L and NL2KR-T.

[1] http://www.kr.tuwien.ac.at/staff/former_staff/ps/aspccgtk
[2] http://nlp.stanford.edu/software/tagger.shtml
[3] http://potassco.sourceforge.net/

### 3.4 Executing the scripts

To execute any of the scripts one needs to go to the NL2KR's root directory and run

```
./<script name> <RunConfiguration file> [Optional params]
```

where *script name* is the name of one of the six scripts (e.g. ./Lambda.sh), RunConfiguration file contains corresponding parameters for the script, and optional parameters are for the Java Virtual Machine (JVM) to execute the module that corresponds to the script. For learning and testing large dataset with NL2KR-L or NL2KR-T, it is recommended to provide more memory for the JVM and enable garbage collection if needed (i.e. Use -Xmx and -XX:-UseGCOverheadLimit).

### 3.5 Lambda application

To use the lambda application script, the function and the argument of the lambda application need to be provided. For example, the following snippet in the RunConfiguration file specifies that we need to apply the argument $\#x.x@mia$ to the function $\#y.\#x.loves(x, y)$ (note: # is for $\lambda$).

```
function=#y.#x.loves(x,y)
argument=#x.x@mia
```

The output of the lambda application is $function@argument$. According to lambda application, the $y$ variable is replaced by $\#x.x@mia$ and the result is $\#x.loves(x, \#x0.x0@mia)$, where $x$ in the argument is renamed to $x0$. Running the lambda application script with the preceding configuration yields the output:

```
Function=#y.#x.loves(x,y)
Argument=#x.x@mia
Result= #x.loves(x,#x0.x0 @ mia)
```

However, in the following configuration, the argument cannot be applied to the function since there is no free variable in the function.

```
function = loves(x,y)
argument = #x.x @ mia
```

The output thus would be

```
Function = loves(x,y)
Argument = #x.x @ mia
Cannot apply the argument#x.x @ mia to the function loves(x,y)
```

### 3.6 Inverse application

Given two lambda expressions $g$ and $h$, the lambda application gives us $f = g@h$ or $f = h@g$. But sometime, we have only $f$ and $g$ and we need to find $h$. The inverse application allow us to calculate the lambda expression $h$ so that $f = g@h$ or $f = h@g$. More details about the inverse application can be found in [2]. For inverse application, we need to provide the parent (lambda expression $f$), the right child ($r$) or the left child ($l$). Given one child, the module will calculate the other child so that $f = l@r$.

In the first example below, since there does not exist a lambda expression $h$ (right child) so that $mia@h = \#x.loves(x, mia)$, the inverse algorithm returns right child expression = null. However, when $mia$ is the right child, inverse lambda returns $Leftchild = \#x1.\#x.loves(x, x1)$ because $\#x1.\#x.loves(x, x1)@mia = \#x.loves(x, mia)$. In case the CCG parse tree specifies that the meaning of "Mia" must be in the left child, using left child as $\#x.x@mia$ instead of $mia$ will do the trick and let us have the same right child: $\#x1.\#x.loves(x, x1)$.

*Example 1.* Input:

```
parent=#x.loves(x,mia)
left_child=mia
```

Output:

```
Parent = #x.loves(x,mia)
Left child =mia
Right child = null
```

*Example 2.* Input:

```
parent=#x.loves(x,mia)
right_child=mia
```

Output:

```
Parent = #x.loves(x,mia)
Right child = mia
Left child = #x1.#x.loves(x,x1)
```

*Example 3.* Input:

```
parent=#x.loves(x,mia)
left_child=#x.x @ mia
```

Output:

```
Parent = #x.loves(x,mia)
Left child =#x.x @ mia
Right child = #x1.#x.loves(x,x1)
```

### 3.7 Generalization

The inverse lambda module is not always adequate to learn new meanings of words when we lack meaning of words that will allow us to use the inverse lambda module. To address that we have developed a generalization module in NL2KR system, where the generalization technique described in [2] is implemented. For example, if we want to find the meaning of the word *plays* with the category $(S\backslash NP)/NP$ using generalization, we can use the lexical entry (*eats*, $(S\backslash NP)/NP$, $\lambda y.\lambda x.eats(x, y)$) in the lexicon, where the category of the word *eats* is same as that of the word *plays*. The generalization module will add a new lexical entry (*plays*, $(S\backslash NP)/NP$, $\lambda y.\lambda x.plays(x, y)$) to the lexicon. The input of the generalization module is the file path for lexicon (existing dictionary) and a new word, which we want to generalize. Following is an illustration of the use of generalization with the RunConfiguration file having the following:

```
lexicon=./examples/sample/dictionary.txt
word=Mia
ccg=N
```

where **dictionary.txt** contains

```
Vincent     N     vincent
Vincent     N     #x.vincent(x)
takes (S\NP)/NP   #w. #z. (w@ #x. takes(z,x) )
plane N     #x. plane(x)
boxer N     #x. boxer(x)
fights      S\NP  #x. fight(x)
```

In this case, the word **Mia** can be obtained by generalization from the words Vincent, plane and boxers, each of category N (meaning noun); and the output is

```
New lexical items learned through Generalization:
Mia     [N]   #x.mia(x)
Mia     [N]   mia
```

We can restrict generalization by modifying the config.properties file. For example, adding the following snippet to config.properties will skip the generalization process for NP and N categories, and generalization for the words: *on*, *of*, *by* and *in*.

```
GENERALIZATION_D_EXCLIST=[NP],[N]
GENERALIZATION_D_PREPOSITIONLIST=on,of,by,in
```

### 3.8 CCG parser

The input of the CCG parser module is the sentence we want to parse and an optional path of the file containing the words and their additional categories we want to use. The parser will parse the input sentence and output its CCG parse tree. Our CCG parser is based on *ASPccgTk* [3] with some modifications such as our use of the Stanford Part-Of-Speech tagger [4] instead of the C&C Part-Of-Speech tagger [5] to improve accuracy. Following is an example snippet of the RunConfiguration file.

```
sentence='Every boxer walks'
syntaxFile=./examples/sample/syntax.txt
```

where **syntax.txt** contains

```
takes (S\NP)/NP
Every (S/(S\NP))/NP
Some  (S/(S\NP))/NP
walks S\NP
fights      S\NP
loves (S\NP)/NP
```

The output of the CCG parser is a parse tree of the sentence "Every boxer walks" in ASP format as follows

```
nl2kr_token(t1, "Every", "(S/(S\NP))/NP", 1).
nl2kr_token(t2, "boxer", "NP", 2).
nl2kr_token(t3, "walks", "S\NP", 3).
nl2kr_token(t4, "Every boxer", "S/(S\NP)",1).
nl2kr_token(t5, "Every boxer walks", "S", 1).
nl2kr_child_left(t4, t1).
nl2kr_child_right(t4, t2).
nl2kr_child_left(t5, t4).
nl2kr_child_right(t5, t3).
nl2kr_valid_rootNode(t5).
```

The predicate *nl2kr_valid_rootNode* is used to specify the root node of the parse tree (corresponds to the whole sentence). *nl2kr_token* is used to specify nodes in the tree and *nl2kr_child_left*, *nl2kr_child_right* are for the left child and the right child of a node. The four atoms of *nl2kr_token* are respectively the node ID, the corresponding phrase, its CCG category and its starting position in the sentence.

### 3.9 NL2KR-L: the learning module of NL2KR

To run the learning module NL2KR-L we need to set the initial lexicon file path, the override file for syntax categories (optional), the training data file and the output dictionary file path (optional) in the RunConfiguration file of the NL2KR-L. Following is an example snippet of the RunConfiguration file.

```
Ldata=./examples/sample/train.txt
Ldictionary=./examples/sample/dictionary.txt
Lsyntax=./examples/sample/syntax.txt
Loutput=./examples/sample/dictionary_train.out
```

For example, the preceding snippet specifies that: the training data is in **./examples/train.txt**, the initial lexicon is in **./examples/sample/dictionary.txt**, and the override syntax categories are in **./examples/sample/syntax.txt**. The override syntax categories will be used in CCG parsing step as showed in the previous subsection. If it is not specified, the output dictionary is saved as dictionary_train.out in **./output** folder.

The training data file contains the training sentences and their formal representation such as:

```
Some boxer walks  EX. (boxer(X) ^ walk(X))
John takes a plane      EX. (plane(X) ^ takes(john, X))
John walks  walk(john)
```

In the above, $EX$ denotes $\exists X$.

The initial lexicon contains words and the their meanings that we already know:

```
John  N     john
takes (S\NP)/NP   #w. #z. (w@ #x. takes(z,x) )
plane N     #x. plane(x)
boxer N     #x. boxer(x)
fights      S\NP  #x. fight(x)
```

The NL2KR-L sub-part learns the new meanings of words in multiple iterations. It stops when it cannot learn any new word. Below we give the output of the script with example inputs.

We start with some snippets of the running output in the following. From line 5 to 9, NL2KR-L was checking if it has the meaning of *Some boxer* and *walks*. It then learned the meaning of *walks* by generalization.

From line 15 to 19, NL2KR-L was trying to learn the meaning of *Some boxer* given the meaning of *walks* and the meaning of the whole sentence *Some boxer walks* from the training data. Using inverse lambda, it figured out that the meaning of *Some boxer* is $\#x1.EX.boxer(X) \wedge x1@X$.

NL2KR-L did not go further to the meaning of "some" because the meaning "boxer" of *boxer* was not helpful.

However, using the second parse tree where the meaning $\#x.boxer(X)$ of *boxer* is used, NL2KR-L can get the meaning of *some* (line 42-49) : $\#x2.\#x1.EX.x2@X \wedge x1@X$.

```
******Learning lexicon ...
...
Processing sentence number 1
Processing Parse Tree 1
Word : Some boxer walks sem::null
Both children do not have current lambda expression: Some boxer,walks
Generalizing for leafs with no expected lambda: walks
New lexical item Learned by Expansion: walks////[S\NP]////#x.walk(x)
...
Processing sentence number 1

Processing Parse Tree 1

Word : Some boxer walks sem::null
Applying inverse : EX.boxer(X) ^ walk(X) #x.walk(x)
INVERSE_L Tried:
Some boxer walks(H) = EX.boxer(X) ^ walk(X)
walks(G) = #x.walk(x)
Some boxer(F) = #x1.EX.boxer(X) ^ x1 @ X
Word : walks sem::#x.walk(x)
Word : Some boxer sem::null
Applying inverse : #x1.EX.boxer(X) ^ x1 @ X boxer
INVERSE_L Tried:
Some boxer(H) = #x1.EX.boxer(X) ^ x1 @ X
boxer(G) = boxer
Some(F) = null
Generalizing for leafs with no expected lambda: Some
Generalizing for leafs with no expected lambda: boxer
Word : boxer sem::boxer
Word : Some sem::null

Processing Parse Tree 2

Word : Some boxer walks sem::null
Applying inverse : EX.boxer(X) ^ walk(X) #x.walk(x)
INVERSE_L Tried:
Some boxer walks(H) = EX.boxer(X) ^ walk(X)
walks(G) = #x.walk(x)
Some boxer(F) = #x1.EX.boxer(X) ^ x1 @ X
Word : walks sem::#x.walk(x)
Word : Some boxer sem::null
Applying inverse : #x1.EX.boxer(X) ^ x1 @ X #x.boxer(x)
INVERSE_L Tried:
Some boxer(H) = #x1.EX.boxer(X) ^ x1 @ X
boxer(G) = #x.boxer(x)
Some(F) = #x2.#x1.EX.x2 @ X ^ x1 @ X
Word : boxer sem::#x.boxer(x)
Word : Some sem::null
New lexicon Learned: Some////[(S/(S\NP))/NP]////#x2.#x1.EX.x2 @ X ^ x1 @ X
```

At the end of the learning phase, parameter estimation is run to assign the weights for each meaning of words. NL2KR-L then uses those meanings to check if they are enough to translate the training sentences correctly.

```
******Evaluation on training set ...

Processing training sentence: Some boxer walks
Predicted Result: EX.boxer(X) ^ walk(X)
Correct Prediction

Processing training sentence: John takes a plane
Predicted Result: EX.plane(X) ^ takes(john,X)
Correct Prediction
```

```
...
```

Following is the output lexicon learned with the example inputs mentioned earlier. Each row contains a word, its CCG category, its meaning and the associated weight. Compared to the initial dictionary, we can see that NL2KR-L learned 14 more word meanings. Note that some words such as *likes* and *eats* are in the "syntax.txt".

```
Some  [(S/(S\NP))/NP]   #x2.#x1.EX.x2 @ X ^ x1 @ X    0.0074364278
fight [S\NP]      #x.fight(x) 0.01
boxer [N]   #x.boxer(x) 0.060000002
boxer [N]   boxer -0.041248113
a     [NP/N]      #x4.#x2.EX.x4 @ X ^ x2 @ X    0.009887816
John  [NP]  john  0.0073314905
John  [N]   john  0.01
eats  [(S\NP)/NP] #w.#z.w @ #x.eats(z,x)  0.01
fights      [S\NP]      #x.fights(x)      0.01
fights      [S\NP]      #x.fight(x) 0.01
takes [(S\NP)/NP] #w.#z.w @ #x.takes(z,x) 0.01
walks [S\NP]      #x.walks(x) -0.08722576
walks [S\NP]      #x.walk(x)  0.10615976
plane [N]   #x.plane(x) 0.059950046
plane [N]   plane -0.03995005
likes [(S\NP)/NP] #w.#z.w @ #x.likes(z,x) 0.01
flies [S\NP]      #x.fly(x)   0.01
flies [S\NP]      #x.flies(x) 0.01
loves [(S\NP)/NP] #w.#z.w @ #x.loves(z,x) 0.01
```

### 3.10 NL2KR-L in the Geoquery domain

In this subsection, we present an example of using NL2KR-L for the GEOQUERY[4] domain. GEOQUERY uses a Prolog based language to query a database with geographical information about the U.S. The input of NL2KR-L is specified in the RunConfiguration file as:

```
Ldata=./examples/geoquery/train.txt
Ldictionary=./examples/geoquery/dictionary.txt
Lsyntax=./examples/geoquery/syntax.txt
Loutput=
```

where ./examples/geoquery/train.txt contains

```
How large is texas      answer(X) ^ size(B,X) ^ const(B,sid,texas)
How high is mountmckinley     answer(X) ^ elevation(B, X) ^ const(B,
    pid,mountmckinley)
How big is massachusetts      answer(X)^ size(B, X) ^ const(B,
    sid,massachusetts)
How long is riogrande   answer(X)^ len(B, X) ^ const(B, rid,riogrande)
How tall is mountmckinley     answer(X)^ elevation(B, X)^ const(B,
    pid,mountmckinley)
```

./examples/geoquery/dictionary.txt contains

```
How   S/S   #x.answer(X) ^ x@X
texas NP    #x.const(x,sid,texas)
mountmckinley     NP    #x.const(x,pid,mountmckinley)
massachusetts     NP    #x.const(x,sid,massachusetts)
riogrande   NP    #x.const(x,rid,riogrande)
is    (S\NP)/NP   #y. #x.x @ y
```

and ./examples/geoquery/syntax.txt contains

[4] http://www.cs.utexas.edu/users/ml/geo.html

```
How   S/S
texas NP
mountmckinley     NP
massachusetts     NP
riogrande   NP
is    (S\NP)/NP
rivers      NP
large NP
is    (S\NP)/NP
high  NP
big   NP
long  NP
the   NP/NP
How   S/(S\NP)
long  S\NP
tall  NP
colorado    NP
arizona     NP
```

After the learning module is executed, 15 more word meanings were learned by NL2KR-L and the result is:

```
is    [(S\NP)/NP] #y.#x.x @ y -0.0015125279
texas [NP]  #x.const(x,sid,texas)   0.07666666
texas [NP]  #x.const(x,rid,texas)   -0.023256822
texas [NP]  #x.const(x,pid,texas)   -0.023256822
riogrande   [NP]  #x.const(x,pid,riogrande)     -0.02315781
riogrande   [NP]  #x.const(x,rid,riogrande)     0.07646726
riogrande   [NP]  #x.const(x,sid,riogrande)     -0.02315781
mountmckinley     [NP]  #x.const(x,sid,mountmckinley) -0.055226557
mountmckinley     [NP]  #x.const(x,pid,mountmckinley) 0.14075616
mountmckinley     [NP]  #x.const(x,rid,mountmckinley) -0.055226557
massachusetts     [NP]  #x.const(x,rid,massachusetts) -0.023190754
massachusetts     [NP]  #x.const(x,sid,massachusetts) 0.0765336
massachusetts     [NP]  #x.const(x,pid,massachusetts) -0.023190754
long  [NP]  #x3.#x1.len(B,x1) ^ x3 @ B    0.010111484
How   [S/S] #x.answer(X) ^ x @ X    0.009999999
high  [NP]  #x3.#x1.elevation(B,x1) ^ x3 @ B    0.010112147
big   [NP]  #x3.#x1.size(B,x1) ^ x3 @ B   0.010111821
tall  [NP]  #x.const(x,rid,tall)    -0.014923776
tall  [NP]  #x.const(x,pid,tall)    -0.014923776
tall  [NP]  #x3.#x1.elevation(B,x1) ^ x3 @ B    0.084677815
tall  [NP]  #x.const(x,sid,tall)    -0.014923776
large [NP]  #x3.#x1.size(B,x1) ^ x3 @ B   0.010112498
```

### 3.11 NL2KR-T: the translation sub-part of NL2KR

Similar to NL2KR-L, in the RunConfiguration file of NL2KR-T, we need to set the lexicon file path, the override file for syntax categories(optional), and the testing data file as given below:

```
Tdata=./examples/sample/test.txt
Tdictionary=./output/dictionary_train.out
Tsyntax=./examples/sample/syntax.txt
```

For example, the preceding snippet specifies that: the testing data is in **./examples/sample/test.txt**, the lexicon is in **./output/dictionary_train.out**, and the override syntax categories are in **./examples/sample/syntax.txt**. The lexicon should be the lexicon learned by NL2KR-L.

The content of **./examples/sample/test.txt** is

```
Mia sleeps  sleep(mia)
John catches a bus      EX. (bus(X) ^ catches(john, X))
```

Running the NL2KR-T script with the inputs specified above, we have the following output where *John catches a bus* is translated to *EX. (bus(X) ∧ catches(john, X))* as expected.

```
******Parsing Sentences ...
...
Parsing test sentence: John catches a bus
Expected Representation: EX. (bus(X) ^ catches(john, X))
Generalizing bus = [bus : [N] : #x.bus(x), bus : [N] : bus]
Generalizing catches = [catches : [(S\NP)/NP] : #w.#z.w @ #x.catches(z,x)]
Predicted Result: EX.bus(X) ^ catches(john,X)
Correct Prediction
...
```

Note that the expected translation in "test.txt" is optional. Without it, the evaluation is not correct but NL2KR-T still gives its results.

## 4 Conclusion and Future Work

In this work, we presented the NL2KR system, which is used for translating natural language to a formal representation. The input of the NL2KR system are training sentences and their formal representation; and an initial lexicon of some known meanings of words. NL2KR system will try to learn the meaning of others words from the training data. We presented six scripts to execute several modules of NL2KR and show how to use them through examples.

In the future, we plan to make NL2KR more scalable and add more features to the NL2KR system such as (1) automatically constructing the initial lexicon and (2) using more knowledge such as word sense to select the correct meaning of words.

## References

1. Montague, R.: English as a Formal Language. In Thomason, R.H., ed.: Formal Philosophy: Selected Papers of Richard Montague. Yale University Press, New Haven, London (1974) 188–222
2. Baral, C., Dzifcak, J., Gonzalez, M.A., Zhou, J.: Using Inverse lambda and Generalization to Translate English to Formal Languages. CoRR **abs/1108.3843** (2011)
3. Lierler, Y., Schüller, P.: Parsing Combinatory Categorial Grammar via Planning in Answer Set Programming. In Erdem, E., Lee, J., Lierler, Y., Pearce, D., eds.: Correct Reasoning. Volume 7265 of Lecture Notes in Computer Science., Springer (2012) 436–453
4. Toutanova, K., Klein, D., Manning, C.D., Singer, Y.: Feature-rich part-of-speech tagging with a cyclic dependency network. In: NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, Morristown, NJ, USA, Association for Computational Linguistics (2003) 173–180
5. Clark, S., Curran, J.R.: Parsing the WSJ using CCG and log-linear models. In: ACL '04: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics, Morristown, NJ, USA, Association for Computational Linguistics (2004) 103