Implementing Natural Language to Alloy Transformations

Kevin Driver, Russell Glasser, Louis Helm, Oswin Housty

University of Texas, Austin, TX, USA
May 05, 2006

Abstract

This article describes how to implement an extensible natural language processor using JavaCC and JTB to convert Knight and Knave logic puzzles written in English into Alloy code. We also provide working code to automatically solve these Alloy-coded problems once converted.

Introduction to Knight and Knave puzzles

Logician Raymond Smullyan is the author of many books that contain logic puzzles, including *What is the name of this book?*, *The Lady or the Tiger?* and *Forever Undecided*. Smullyan's most famous puzzles are undoubtedly his stories about the far off "Island of Knights and Knaves."

On this island there are only two types of people: Knights, who always tell the truth, and Knaves, who always lie. In a typical puzzle, the reader is asked to imagine that he meets one or more natives of this island, who make a series of factual statements. The reader is then asked to deduce, based only on the statements presented, which people are Knights and which are Knaves.

The problems range from the simple (one or two islanders who make short sentences) to the complex (involving any number of islanders who make long statements with many clauses and conditions). Here is an example of a basic puzzle found early in *What is the name of this book?*:

> Two people, A and B, were standing together in a garden. A stranger passed by and asked "Are you knights or knaves?" A said: "Either I am a knave or B is a knight." What are A and B?

The solution given later in the book (abridged here) is:

> Suppose A is a knave. Then the statement "Either I am a knave or B is a knight" must be false. This means that it is neither true that

A is a knave nor that B is a knight. So if A were a knave, then it would follow that he is not a knave – which would be a contradiction. Therefore A must be a knight.

Since A is a knight, his statement is true. Therefore at least one of the possibilities holds: (1) A is a knave; (2) B is a knight. Since possibility (1) is false (since A is a knight) then possibility (2) must be the correct one, i.e., B is a knight. Hence A and B are both knights.

Such simple puzzles can easily be solved in the heads of most undergraduate computer science students, but there are far more complicated puzzles that are not so simply explained. On the University of Hong Kong's philosophy department website [http://www.hku.hk/cgi-bin/philodep/knight/puzzle], we find a repository of computer generated Knight/Knave puzzles.

One of the more complicated puzzles is as follows:

A very special island is inhabited only by knights and knaves. Knights always tell the truth, and knaves always lie. You meet three inhabitants: Abe, Zoey and Zippy. Abe says, "At least one of the following is true: that Zippy is a knave or that I am a knight." Zoey says, "Abe could claim that I am a knave.' Zippy claims, `Neither Abe nor Zoey are knights."

Later puzzles at the same site involve up to seven inhabitants. In his other books, Smullyan introduces even more complicated concepts. For example, in some puzzles set in Transylvania, all inhabitants are either humans (who tell the truth) or vampires (who always lie); but additionally, the inhabitants are either sane (and hence believe only true statements) or insane (and believe false statements). Hence, an insane vampire might falsely believe that 2+2=5, but he would "lie" and tell you that 2+2=4, thereby making a correct statement! And in even later chapters, instead of answering questions with "yes" and "no," they use the made-up words "bal" and "da," which could mean either one.

In principle, there is nearly unlimited potential to multiply the complexity of such problems. But even restricting ourselves to the basic Knight/Knave category of puzzle, as the number of inhabitants and the number of clauses in their statements increase, the problems can quickly

become so complicated as to overwhelm the reasoning capabilities of most readers.

An Alloy Framework

It is clear that this sort of logic puzzle is an ideal candidate for modeling in Alloy. Alloy is designed to model abstract logical constructs, and determine the consistency and specific solutions of a set of conditions.

For this project, we decided to create a system that goes through the following steps:

1. A Knight/Knave puzzle is written, in a limited subset of plain English, and stored in a file.
2. A parser reads the file and translates it into an Alloy program.
3. The program is then loaded into Alloy, and the solutions are found, if any exist.

The first task was to develop a general framework for modeling the world of Knights and Knaves. We created the following basic types:

- "Type" is an abstract superset of "Knights" and "Knaves."
```
sig Type { }
one sig Knight, Knave extends Type {}
```
- "TruthValue" is an abstract set determining the truth or falsehood of a statement. Specific TruthValues are defined to be either "True" or "False."
```
abstract sig TruthValue { }
one sig True, False extends TruthValue { }
```
- "Islanders" are people who have a type of either Knight or Knave. They are able to make a set of "claims", or statements, which are used determine which type they are.
```
abstract sig Islanders {
  type: one Type,
  claims: set Statement
}
```
- "Statements" are claims made by Islanders, which have a truth value (either true or false).
```
abstract sig Statement {
  value: TruthValue
}
```
- The rest of the basic Alloy framework involves specific types of statements such as simple declarative claims (i.e., "Carl is a knight"), negations (i.e., "John is not a knight"), conjunctions (i.e.,

"Sue is a knight and Frank is a knave"), and disjunctions (i.e., "Tony is a knave or Jen is a knight").

- We specify the condition that an islander can make a statement that is "True" if and only if he or she is a Knight.

```
fact {
  all i: Islanders | all s: i.claims |
    i.type = Knight <=> s.value = True
}
```

The next task was to generate Alloy code for a few sample problems. For instance, consider the example puzzle above, where there are two people, and only one of them makes a statement: "Either I am a knave or B is a knight."

Our grammar does not allow for the self referential statement ("I am") so we first reword the sentence as "Either A is a knave or B is a knight." Then we define the following objects:

```
one sig A, B extends Islanders {}
```

And we also define three statements: S1, "A is a knave"; S2, "B is a knight"; and S3, "S1 or S2". The resulting statements looks like this:

```
one sig S1 extends Isa{}
{
  A in target
  isa = knave
}

one sig S2 extends Isa{}
{
  B in target
  isa = knight
}

one sig S3 extends Or{}
{
  clause = S1+S2
}
```

Finally, we express the fact that A said the third statement:

```
fact { says ( A, S3 ) }
```

With all this work completed, we are able to plug the resulting code into Alloy and see the following result:

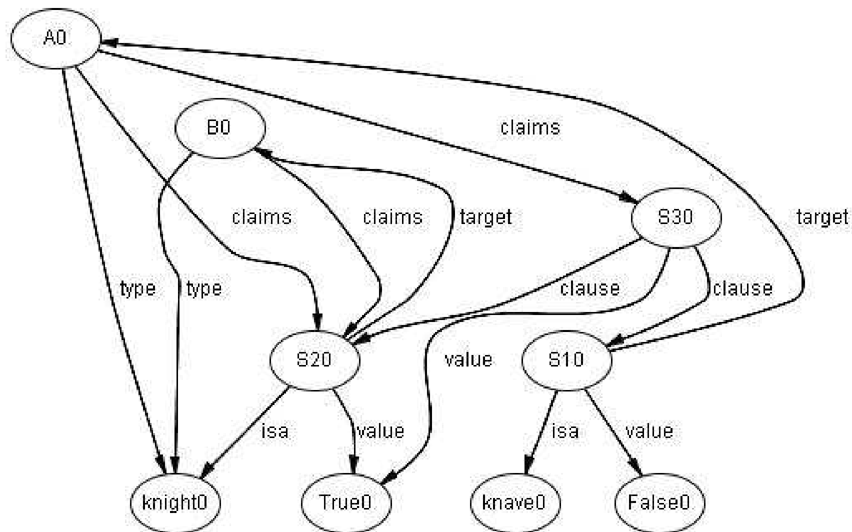

Figure 1: Sample output from Alloy

As you can see from the picture, the individual islanders “A” and “B” are represented in the upper left corner; the islander types “Knight” and “Knave” are represented on the bottom of the graph, as are the possible truth values “True” and “False”. Statements S1 and S2 are atomic statements, expressing that “somebody isa something” (either Knight or Knave). S1 points upward to “A”, and downward to “Knave”, indicating the claim that “A is a Knave.” It also points to the value of “False”, meaning the program has determined that S1 was not a true statement.

S3 is a disjunction of S1 and S2, so it points to both sub-statements, with the relationship of “clause.” Finally, “A” points to “S3” with the relationship “claims,” indicating that statement S3 is in the set of statements that A has uttered. S3 has a truth value of “True.” Since A has spoken the truth, A is a Knight, which is reflected in the fact that A’s “type” arrow points to “Knight.” B is also a Knight. Thus, we have discovered a complete solution to the puzzle.

Knight/Knave Grammar Elements

In processing any subset of natural language, a grammar is needed. A grammar defines a set of possible language productions which completely maps the set of meaningful inputs adhering to that grammar. Knight/Knave problems themselves have a relatively simple grammar. However, we found during the course of our work, that it is possible to add many additional extensions to the base grammar in order to enhance the problem scope we could represent.

Grammar design is outside the scope of this paper. However, two key ideas behind grammar design are relevant to our work. The first of these is ambiguity. Most modern-day grammar tools can warn about this type of problem, but it is critical to ensure that multiple inputs cannot follow more than one grammar production. Secondly, intelligent design was necessary in order to construct a grammar that could be easily and intuitively expanded and enhanced. In designing our grammar, we were careful to keep these two concepts in mind.

Preliminarily, we wanted to allow for the most basic Knight/Knave problems. It is desirable to be able to make statements about particular islanders simply describing whether these islanders were Knights or Knaves. This grammar was rudimentary, and easily implemented in the form `Identifier AssignmentExpression Islander`. `AssignmentExpression` is expanded in the productions to support "is a" or "is not a" clauses.

In most Knight/Knave problems, a given islander could state a claim about one or more islanders in a simple or moderately complex way. Modifications to the basic grammar outlined above are necessary. The next iteration of our grammar starts out in the form: `Identifier SaysLiteral Statement`.

This form can then be expanded via the `Statement` in order to allow for three main types of productions: `SimpleStatementAndSimpleStatement`, `SimpleStatementOrSimpleStatement`, and `SimpleStatement`.

`SimpleStatement` fully encompasses the initial grammar outlined above, while the "And" and "Or" clauses join two `SimpleStatement` clauses in a conjunctive or disjunctive manner. This grammar takes advantage of lookaheads. These will be discussed in more detail below.

Conceptually, this iteration of the grammar lays the foundations for parsing robust enough to handle an example such as this:

```
Bob says Alice is a Knight and Carol is a Knave.
Alice says Bob is a Knave.
Carol says either Alice is a Knight or Carol is
a Knight.
```

Thus, with these and similar examples in mind, a grammar is constructed. The final grammar can be viewed in Appendix C.

With a robust grammar, how can natural language be transformed into the Alloy modeling language?

A tool exists, which was developed at Purdue University and is now maintained at UCLA, called JTB or Java Tree Builder. JTB can parse a given grammar and generate a syntax tree and set of interfaces for visitor classes for the grammar. It also generates an adjusted grammar file output, which is then parsed by another tool called JavaCC. JavaCC then generates a set of scanner and parser classes based on the JTB grammar. JavaCC can be thought of as combining the functionality of lex and yacc into a single tool for the Java platform. Both tools are used extensively in Appel's Modern Compiler Implementation in Java.

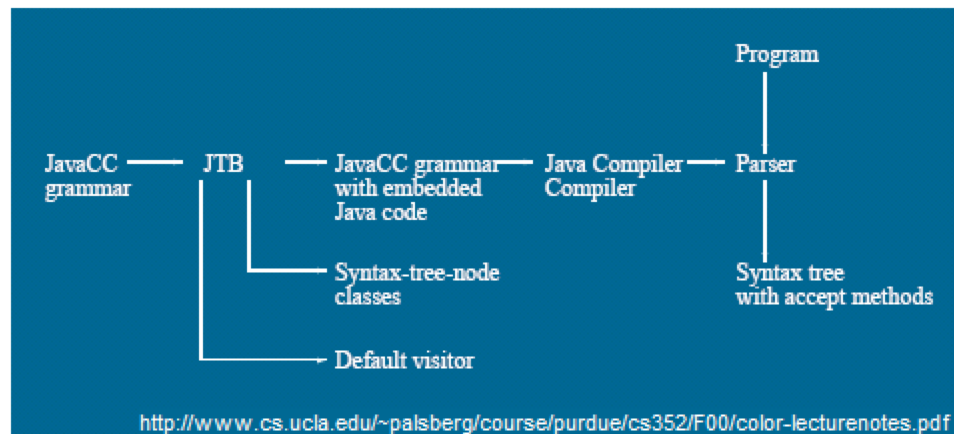

Figure 2: How JavaCC and JTB combine to create a new compiler

The benefits of using this approach with JTB and JavaCC are twofold. Once a grammar has been defined, the tokenizing and parsing is taken care of by the generated classes. The syntax tree classes recognize their corresponding grammar elements, freeing the developer from the mundane details of parsing. Constructions like lookaheads further simplify this process by aiding JavaCC in distinguishing between productions in the grammar. JavaCC allows for LL(k) with lookahead parsing while utilizing the syntax tree generated by JTB in order to "visit" a predefined set of possible nodes which make up a valid given input.

This "visitor" model allows for intuitive representation of the grammar in the context of the familiar tree data structure. Nodes can be parsed and dealt with at the leaf level and then passed up the tree to non-leaf nodes in order for further processing in context; thus, semantic meaning of the leaf nodes can be established as visitors pass productions further up the tree. Another benefit of the "visitor" tree model is that it allows for transformation between intermediate representations. Transforming from a high-level language such as Java to Assembly requires several intermediate representations. Fortunately, with intelligent design of some foundational Alloy code, transformation from Knight/Knave problems in natural language to Alloy code solely requires one transformation.

At its heart Alloy is about sets and relations. Knight/Knave problems are fundamentally simplistic enough so that a set of islanders and statements/claims about them can be intuitively mapped directly to Alloy code. This is why this particular problem domain was chosen for research in NLP/Alloy interfaces.

Common Alloy code was developed which can be shared between any such Knight/Knave problem. There are direct mappings from the grammar productions to this core Alloy code, which allows for the 1:1 intermediate representation from the NLP processor. For example, `AssignmentExpression` maps directly to an "Isa", or "Isa" in conjunction with a "Not." In this way code can be shared and reused between models and NLP to Alloy generation requires no intermediate representation transformations.

Visitors Class Implementation

A default visitor class is produced for each grammar when the JTB is run on the ".jjt" grammar file. A depth-first traversal is used to visit each leaf and non-leaf node of the abstract syntax tree. To parse the grammar, we customized the default visitor classes to translate individual parts of the grammar into Alloy. A new customized visitor method was defined for each actor in the production. Each actor calls child elements in the syntax tree using "accept" methods until leaf nodes are reached. The "accept" methods for non leaf nodes invoke the visitor methods for the calling actor. The "accept" methods in leaf nodes carry out actions to retrieve the leaf information for the node. Once the leaf node is reached, parents are recursively returned to until the root node is reached.

Consider the following example:

```
Sue says Zippy is a knave.
```

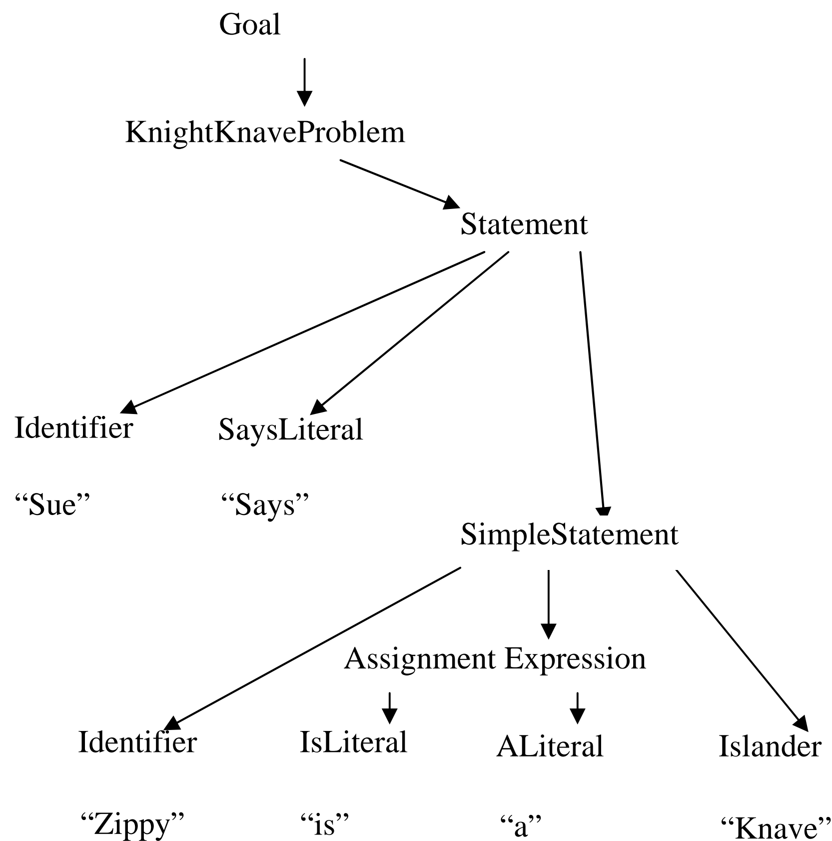

Figure 3: Example Abstract Syntax Tree

The program starts by running the main method and passing the input line above from a file. After the input is obtained, a "root" visitor is instantiated. Main runs the "accept" method on the root object and passes a new instance of a "Knight Knave visitor parser" as the new visitor method. The “accept” method of "Knight Knave visitor parser" calls the “visit” method of the "Goal" object to initiate the tree traversal.

The "Goal" begins the tree traversal by calling the "accept" method of its child node, "KnightKnaveProblem".

The "KnightKnaveProblem" accept method calls the visit method of its "accept" method of its' child, "Statement".

The "Statement" visit method calls the "accept" methods of its' children, "Identifier", "SaysLiteral" and "Simple Statement".

"Identifier" is a leaf node and calls the "accept" for the nodeToken, the nodeToken visit method obtains the string token and returns control to the "Identifier" visit method. The "Identifier" visit method stores the data for later use during the generation of the Alloy code. Control is returned

to the "Statement" visit method. The "Statement" visit method stores the return parameter from "Identifier" and calls the "SaysLiteral " accept method.

The "SaysLiteral" is also a leaf node, so its visit method calls its "accept" method for the nodeToken. The nodeToken visit method obtains the string token and returns control to the "SaysLiteral" visit method. The "SaysLiteral" returns control to the "Statement" node which calls the "SimpleStatement" visit method.

The "SimpleStatement" method calls the accept method for is children "Identifier", “AssignmentExpressoin" and “Islander.”

"Identifier" is a leaf node and calls the "accept" for the nodeToken, the nodeToken visit method obtains the string token and returns control to the "Identifier" visit method. The "Identifier" visit method stores the data for later use during the generation of the Alloy code. Control is returned to the "SimpleStatement" visit method. The "SimpleStatement" visit method stores the return parameter from "Identifier" and calls the "AssignmentExpression" accept method.

The "AssignmentExpression" method calls the accept methods for its children "IsLiteral" and "ALiteral".

The "IsLiteral" is a leaf node, so its visit method calls the "accept" method for nodeToken. The nodeToken visit method obtains the string token and returns control, obtaining the string token for "is". Control is returned to "AssignmentExpression".

The "ALiteral" is a leaf node, so its visit method calls the "accept" method for nodeToken. The nodeToken visit method obtains the string token and returns control, obtaining the string token for "a". Control is returned to "AssignmentExpression".

AssignmentExpression combines the "IsLiteral" and the "ALiteral" into a "isa" "AssignmentExpression". Control is returned to "SimpleStatement," where the information is processed into Alloy code. "SimpleStatement" then calls its accept, where the "Islander" accept method is run. The islander is a leaf node, so its visit method returns the islander type as a "knave".

After the final leaf node is reached, parameters are passed up the tree until control is returned to the root. The root passes control back to the main program where a call is made to a routine to create the “.als” file based on static code (Alloy definitions that do not change from problem to problem) and dynamic data obtained during the parsing of each Knight/Knave problem.

Visitors make object-oriented systems more flexible by allowing the manipulation of composite objects and the separation of unrelated operations. This flexibility is obtained without changing existing classes of the objects. Instead of using dedicated methods to complete an

operation, the visitor uses an "accept" method in each class and code within a "visitor" class to carry out a specific action.

To create a visitor, code must be inserted directly into an object structure. Next an accept method must be included in each object class. Finally, a visitor class must be defined with a visit method for each actor in the production.

Current Project Extensibility

One benefit of using a natural language parser is that the program has almost unlimited extension capabilities. Due to time considerations, we worked with a very limited subset of English, but there are many ways that this project could be improved in the future with more English language concepts. Some additional types of statements that we considered include:

- Expanded support of pronouns, i.e., accurate support of the statement "I am a knight."
- Hypothetical claims: "John could say that I am a knight"; "Only a knight would say that John is a knave."
- More complex negation: "It is not the case that Zoey is a knight and Zippy is a knave."
- If-then constructs; if-and-only-if constructs.
- Statements which are presumed to be always true or always false: "2+2=4"; "2+2=5"; "The sky is blue"; "The sky is yellow"
- Real world statements whose truth-value is unknown, but whose value we wish to solve for: "Either I am a knave or the treasure is behind door number one."

Further Applications

In developing this NLP to Alloy engine, it is obvious that we would be mindful of further applications of such work. We consider further applications in two realms. First, how can our current engine be extended in order to be relevant to other problem domains? One such application might be in boolean formulae. For instance, modifying the grammar only slightly, we could obtain statements of the form:

```
Formula1 says either x1 is true or x2 is true.
Formula2 says x1 is false and x3 is true.
```

These statements could be used to correct boolean errors and fill in truth assignments. Another slight alteration to this grammar could be used in modeling network paths and representing them in Alloy. It is a common algorithm in networking to decide which routing ports must be open and closed in order to construct a minimum spanning tree.

```
Router1 says r1r2 is open or r1r3 is open.
Router2 says r1r2 is closed and r2r3 is open.
```

These types of statements demonstrate states within the process of constructing such a minimum spanning tree and can be used in order to model state as a network proceeds through this or similar algorithms.

Clearly these are straightforward applications of the existing grammar with slight modification. Other related applications may exist as well, but these are the most obvious. Speaking more abstractly, but extrapolating from the progress we have made here, it may be possible at some point in the future to describe specifications according to a much richer grammar and to generate reasonable and robust Alloy models from natural language. Imagine writing specifications in natural language and being able to generate Alloy models directly from these specifications. We are confident that given a proper grammar and the proper number of intermediate representations between natural language and Alloy, it is possible to describe a wide variety of problems in natural language and to convert them into an analogous Alloy representation. Consider broader applications. If a particular grammar and set of productions could be applied to a set of eyewitness testimonies, it could be possible to model courtroom proceedings and check such proceedings for consistency. Perhaps a grammar could be designed to construct a data structure based on a natural language description. Imagine Alloy code which defines a tree, then describing in natural language that the definition of this tree should be altered to describe a binary tree or a red-black tree. Many of the Alloy representations of data structures read pretty closely to natural language in their current form, so this is not a drastic logical leap.

There are important ramifications to work in this space. With a powerful modeling language like Alloy and the toolset that surrounds it coupled with the robustness of natural language, it is possible that computers could one day be constructing models, composing software, or making decisions based on natural language input. Natural Language Processing and Alloy stand to take the computing world by storm with the proper amount of research and effort.

Related Work

A great body of work exists which deals with Natural Language Processing and modeling languages like Alloy, but chiefly separately. A natural language processing paper out of Stanford outlines many of the challenges faced in NLP: varying semantic concerns, plurality, scope, implied information [4]. These challenges face the whole of natural language, and they are real concerns. As for Alloy, MIT is continuously developing the language and its features. Many core examples and useful logic proofs have been implemented in this language. Valuable information exists in this space as well.

Perhaps where each is lacking is in the synergy. NLP on the whole is a difficult problem, but what about imposing reasonable constraints on the grammar? A human can be taught to type instructions in a certain subset of natural language without having to learn the ins and outs of software development. From the other direction, a large body of work and examples from the modeling side lends itself to easy extension of these models by simple alterations utilizing natural language. Combining the separate but related work seems to be an unharvested area, but one ripe with potential.

The key seems to be coercing industry and academia to cooperate. Microsoft Research, for example, has a dedicated NLP group, and is obviously more commercially interested in applications of NLP. MIT's work on Alloy is clearly very academic in nature – this is demonstrated clearly even in the default examples which are distributed. Combining these two entities, however, stands to create an invaluable amalgam of the commercial/industrial interests of large corporations with the research-oriented focus of the university. There is interesting theory in this space for the academics as well as potential for financial gain for the businessmen. The differing factions of related work need only combine their efforts.

Works Cited and References

1. Smullyan, Raymond. What is the Name of This Book? Touchstone Publishing, 1986.
2. Smullyan, Raymond. Forever Undecided. Oxford Paperbacks, 2000.
3. Jackson, Peter & Moulinier, Isabelle. Natural Language Processing for Online Applications: Text Retrieval, Extraction, and Categorization. John Benjamin's Publishing Co., 2002.
4. Lev, Iddo; MacCartney, Bill; Manning, Christopher D.; & Levy, Roger. Solving Logic Puzzles: From

Robust Processing to Precise Semantics. Online: http://nlp.stanford.edu/~wcmac/papers/robust_precise_acl04.pdf
5. Ljungberg, Anna & Schwitter, Rolf. How to Write a Document in Controlled Natural Language. Online: http://www.ics.mq.edu.au/~rolfs/papers/adcs2002-short.pdf
6. Amble, Tore. Automated Solving of Problems stated in Natural Language. Online: http://www.nik.no/1995/amble.pdf
7. Bird, Steven; Klein, Ewan; Loper, Edward. NLTK Tutorial: Parsing. Online: http://nltk.sourceforge.net/tutorial/parsing.pdf
8. Rips, LJ. "The Psychology of Knights and Knaves." Cognition. 1989 Mar;31(2):85-116.
9. Kodaganallur, V. "Incorporating language processing into Java applications: a JavaCC tutorial." IEEE. Volume 21, Issue 4, 2004. p 70-77.
10. http://www.cs.princeton.edu/~appel/modern/java/JLex/
11. http://www.cs.princeton.edu/~appel/modern/java/CUP/
12. http://www.cs.purdue.edu/jtb/ ; http://compilers.cs.ucla.edu/jtb/
13. https://javacc.dev.java.net/
14. http://alloy.mit.edu
15. http://www.hku.hk/cgi-bin/philodep/knight/puzzle
16. Appel; Palsberg. Modern Compiler Implementation in Java. Cambridge University Press, 2002.
17. Clarke, E.M. Model checking. MIT Press, 1999.

Appendix A - Alloy K&K grammar solver (static stub)

```
module KnightKnave

abstract sig Islanders {
  type: one Type,
  claims: set Statement
}

sig Type { }

one sig Knight, Knave extends Type {}
fact { #Type = 2 }

abstract sig Statement {
  value: TruthValue
}

abstract sig TruthValue { }
one sig True, False extends TruthValue { }
//fact { #TruthValue = 2 }

// DEFINITION OF "Isa" rule
abstract sig Isa extends Statement {
  target: Islanders,
  isa: Type
}
fact {
  all i: Isa |
    i.value = True <=> i.target.type = i.isa
}

// DEFINITION OF "And" rule
abstract sig And extends Statement {
  clause: set Statement
}
fact {
  all a: And |
    a.value = True <=> all s: a.clause | s.value=True
}

// DEFINITION OF "Or" rule
abstract sig Or extends Statement {
  clause: set Statement
}
fact {
  all a: Or |
    a.value = True <=> some s: a.clause | s.value=True
}

// DEFINITION OF "Not" rule
abstract sig Not extends Statement {
```

```
  subst: Statement
}
fact {
  all a: Not |
  a.value = True <=> a.subst=Knight <=> a.subst=Knave
}

pred says (i: Islanders, s: Statement) {
  s in i.claims
}

// knights say true things and knaves say false things
fact {
  all i: Islanders | all s: i.claims |
    i.type = Knight <=> s.value = True
}

pred solve () { }


// Puzzles go below this line
```

Appendix B - JTB input (knightknave.jjt)

```
options {
  JAVA_UNICODE_ESCAPE = true;
  VISITOR=true;
}

PARSER_BEGIN(KnightKnaveParser)
  public class KnightKnaveParser {
  }
PARSER_END(KnightKnaveParser)

SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

TOKEN :
{
  < IS: "is" >
| < A: "a" >
| < AND: "and" >
| < OR: "or" >
| < EITHER: "either" >
| < NOT: "not" >
```

```
| < KNAVE: "knave" >
| < KNIGHT: "knight" >
| < SAYS: "says" >
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}

/****************************************
* The Grammar Starts Here               *
****************************************/
void Goal() :
```

```
{}
{
  KnightKnaveProblem()
  <EOF>
}

void KnightKnaveProblem() :
{}
{
  ( Identifier() SaysLiteral() Statement() )*
}

void Statement() :
{}
{
  LOOKAHEAD(6)
  SimpleStatementAndSimpleStatement()
| SimpleStatementOrSimpleStatement()
| SimpleStatement()
}

void SimpleStatement() :
{}
{
  Identifier() AssignmentExpression() Islander()
}

void AssignmentExpression() :
{}
{
  LOOKAHEAD(2)
  IsLiteral() ALiteral()
| IsLiteral() NotLiteral() ALiteral()
}

void SimpleStatementAndSimpleStatement() :
{}
{
  SimpleStatement() AndLiteral() SimpleStatement()
}

void SimpleStatementOrSimpleStatement() :
{}
{
  EitherLiteral() SimpleStatement() OrLiteral()
SimpleStatement()
}

void Islander() :
{}
{
  "knight"
```

```
| "knave"
}

void EitherLiteral() :
{}
{
  "either"
}

void OrLiteral() :
{}
{
  "or"
}

void IsLiteral() :
{}
{
  "is"
}

void ALiteral() :
{}
{
  "a"
}

void NotLiteral() :
{}
{
  "not"
}

void SaysLiteral() :
{}
{
  "says"
}

void AndLiteral() :
{}
{
  "and"
}

void Identifier() :
{}
{
  <IDENTIFIER>
}
```

Appendix C - BNF for knightknave.jj

NON-TERMINALS

```
Goal                                 ::=    KnightKnaveProblem <EOF>
KnightKnaveProblem                   ::=    ( Identifier SaysLiteral
Statement )*
Statement                            ::=    SimpleStatement
                                     |
        SimpleStatementAndSimpleStatement
                                     |
        SimpleStatementOrSimpleStatement
SimpleStatement                      ::=    Identifier
AssignmentExpression Character
AssignmentExpression                 ::=    IsLiteral ALiteral
                                     |      IsLiteral NotLiteral ALiteral
SimpleStatementAndSimpleStatement    ::=    SimpleStatement "and"
SimpleStatement
SimpleStatementOrSimpleStatement     ::=    "either" SimpleStatement "or"
SimpleStatement
Islander                             ::=    "knight"
                                     |      "knave"
IsLiteral                            ::=    "is"
ALiteral                             ::=    "a"
NotLiteral                           ::=    "not"
SaysLiteral                          ::=    "says"
Identifier                           ::=    <IDENTIFIER>
```

Appendix D - Visitor Class

```
package visitor;
import syntaxtree.*;
import java.util.*;
import java.io.*;

/**
 * Provides default methods which visit each node in the
tree in depth-first
 * order.  Your visitors may extend this class.
 */
public class KnightKnaveParserVisitor extends
GJDepthFirst<Object,Object> {
   private int statementNumber = 0;
private String StaticObjectlloy0 =  "module KnightKnave \n
\n abstract sig Islanders { \n ";
private String StaticObjectlloy1 = " type: one Type, \n
claims: set Statement \n }";
private String StaticObjectlloy2 = " sig Type { } \n one
sig Knight, Knave extends Type {}";
private String StaticObjectlloy3 = " fact { #Type = 2 } \n
abstract sig Statement { \n value: TruthValue \n }";
private String StaticObjectlloy4 = " abstract sig
TruthValue { } \n  one sig True, False extends TruthValue {
} \n";
private String StaticObjectlloy5 = " abstract sig Isa
extends Statement { \n  target: Islanders, \n isa: Type \n
} \n";
```

```
private String StaticObjectlloy6 = " fact { \n all i: Isa |
\n  i.value = True <=> i.target.type = i.isa \n } \n";
private String StaticObjectlloy7 = " abstract sig And
extends Statement { \n clause: set Statement \n } \n";
private String StaticObjectlloy8 = " fact { \n  all a: And
| \n a.value = True <=> all s: a.clause | s.value=True \n }
\n";
private String StaticObjectlloy9 = " abstract sig Or
extends Statement { \n clause: set Statement \n } \n";
private String StaticObjectlloy10 = " fact { \n all a: Or |
\n a.value = True <=> some s: a.clause | s.value=True \n }
\n";
//private String StaticObjectlloy11 = " abstract sig Not
extends Statement { \n subst: Statement \n }";

private String StaticObjectlloy11 = " abstract sig Not
extends Statement { \n target: Islanders, \n isnota: Type
\n } \n";
private String StaticObjectlloy12 = " fact { \n all i: Not
| \n  i.value = False <=> i.target.type = i.isnota \n }
\n";

//private String StaticObjectlloy11 = " abstract sig Not
extends Statement { \n subst: Type \n }";
//private String StaticObjectlloy12 = " fact { \n all a:
Not | \n a.value = True <=> a.subst.value=False \n } \n";
//private String StaticObjectlloy12 = " fact { \n all a:
Not | \n  a.value = True <=> a.subst=Knight <=>
a.subst=Knave \n } \n";

private String StaticObjectlloy13 = " abstract sig Same
extends Statement \n { target1, target2: Islanders \n } \n
";
private String StaticObjectlloy14 = " fact { \n all a: Same
| \n a.value = True <=> (a.target1.type = a.target2.type)
\n } \n";
private String StaticObjectlloy15 = " abstract sig
Different extends Statement { \n target1, target2:
Islanders \n } \n";
private String StaticObjectlloy16 = " fact { \n all a:
Different | \n a.value = True <=> (a.target1.type !=
a.target2.type) \n } \n";
private String StaticObjectlloy17 = " abstract sig WouldSay
extends Statement { \n  type: Type, \n  subject: Statement
\n } \n ";
private String StaticObjectlloy18 = " fact { \n all a:
WouldSay | \n a.value = True <=> (a.subject.value=True <=>
a.type=Knight) \n } \n";
private String StaticObjectlloy19 = " pred says (i:
Islanders, s: Statement) { \n s in i.claims \n } \n";
```

```
private String StaticObjectlloy20 = " fact { all i:
Islanders | all s: i.claims | \n i.type = Knight <=>
s.value = True \n } \n pred solve () { } \n";
   String KinghtOrKnave;
   String firstStatement;
   int scope = 0;
   String secondStatement;
   List list = new LinkedList();
   List statementIDList = new LinkedList();
   List statementList = new LinkedList();
   List IdentifierList = new LinkedList();

   public void createALS() throws Exception {
     FileOutputStream out = new
FileOutputStream("KnightKnave.als");
     PrintStream p = new PrintStream(out);
     try {
       p.println(StaticObjectlloy0);
       p.println(StaticObjectlloy1);
       p.println(StaticObjectlloy2);
       p.println(StaticObjectlloy3);
       p.println(StaticObjectlloy4);
       p.println(StaticObjectlloy5);
       p.println(StaticObjectlloy6);
       p.println(StaticObjectlloy7);
       p.println(StaticObjectlloy8);
       p.println(StaticObjectlloy9);
       p.println(StaticObjectlloy10);
       p.println(StaticObjectlloy11);
       p.println(StaticObjectlloy12);
       p.println(StaticObjectlloy13);
       p.println(StaticObjectlloy14);
       p.println(StaticObjectlloy15);
       p.println(StaticObjectlloy16);
       p.println(StaticObjectlloy17);
       p.println(StaticObjectlloy18);
       p.println(StaticObjectlloy19);
       p.println(StaticObjectlloy20);
       for (int i=0; i < list.size(); i++){
         p.println(list.get(i));
       }
       String ID = "\n one sig ";
       for(int i=0; i < IdentifierList.size(); i++){
         ID += " " + IdentifierList.get(i);
             if(i != (IdentifierList.size()-1)){
                 //p.print(",");
                 ID += ",";
             }
       }
```

```
        ID += " extends Islanders{} \n";
        p.println(ID);
        p.println("\n run solve for " + scope + " but " +
scope + " Islanders, " + statementNumber + " Statement ");
        p.close();
      } catch (Exception e) {
        System.err.println("Error writing to file.");
      }
   }
   //
   // Objectuto class visitors--probably don't need to be
overridden.
   //
   public Object visit(NodeList n, Object argu) {
      Object _ret=null;
      int _count=0;
      for ( Enumeration<Node> e = n.elements();
e.hasMoreElements(); ) {
         e.nextElement().accept(this,argu);
         _count++;
      }
      return _ret;
   }

   public Object visit(NodeListOptional n, Object argu) {
      if ( n.present() ) {
         Object _ret=null;
         int _count=0;
         for ( Enumeration<Node> e = n.elements();
e.hasMoreElements(); ) {
            e.nextElement().accept(this,argu);
            _count++;
         }
         return _ret;
      }
      else
         return null;
   }

   public Object visit(NodeOptional n, Object argu) {
      if ( n.present() )
         return n.node.accept(this,argu);
      else
         return null;
   }

   public Object visit(NodeSequence n, Object argu) {
      Object _ret=null;
      int _count=0;
```

```
      for ( Enumeration<Node> e = n.elements();
e.hasMoreElements(); ) {
         e.nextElement().accept(this,argu);
         _count++;
      }
      return _ret;
   }

   public Object visit(NodeToken n, Object argu) { return
null; }

   //
   // User-generated visitor methods below
   //

   /**
    * f0 -> KnightKnaveProblem()
    * f1 -> <EOF>
    */
   public Object visit(Goal n, Object argu) {
      Object _ret=null;
      n.f0.accept(this, argu);
      n.f1.accept(this, argu);
      return _ret;
   }

   /**
    * f0 -> ( Identifier() SaysLiteral() Statement() )*
    */
   public Object visit(KnightKnaveProblem n, Object argu) {
      Object _ret=null;
      n.f0.accept(this, argu);
      Identifier e;
        NodeListOptional g = (NodeListOptional)n.f0;
      String SimpleObjectlloy = "\nfact {";
      for(int i=0; i < g.size(); i++){
              NodeSequence f =
(NodeSequence)n.f0.elementAt(i);
              e = (Identifier)f.elementAt(0);
              NodeSequence y =
(NodeSequence)n.f0.elementAt(i);
              Statement z = (Statement)y.elementAt(2);
               if (statementIDList.get(i) == z ){
                SimpleObjectlloy += "  says (" + e.f0 + ",
"+ statementList.get(i) +" ) \n" ;
               }
         }
        SimpleObjectlloy += "} \n";
        list.add(SimpleObjectlloy);
      return SimpleObjectlloy;
```

```
    }

    /**
     * f0 -> SimpleStatementObjectndSimpleStatement()
     *       | SimpleStatementOrSimpleStatement()
     *       | SimpleStatement()
     */
    public Object visit(Statement n, Object argu) {
             String State = (String)n.f0.accept(this, argu);
             statementList.add(State);
             statementIDList.add(n);
             return (State);

    }

    /**
     * f0 -> Identifier()
     * f1 -> Objectssignment Expression()
     * f2 -> Islander()
     */
    public Object visit(SimpleStatement n, Object argu) {
       Object _ret=null;
       String id = (String)n.f0.accept(this, argu);
       String assignObj = (String)n.f1.accept(this, argu);
       String assignProp = (String)n.f1.accept(this, argu);

       if (assignProp == "Isa"){
             assignProp = "isa";
       }
       String characte = (String)n.f2.accept(this, argu);
       statementNumber++;
       String StatementID = "S"+statementNumber;
       String SimpleObjectlloy;
       if (assignObj != "Not"){
             SimpleObjectlloy = "one sig " + StatementID + "
extends " + assignObj + "{} \n { \n  " + id + " in target
\n  " + assignProp + " = " + characte + "\n }";
       }else{
             SimpleObjectlloy = "one sig " + StatementID + "
extends " + assignObj + "{} \n { \n  " + id +" in target \n
isnota = " + characte + "\n }";
       }
       list.add(SimpleObjectlloy);
       return StatementID;
    }

    /**
     * f0 -> IsLiteral() ObjectLiteral()
     *       | IsLiteral() NotLiteral() ObjectLiteral()
     */
```

```
   public Object visit(AssignmentExpression n, Object argu)
{
      Object _ret=null;
      n.f0.accept(this, argu);
      NodeSequence g = (NodeSequence)n.f0.choice;
      String Lit = "";
      if (g.size() == 2){
                  Lit = "Isa";
      }else{
                  Lit = "Not";
      }
      return Lit;
   }

   /**
    * f0 -> SimpleStatement()
    * f1 -> ObjectndLiteral()
    * f2 -> SimpleStatement()
    */
   public Object visit(SimpleStatementAndSimpleStatement n,
Object argu) {
      Object _ret=null;
      String Statement1 = (String)n.f0.accept(this, argu);
      n.f1.accept(this, argu);
      String Statement2 = (String)n.f2.accept(this, argu);
      statementNumber++;
      String StatementID = "S" + statementNumber;
      String CompdAlloy = "one sig " + StatementID + "
extends And { } \n { \n clause = " + Statement1 + " + " +
Statement2 + "\n }";
      list.add(CompdAlloy);
      return StatementID;
   }

   /**
    * f0 -> EitherLiteral()
    * f1 -> SimpleStatement()
    * f2 -> OrLiteral()
    * f3 -> SimpleStatement()
    */
   public Object visit(SimpleStatementOrSimpleStatement n,
Object argu) {
      Object _ret=null;
      n.f0.accept(this, argu);
      String Statement1 = (String)n.f1.accept(this, argu);
      n.f2.accept(this, argu);
      String Statement2 = (String)n.f3.accept(this, argu);
      statementNumber++;
      String StatementID = "S" + statementNumber;
```

```
      String ConjObjectlloy = "one sig " + StatementID + "
extends Or {} \n { \n clause = " + Statement1 + " + " +
Statement2 + "\n }";
      list.add(ConjObjectlloy);
      return StatementID;
   }

   /**
    * f0 -> "knight"
    *       | "knave"
    */
   public Object visit(Islander n, Object argu) {
      Object _ret=null;
      n.f0.accept(this, argu);
      String charDef = n.f0.choice.toString();
      if (charDef == "knight"){
            charDef = "Knight";
      }else{
                  if (charDef == "knave"){
                        charDef = "Knave";
                  }
      }
      return charDef ;
   }

   /**
    * f0 -> "either"
    */
   public Object visit(EitherLiteral n, Object argu) {
      Object _ret=null;
      n.f0.accept(this, argu);
      return _ret;
   }

   /**
    * f0 -> "or"
    */
   public Object visit(OrLiteral n, Object argu) {
      Object _ret=null;
      n.f0.accept(this, argu);
      return _ret;
   }

   /**
    * f0 -> "is"
    */
   public Object visit(IsLiteral n, Object argu) {
      Object _ret=null;
      n.f0.accept(this, argu);
      return _ret;
```

```
}

/**
 * f0 -> "a"
 */
public Object visit(ALiteral n, Object argu) {
   Object _ret=null;
   n.f0.accept(this, argu);
   return _ret;
}

/**
 * f0 -> "not"
 */
public Object visit(NotLiteral n, Object argu) {
   Object _ret=null;
   n.f0.accept(this, argu);
   return _ret;
}

/**
 * f0 -> "says"
 */
public Object visit(SaysLiteral n, Object argu) {
   Object _ret=null;
   n.f0.accept(this, argu);
   return _ret;
}

/**
 * f0 -> "and"
 */
public Object visit(AndLiteral n, Object argu) {
   Object _ret=null;
   n.f0.accept(this, argu);
   return _ret;
}

/**
 * f0 -> <IDENTIFIEObject>
 */
public Object visit(Identifier n, Object argu) {
   Object _ret=null;
     n.f0.accept(this, argu);

   for(int i=0; i < IdentifierList.size(); i++){
         String IDL = (String)IdentifierList.get(i);
          if(IDL == n.f0.toString()){
           return n.f0.toString();
          }
```

```
        }
        IdentifierList.add(n.f0.toString());
        scope++;
        return n.f0.toString();
    }

}
```

Appendix E – Sample Test Cases

Test 1
```
Zoey says Mel is a knave
Mel says Zoey is not a knave and Mel is not a knave
```

Test 15
```
Betty says Peggy is a knave
Peggy says Betty is a knight and Peggy is a knight
```

Test 18
```
Alice says either Ted is a knave or Alice is a knight
Ted says either Alice is a knight or Ted is a knight
```

Test 88
```
Zippy says Zeke is not a knave
Zeke says Bill is a knave
Bill says either Bill is a knight or Zeke is a knight
```

Test 123
```
Zippy says either Bill is a knight or Alice is a knave
Alice says Bob is a knave
Bill says Alice is a knave
Bob says either Bill is a knave or Bob is a knight
```

Test 182
```
Sally says either Bart is a knight or Rex is a knight
Rex says Bart is a knave
Joe says either Bart is a knave or Rex is a knave
Carl says Rex is a knave
Bart says Carl is a knave and Rex is a knave
```