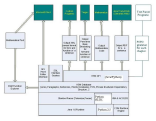
A Knowledge Management System for Mathematical Text



A University Thesis presented to the faculty

of

California State University East Bay

In Partial Fulfillment

of the requirements for the Degree

Master of Science in Computer Science

by

Arvind Punj

September, 2009

A Knowledge Management System for Mathematical Text

by
Arvind Punj

Approved: Date:

_____ _____
_____ _____
_____ _____

Acknowledgements

I would like to express my gratitude to Dr. Christopher Morgan for providing mentorship, guidance, and support throughout this entire thesis. It has been an enormous learning experience writing this thesis and I am grateful to be able to contribute to the KSM project. In addition, I would like to thank Dr. Lawrence Morgan for providing linguistic guidance on this thesis.

Table Of Contents

1.0 Introduction

This thesis explores issues in developing computerized systems designed to help people understand mathematical texts. The thesis is also a report on a proof-of-concept implementation of software that solves some of the problems that we identify. We survey an existing computerized formal mathematical system and study techniques and implementations currently used in the software community for natural language evaluation. The techniques and theory discussed are from the fields of linguistics, computer science, and mathematics. We deliver an implementation of a basic API as part of the development phase of the thesis. This API is built with the intention of providing a basis for future work on a "Knowledge System for Mathematics" (KSM) framework.

The idea behind this survey is to understand the theory behind software solutions currently implemented in the domain of natural language processing and to examine how current linguistic theory applies to developing better systems. We explore state-of-the-art theories in natural language processing and linguistics which need to be understood to develop a software framework. Understanding the theories help us take advantage of considerable knowledge and technology and reuse and augment the current techniques to solve issues presented during the development of an evaluation system for mathematical text. We will refer to our solution as the KSM database framework.

We start with the Coq tool (http://coq.inria.fr/) , which is one of the theorem-proving systems in wide and active use in the industry. Coq is based on "Calculus of Constructions". We touch upon this type theory ("Calculus of Constructions"). This provides a model of what can be done once text is fully analyzed. The Coq system has theorem proving which is implemented by using its formalized language specification, "Gallina". This formalization helps in highlighting the parts of the natural language text which are required to be translated and understood. In Coq, the syntax and semantics of text are addressable and computable.

Type theory has its origins around the time of the publication of the Church-Turing thesis. Type theory [Simon 1999] is one topic of discussion in the background section. Type theory presents us with a hierarchy which differentiates "individuals" from "set of individuals" and in doing so avoids Russell's paradox. Type theory can be used to avoid making contradictory statements within a system. Type theory leads to Curry-Howard isomorphism which gives us correspondence between logic and programming languages [CF58, How80].

The mathematical text that we parse is written in English language as well as consisting of embedded formalized structures which are formulae, tables, equations, etc. This formulaic notation is a more formalized notation with fixed vocabulary. The English language surrounding the formulae refers to this formalized notation in some manner. The formalized structures and the natural language form the two parts that need to be evaluated in the KSM framework.

We use Robert Ash's book "Basic Abstract Algebra" as an example reference to be evaluated for this thesis. This presents us with a starting text for understanding abstract algebra, which is a key foundational course for an undergraduate mathematics degree program. The mathematics undergraduate students at this level are our audience for this initial KSM database-system. We survey grammars and techniques used by the tools such as Stanford's natural language parser. In this thesis we start with mention of the first published grammar known to linguistics, which is by the Sanskrit grammarian Panini. We count it as the origin of natural language grammars. This is to say the origin of natural language grammatical analysis and description. The ancient tradition ties in closely with the kinds of grammatical analysis proposed by Chomsky and his followers, in modern times. We go on to discuss the following kinds of grammatical analysis currently in use: Context-free grammar (CFG), as opposed to Context-sensitive grammar, Probabilistic context-free grammar (PCFG), Dependency grammar, concerned with syntactic heads and dependents (as opposed to Phrase structure grammar) and two competing current theories of grammar: "Head-driven Phrase Structure Grammar" (HPSG) and "Role and Reference Grammar" (RRG). Natural language processing is introduced to set the stage for discussion of the Stanford natural language parser.

We propose to store the representation of the evaluation in relational types in a database. The other representations, both of which are widely in use in the industry, are representations using XML and LaTex formats.

We use Extreme programming methodology as our software methodology for software development portion of this thesis. Extreme programming is an Agile methodology. This Agile methodology helped us in getting feedback after completion of an experiment. The experiments were either theory-based write-ups or implementation of a software system to prove a concept or implement an API for the KSM framework. We present object-oriented as well as function-based approaches and draw conclusions about their usage in the context of implementing software for the KSM framework.

We evaluate the structure of mathematical text using a multi-dimensional framework which has physical layout, form, and function as three main dimensions. The physical dimension represents the physical layout of the text. The form dimension here refers to the syntax of mathematical text. The functional dimension refers to the semantic properties of natural language such as object, subject. The form dimension's addressability is touched upon using linguistic and positional properties of a sentence and word. The functional dimension's addressability is built on higher order constructs, such as parts of speech, subject, object, and positional properties of context markers, such as theorem definition, theorem proof, section beginning, etc.

The evaluation at the physical layout, form, and function dimensions makes use of different algorithms, Turing machines, and grammar. The algorithms which are touched upon in these dimensions correspond to segmentation of words, tagging of parts of

speech, creation of phrase structure, creation of dependency structures, and inferring PCFG grammar for each chapter.

In the course of the thesis, the Stanford natural language parser is used to evaluate form and functional dimensions as mentioned above. We infer PCFG grammar using the Natural Language Tool Kit of Python. Over the course of the thesis methodology sections try to build upon this background and help in providing solutions to generate the result section of the survey.

1.1 Problem Statement

Mathematical texts have been written for centuries using natural language. The particular area of geometry has an especially long history. In 283-306 BC Euclid expressed numerous proofs for geometrical constructions in natural language. Here is Euclid's text in Greek [Calvert 2000] showing that the biggest line segment which can fit in a circle is the diameter of the circle:

Εἰς τὸν δοθέντα κύκλον τῇ δοθείσῃ εὐθείᾳ μὴ μείζονι οὔσῃ
into the given circle to the given line not greater being
τῆς τοῦ κύκλου διαμέτρου ἴσην εὐθεῖαν ἐναρμόσαι.
than the circle's diameter (an)equal line to fit

Here is the "Bodhyana sutra" or "Bodhyana theorem" written by Bodhyana in 800 B.C., which states "A rope stretched along the length of the diagonal produces an area which the vertical and horizontal sides make together". Here is the original text in Sanskrit:

dīrghasyākṣaṇayā rajjuH pārśvamānī, tiryaDaM mānī,

cha yatpṛthagbhUte kurutastadubhayāṅ karoti.b

Here is an example from a modern day mathematical text [Fulton 1998]:

> For any scheme X, $K^{o}X$ denotes the Grothendieck group of vector bundles (locally free sheaves) on X.

This is a text by William Fulton on intersection theory. These examples illustrate how a variety of geometric concepts have been expressed over the years. The language is often very formal and carefully structured. The vocabulary consists of ordinary words of the natural language that are given special meaning.

The widespread availability of computers opens up many possibilities for assisting people in reading and writing such text. Computers are well suited to handling carefully constructed language. Indeed, modern computer software is written in very formal language and is automatically translated to language that can be directly executed on the computer. Mathematical concepts have also been expressed and understood in formal computerized systems like the COQ proof assistant [http://coq.inria.fr], which has richer, deeper structures with consistency and limited vocabulary. In such systems, proofs can be automatically checked for correctness. This makes it attractive to translate mathematical text from natural language to formal systems.

If we do not wish to completely translate mathematical text into such a formal system, computer software can still provide a significant assist in its analysis.

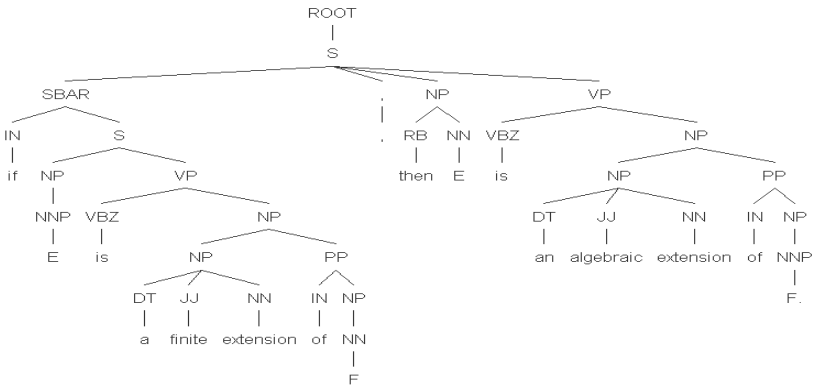
Noun phrases which are used in mathematical text convey a consistent type or category of mathematical concept, e.g. "(NP (DT a) (JJ commutative) (NN division) (NN ring))", "(NP (DT a) (JJ cyclic) (NN group))", "(NP (DT a) (JJ cyclic) (NN

subgroup))". Here we have used computer software to parse mathematical text in abstract algebra [Ash 2000].

Also consider the consistency of the "if _ then _", "for all" and "such that" constructions in mathematical text. These are consistent in the semantics they convey and the vocabulary they use. They delineate the natural language boundaries of clauses and phrases. Here is a sample "if_then" segment from [Ash 2000]:

"if E is a finite extension of F, then E is an algebraic extension of F."

Here is how it is analyzed by a current state-of-the-art natural language parser [Stanford 2009]:



The parser separated the "if E is a finite extension of F" as a separate clause (labeled SBAR), branching to the left, and the rest of the sentence in other branches on the right, including a verb phrase starting with the word "is". This is a very common sequence in mathematical text that corresponds to the logical structuring of mathematical ideas.

The parser further found the noun phrases: "E", "F", "a finite extension", "an algebraic extension", and larger noun phrases obtained by adding prepositional phrases to the last

two phrases. Note that it erroneously identified “then E” as a noun phrase. With that one exception, it correctly identified the mathematical concepts (by noun phrases) in this text.

The parsers we used are publicly available and are based upon modern linguistic theories. However, they are not perfect. At the same time, modern linguistic theory continues to advance in ways that are compatible with modern computer science. The special sub-domain of computer science called Natural Language Processing (NLP) brings these fields together.

Given the wealth of mathematical texts written in natural language and the availability of work in modern linguistic theory that has been done in close collaboration with computer scientists, the problems that we wish to solve in this thesis are to 1) identify current software that can assist with the analysis of mathematical text, 2) identify deficiencies with this software, 3) identify modern linguistic theories that might be helpful in overcoming these deficiencies, and 4) design and implement a proof-of-concept system that allows mathematicians to effectively investigate concepts and structures of mathematical texts.

Elaborating on the fourth point, the implementation of software tries to solve the following sub-problems:

-Extract unique vocabulary from mathematical text.
-Extract lexical and phrase fragments found in statements in mathematical text.
-Provide interfaces for searching through lexical and phrasal fragments.
-Extract dependencies between words in a sentence.

-Provide a common method for storing and accessing these results.

1.2 Solution

We develop the KSM framework to create a model in which we can express types and axioms with constraints, domain and range of the solution space of our problem. Our problem domain is parsing of mathematical text, so we have a database to model the domain and range of types. We also provide a function set which we use to build this database and parse the mathematical text.

Sentences used in mathematical text have syntactical structure (or form) that is mapped to the meaning (semantic or function) of the sentence. The physical layout of mathematical text consists of lines, sentences, words, paragraphs, section, pages, chapters, etc. The algorithms for segmentation of words, tagging of parts of speech, creation of phrase and dependency structures are implemented in the physical layout, form and function dimensions. The types generated from the segmentation, tagging and creation tasks are stored in relational format or are returned as values from the function set of the KSM framework.

We use algorithms implemented in the Stanford parser for segmentation, tagging, and phrase creation tasks. The segmentation is done at line, sentence, and word levels. The lines, sentences, and words are stored in relational structures (tables). We use relational tables for recording the layout of mathematical text. SQL queries help to provide addressability of this layout. SQL tables provide transient states to address different

types, such as sentence to parsed sentence. These two states are stored next to each other as part of the type (table). The SQL interface of the relational structures (tables, queries) leads us to the creation of new types, such as paragraphs, sections, page, chapter and theorems, etc.

The KSM framework is concerned with building the physical layout, form, and functional dimensions. The three dimensions mentioned are implemented using multiple Turing machines and training datasets like Penn-Treebank. The KSM framework uses function libraries and existing implementations provided by the Stanford natural language parser.

The KSM system outputs phrase, dependency structures of the simple text, and PCFG (Probabilistic context free grammar) for each chapter. The representation formats used for these outputs use different standard markup languages. The phrase structure output is represented in bracketing syntax (as used in lists in LISP programming language). Phrase structure output is also represented in XML, RDF (Resource Definition format) form, and as a Mathematica formula. The inferred grammar for each chapter is represented in Backus-Naur format.

The KSM framework has a set of functional APIs which permits the extraction of information regarding the text. This information extraction is done by parsing of the mathematical text while maintaining the physical layout of the text. The physical layout holds some key information to extract fragments which are important to mathematicians e.g. theorem statement and proof or definitions of terms. The parsed information is maintained in a database which has tables in the database to store the sentence parse and also has parts of sentences classified as subject, modifier and direct object. The sentence

parse is part of the form dimension and the dependency parse is part of the function dimension of the KSM framework. The parsing primarily uses Stanford's English PCFG parser and dependency parser. The KSM framework also has interfaces for searching for particular (regular) expressions in mathematical text, as well as querying trees or phrasal information.

The framework can also use Python NLTK for applying functions on text input. The Python NLTK has parsers of its own for PCFG parsing using different kinds of chart parsing. A description of various functions in the NLTK is provided in detail in the section on Python NLTK. The output representations for the parse of a sentence are expressed in XML notation as well as the default bracketed notation. The output representations can be further customized using Tex according to the user interface desired by the framework.

2.0 Survey and Background

2.1 Grammars

In order to track the evolution, differences, and similarities in grammars written for modern natural languages, it helps us to take note of a grammar written for an ancient language. Brief mention of that grammar, as background, takes us to the origins of grammar writing itself. Panini's grammar for the natural language, Sanskrit, from around 400 B.C, is thought to be the first published grammar. Panini used symbolic logic to write his grammar using symbols which are very natural to computer scientists today. The grammar rules, in eight books or chapters (almost 4000 rules), describe the structure of the Sanskrit language. Present day generative grammarians see Panini's grammar as a generative grammar [Kiparsky Jan. 2002 On the Architecture of Panini's Grammar]. The rules are built upon basic categories of nouns, verbs, vowels, and consonants. The construction of rules is similar to modern-day Backus-Naur form used in computer science. The grammar rules of Sanskrit are transformational and recursive, and are context sensitive. Modern-day grammar rules are transformational, recursive, and context sensitive. Describing grammatical rules in this way, in the modern age, comes from the work of Noam Chomsky. Seeing grammatical rules in these terms helps us in our goal of evaluating natural language.

Chomsky in his 1956 paper "Three Models for the Description of Language" makes parallels between mathematical theory and grammar. According to Chomsky, grammar is

based on a finite number of observed sentences, and it projects this set to an infinite set of grammatical sentences by establishing general laws framed in terms of phonemes, words and phrases. Chomsky has suggested a relationship that grammar makes between a set of grammatical sentences and set of observed sentences. The first model for the description of language given is of a finite state Markov process. This is purely word based. The second model presented by Chomsky is the phrase structure method of describing a language. This method describes the English language based on linguistic classes like noun phrases and verb phrases. The third model was that of transformational grammar. Transformational grammar is based on the linguistic class patterns in a sentence e.g. (NP, Auxiliary, V and NP) to which a function or transformation is applied which reorders and adds certain linguistic classes and words. For example [Chomsky '56], in transforming "the man ate the food" to "the food was eaten by the man" will uses the linguistic class pattern (NP1-Auxillary-V-NP2) to (NP2-Auxillary-be-en-V-by-NP1). This gives a derived description from a valid sentence in English to another valid sentence in English. In this survey we have make use of Chomsky normal form (or CNF) which is a representation in Backus-Naur form of context-free grammar. The representation has one non-terminal on the left-hand side and two or less non-terminals on the right-hand side of each production.

Sag and Wasow (2001), in their book "Syntactic Theory, A Formal Introduction (first edition)", give a history of the development of grammatical theories in the generative

tradition, from their perspective. That perspective is firmly grounded in the theory of Head-driven Phrase Structure Grammar. They see two distinct approaches: Transformational Generative Grammar and Constraint-Based Lexicalist Grammar. Because of the importance of Head-driven Phrase Structure Grammar, and closely related theories, in natural language processing, we follow their perspective in the coming paragraphs as we give a quick tour of the sequence of development of some of the grammatical theories that they discuss.

Generative grammar in modern times began in the 1950s with the work of Chomsky. In transformational generative grammar, a transformational derivation involves two main phases. The first phase generates the phrase structure using context-free grammar (CFG). The second step applies transformational rules to map these into other phrase structures [Sag, Wasow and Bender 2003 "Syntax Theory. A Formal Introduction (second edition)"].

Bresnan's "Realistic" Transformational Grammar (Bresnan 1978) and Montague Grammar (Montague 1970) paved the way for the development of non-transformational generative frameworks [Sag, Wasow and Bender 2003].

Bresnan (1982) went on to develop Lexical functional grammar (LFG) while others developed Generalized Phrase Structured Grammar (GPSG) , and Categorical Grammar (CG). Bresnan's earlier work and the contribution of Montague Grammar paved way for these three non-transformational theories of grammar. Dependency grammar (DG),

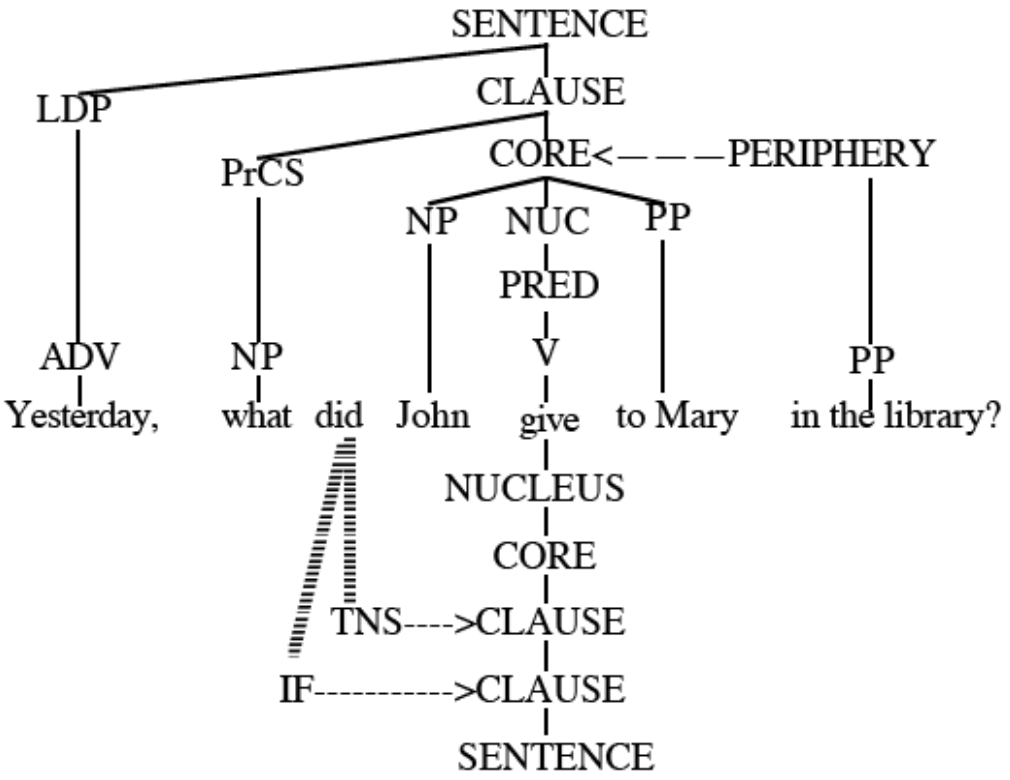
originally outside of the generative tradition, has contributed to the development of theories of generative grammar theories. Another important grammatical theory for natural languages is Construction Grammar which has its historical roots in Berkeley, particularly in the work of Charles J. Fillmore. Construction grammar has a basis in the secondary process which takes place when general rules and principles interact with each other. Head-driven Phrase Structure Grammar (HPSG) is a direct outgrowth of GPSG. Relational Grammar (RG) and Optimality Theory (OT) are two theories of natural language which have also influenced the development of HPSG, in addition to the non-transformational generative theories already mentioned here which either set the stage for the development of HPSG, or have developed in parallel to it, influencing its development. [Sag, Wasow, and Bender 2003].

Dependency grammar (Tesniere 1959) examines relationships between words in a sentence. The relationships are syntactic dependencies involving the modifications or further specification of a syntactic head by a syntactic dependent, such as the modification of a noun by an adjective, or the further specification of a verb by the arguments of the verb. These relationships map the syntax of a sentence to some of the semantic aspects of the sentence.

The Minimalist program (MP) is the most recent development within the still on-going Transformational Generative Grammar tradition. MP strives to find optimality in natural languages. There are two dimensions of MP grammar: phonological and logical.

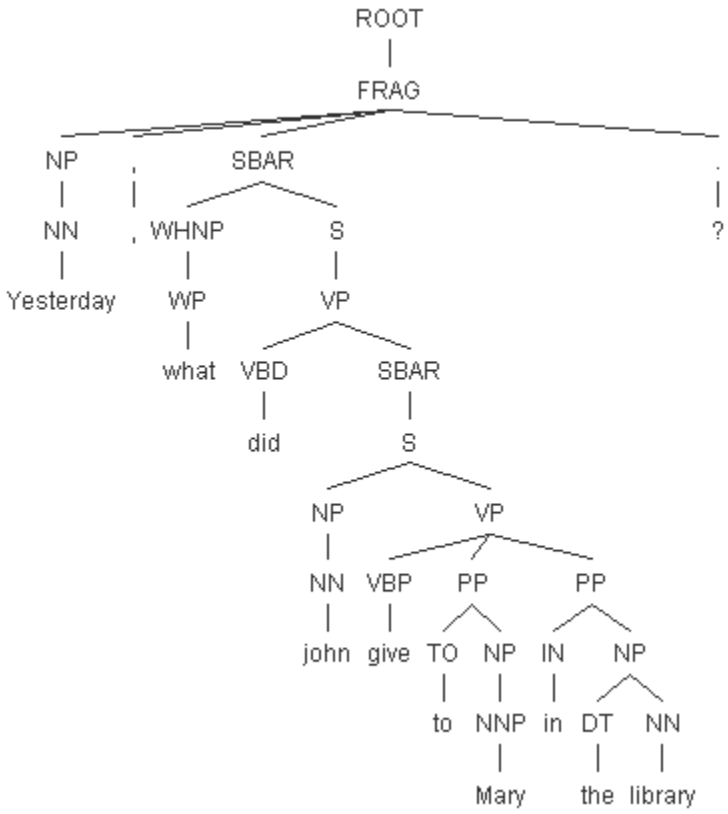
Role and Reference grammar (RRG) is another actively researched theory of grammar. It is a single-layer theory at the syntax level. This layer is represented by the actual form of the sentence [Van Valin 1993]. Here is a brief introduction to the architecture of the theory. RRG has a layered structure of the clause. The layers of a clause consist of a nucleus which consists of a predicate and “the core”. "The clause" contains "the core " which contains the predicate and the arguments of the predicate.

Here is a sentence analyzed using the formalisms of RRG. The sentence that is analyzed below is in English, but RRG presents us with an especially universal view of clause structure, which takes into account the diversity of the world's languages, to a greater extent than other theories, for most part. The greater universality of RRG has the potential to give greater insight into the clause structure of English. We have reason to believe that this pays off when it comes to parsing. See section 2.1.5 further below. RRG gives a common representation for the syntactical categories and structures for different languages such as English, German, Lakhota, Tagalog, Dyirbal and Barai.



[Figure 1]

The two arguments of the predicate “give” are “John” and “Mary” and so are parts of the “CORE”, along with the predicate “give”. “John give to Mary in the library?” is a clause, while the prepositional phrase “in the library?” is the "periphery". This is seen differently in the tree diagram produced as the syntactic parse of the sentence by Stanford’s parser below. The PrCS is the pre-core slot that occurs before the core starts, while “did” is labeled by the two clausal operators of tense (TNS) and illocutionary force (IF), as “did” conveys the tense and conveys the interrogative illocutionary force of the sentence.


ROOT
FRAG
NP
,
SBAR
.
NN
,
WHNP
S
?
Yesterday
WP
VP
what
VBD
SBAR
did
S
NP
VP
NN
VBP
PP
PP
john
give
TO
NP
IN
NP
to
NNP
in
DT
NN
Mary
the
library

[Figure 2]

2.1.1 English in Context-Free Grammar (CFG)

The English language is more complex than formal programming languages in computer science like Java, and Python. Programming languages apply constraints to the user's input to the computer, so that the usage of the language syntax can precisely map to the semantic rules or functional dimension of the language. Programming language syntax is mostly expressed using CFG. The CFG approach to grammar is introduced in compiler or theory of automata class in computer science courses [Sipser 2005].

English in mathematical text books can be parsed using context free grammars with categories derived from parts of speech (Noun, Verb, Noun phrases, Verb phrases). These context free languages with parts of speech categories loosely map to the n-gram word patterns in sentences [Jurafsky 2008]. The n-gram word pattern of a sentence makes a sentence unique, and in mathematical text conveys precisely one meaning (in functional dimension) most of the time. The Stanford parser uses CFG fragments and word patterns to help parse English text [Manning 1999].

2.1.2 Probabilistic Context-Free Grammar (PCFG)

Context-free grammar or CFG consists of a set of production rules with terminals and non-terminals. Probabilistic context-free grammar is defined as a 4-tuple consisting of (N, Σ, R,S). Here N is a set of non-terminals, Σ is a set of terminals, R is a set of production rules, and each non-terminal on the RHS in a rule has a probability assigned to it. This probability\ is assigned based on the occurrence of this non-terminal in the tagged corpora.

There is a process to generate PCFG and a process to map the PCFG to an input sentence. The generation of PCFG can be done using a tagged corpus and a set of CFG rules.

The Stanford parser uses probability of each production to calculate the probability of the parsing of a sentence. The production probability is determined from two view points. These are based on the joint probability distribution of inside and outside probability. The inside probability is calculated based on the words and phrasal parents assigned to the words in the sentence. The outside probability is based on the surrounding words and phrasal trees around the words in the sentence. The equations given by [Manning, Schutze 1999] depict a clear explanation of the mechanics of this calculation. Here are the two equations taken from their work which calculate outside and inside probabilities:

$$\alpha_j(p,q) = P(w_{1(p-1),} N_{pq}{}^{j}, w_{(q+1)m} \mid G) \text{ and } \beta_j(p,q) = P(w_{pq} \mid N_{pq}{}^{j}, G)$$

α is the outside probability and β is the inside probability. The outside probability is based on the joint probability of occurrence of the words around non-terminal and terminal productions of grammar G, where as the inside probability is calculated based on the probability of generating the group of words starting with non-terminal N_j.

The algorithm used to determine the best parse through a sentence using the probability model above is possible using similar algorithm used for Hidden Markov networks (HMM). In HMM we try to determine the most probabilistic path through the network. In PCFG we also need to make sure to follow the sequence of the rules applied e.g. NP-> Det N, VP->VBG, and S->NP VP. This is not true in HMM since we can pick the words at any point, and so we can generate an unacceptable sentence and give it a probability. We can get around this difference between HMM and PCFG by using a start and sink state (Manning and Schutze 1999). The start state will assign the transitions to the HMM states based on the starting words for a sentence, and then use the HMM or use the sink state if the sentence becomes acceptable at any point in time. The Viterbi algorithm can find the most likely path through a Markov network, so we can use the Viterbi algorithm to determine the most likely parse for a sentence. The inside-outside algorithm mentioned earlier is of the order $O(m^3 n^3)$, since we need to parse through the entire length of the sentence and all non-terminal branches for calculating inside and outside probabilities and then calculate the product of the two probabilities. This order mentioned above is a drawback of the inside-outside algorithm.

In the implementation of the Stanford parser PCFG generation is given using a file. This file contains the lexicon available via the corpus and the frequency of usage for all the words in the lexicon. The lexicon is defined as a set of rules which uses bigram parts of speech and phrase context to assign to a particular word. For instance the word run has more than one assignment.

There are a few disadvantages of PCFG. PCFG generated from a given corpora gives preference (probabilistic preference) to usage of a sentence in the corpus, but this usage may not be as clear to the reader of a sentence, since the context of the sentence may have been different. PCFG language model applies the same algorithmic approach to all words in the same category (noun, verb etc) when applying rules of production of the given grammar. The words in the same category have different probabilities of occurrence and so these two approaches are not in harmony with each other. Considering the tree sub-structure from a parsed tree PCFG the probability of the tree substructure is based on joint probability of its sub-trees. The joint probability gives us an approximation of the new tree structure this is an approximation since it is based on product space of tree probabilities in the corpus, hence some sub-trees encountered less in the corpus will reflect on the overall probability of the tree structure. This may conclude a wrong tree structure as having the best/highest probable path through the tree.

2.1.3 Dependency Grammar (DG)

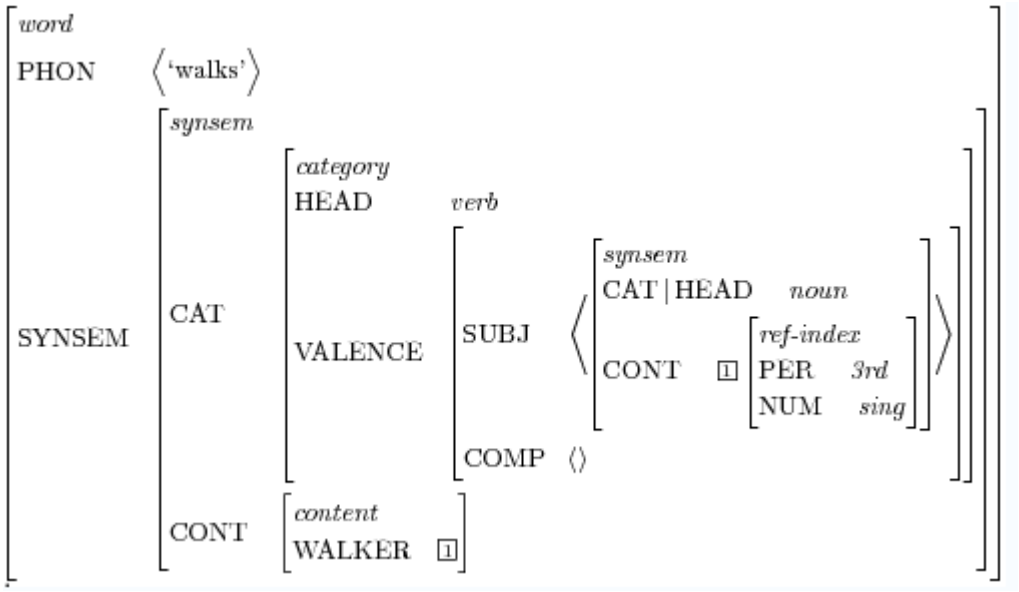
Dependency grammar is a syntactic theory which gives relationships between the syntactic words of a sentence. Dependency grammar has traditionally been depicted as the way to explain semantic meaning of sentences. One such example is subject of a verb relationship. There are many other kinds of relationships which are present apart from the subject-predicate relationship. The seminal work of [Tesniere 1959] has given the basic rules for dependency grammar.

Dependency grammar algorithm created by [Collins 1999] identifies noun phrases and the rest of the words in the sentence and calculates dependencies between them. Collins uses a probabilistic model to assign probability to the noun phrases. This model is based on a reduced sentence from the original by substituting the head word for the noun phrase and removing all punctuation. In consequence to this we have fewer words for calculating the probability. Also, Collins uses the phrase structures (categorical names) in Penn-Treebank and transforms them into a notation to assign dependency probabilities. This use of categorical tags (form) to mark functional categories helps Stanford Parser disambiguate sentence parses during parsing.

2.1.4 Head-Driven Phrase Structure Grammar (HPSG)

Head-driven Phrase Structure Grammar (HPSG) captures the knowledge given by a word in the sentence in different dimensions. These dimensions are well typed which means there are specific sub-dimensions of a dimension and these are decided before parsing of the sentence. The top level dimensions are referred to as “sorts” (Sag 1999). There are two sorts at the first level of classification “PHON” and “SYNSEM”. These denote the phonetical, syntactic and semantic information conveyed by the word. The information of interest to us here is the “SYNSEM” information which captures syntax and semantics of the word. The “SYNSEM” type is further divided into “CONTEXT”, “CONTENT” and “CATEGORY” types.

The feature structure is a chosen representation of HPSG. The feature structure is a directed graph. It can have type or category name represented as label on a directed graph and the type or category value represented in a node. This is also referred to as Attribute value Matrix (AVM). Here is a sample representation of the word “walks” in AVM format from (Pollard & Sag 1999).



[Figure 3]

Here as discussed before we have "PHON","SYNSEM" as the main categories of sorts. The "SYNSEM" tag has "CATEGORY", "CONTENT" dimensions of the SYNSEM type embedded in them. These have further the category of "HEAD" and "VALENCE". The "CONTENT" type marks the symbol we use when "walks" is used in the subject form this is done by a "WALKER" this node is connected to the "SUBJ" node's sub-type for "CONTENT". This is a depiction of the content when "walks" is used in the subject form. The "CONTENT" node of "SUBJ" has further subtypes of "PER", "NUM" which attach the requirement of using the walker in a singular, and in addressing of the sentence is in 3rd person, i.e. we refer to a walker in the subject phrase as a third person doing the walking as in "The walker walks with a cane". These constraints can also be written using simple predicate calculus with sorts represented as binary or unary predicates. This first order logic representation of HPSG built upon by the axioms of second order in

HPSG. This agreement between the subject and verb is referred to as one of the "Principles" [Pollard & Sag 1999].

2.1.5 Role and Reference Grammar (RRG)

Role and Reference Grammar is a linguistic theory which gives a common model of grammar to be reused across languages [Van Valin 1977]. This theory is based on a wide variety of languages like Lakhota, Tagalog, Dyirbal and Barai, German, English. [Guest 2007] has created a parser for English and Dyirbal which is the parsing the two languages successfully. The syntactical parsing done using RRG, strives to capture all of the common structures in the different languages such that these can be compared when required. The classification used by RRG is based on "layered structure of clause" [Van Valin 1997]. This classification has three main components "NUCLEUS", "CORE", "PERIPHERY". The "NUCLEUS" contains the predicate of the sentence. The "CORE" contains arguments to the predicate and the "PERIPHERY" contains adjunct modifiers. These three syntactic dimensions are mapped to semantic model by RRG. RRG strives to provide a bi-directional mapping between Syntax and Semantics. Here are some of the mappings for basic syntax classes [Van Valin 1997].

Semantic Classes	**Syntactic classes**
Predicate	Nucleus
Argument in semantic representation of predicate	Core argument
Non-arguments	Periphery
Predicate+Arguments	Core

Predicate+Arguments+non-Arguments	Clause(=Core+Periphery)

2.2 COQ

The Coq system is a proof assistant system which helps users in creating proofs and well-formed functions http://coq.inria.fr/. Type checking is an important part of how it works (proof). It ensures that the user is following the Coq framework rules, which include deductive reasoning rules of mathematics. We will go over few aspects of Coq which seem relevant to this thesis, for more comprehensive list of syntax and command please refer to the [Coq 2008]. See Appendix B for my independent study work and advisor's essay on this system.

The Coq system uses type checking facility for checking well-formed functions. A function is well-formed if it has a correct syntax dictated by Coq grammar and it has the correct types being passed from one sub-function to another inside the function. Types in functions are validations of the fact that the functions are in the domain of Coq framework. Coq has keyword of "Definition" which is a string which is case sensitive and is used for a purpose of adding vocabulary to the Coq system for a proof. Here is an example of its usage:

Definition testsample := fun t : nat => t*3+2.

The above string in Coq is interpreted as defining a function testsample which is defined as accepting a natural number and using it in the function f(t)=t*3+2. The period is required by Coq at the end of each line.

There are other commands in Coq for computing a function. This command has a syntax of "Eval compute in testsample 2." to compute the function we introduced using the "Definition" string. The result is specified with its type in this case it is " 5: nat".

There is a generic notation used in Coq of "A:B" which implies A is a proof for the logical formula B. This is based on the Curry-Howard Isomorphism which relates logic to type theory. The type A could be a proposition and even B could be a proposition and so this builds the framework of building propositions on top of propositions which is one of the properties of deductive mathematical proofs. Coq helps us search for existing proofs this is done using the "Search" command. The "Search" command extracts strings or sentences out of a proof. This is an mportant activity for reusing parts of a proof. The string level operations done by Coq help us reuse the proofs hidden in other bigger proofs. Here is an example of this command.

```
Search t*3.
fun t: nat => t*3+2.
```

Similar to search is "SearchPattern" command which searches for a pattern of characters.

```
SearchPattern (_ * _ + _).
fun t: nat => t*3+2.
```

These searches with the help of "SearchRewrite" command can rewrite theorems. These helps in testing and verifying various proof scenarios. The "SearchRewrite" command searches for theorems which are proving an equality.

Here is a sequence of actions a user takes once he is ready to create a new proof. The string "Theorem" or "Lemma" is used to signify the beginning of a Theorem or Lemma. The user will type in the formula, Coq will present the types needed to work on this problem and also splits the formula into sub-goals which are often sub-strings of the original formula. The user then tries to split this formula into acceptable portions which are accepted by the Coq-framework. The commands which help split the formula are called tactics. The tactics available to the user are "elim" and "case" which help user create intermediate factual statements and substitute these in the goal, introduction to new variables is required to create the intermediate factual statements. These are introduced using A:B notation where A is the variable name and B is the type of the variable. The "assert" tactic is used to introduce the intermediate statement and its goal. Other tactics like "intuition" are used by the user to reduce the context and remove tautological strings which are present at any point during proving of the Theorem or Lemma. there are other tactics in Coq-framework, please refer to [Coq 2008]. The system informs the user when the proof is complete.

Another important part of Coq is the ability in proofs to import already proven theorems using "Require" string at the beginning of a sentence. The types defined in Theorems can be reasoned with using induction this provides for reusability of types in Theorems. Pattern matching commands like "match" are used to perform string comparisons, these are helpful in writing functions. Recursive functions are also permitted in Coq.

Coq is an example of a computing system for expressions in mathematics which uses Type Theory and a formal language. Coq may help us understand how to represent mathematics at conceptual level. This corresponds to conceptual dimension of our KSM model. This points the way to include Type Theory as part of the KSM model.

2.3 Object-Oriented Approaches

Object-Oriented approach to software development have been used since the 1990s. The Object-Oriented approaches provide for interaction with domain-level concepts at design time. This helps design better systems [Coad 1991] .The Object-Oriented analysis phase lays a good foundation for representing the requirements. The relationships expressed in the Object-Oriented-analysis (OOA) phase creates a type system on which software can be built. The relationships expressed in the OOA phase are aggregation, composition, inheritance, or association. These relationships provide for different ways in which objects navigate the domain space [Booch 1994].

The KSM framework is based on object systems. There are numerous types used in the natural-language evaluation of text. These are words, noun phrases, verb phrases, clauses, sentences, paragraphs, theorem, corollary, section, page, and book. All these types perform certain set of responsibilities in the object framework. These will be discussed more in the methodology section.

Over these types or objects mentioned above, there are patterns established in the framework and in the systems it uses which promote reuse and extensibility. The object-patterns are islands of reusable objects which are fundamental to working of a software or a domain. There are numerous object patterns and these are divided into Creational, Structural and Behavioral patterns. The patterns used in KSM framework and in software

systems built to evaluate natural language systems is another area we will discuss in the methodology section.

Natural language domain itself has pieces of theory which parallel Object-Oriented systems. Parts of speech or categories of speech have different information tagged to it like variable types. Adjective phrases serving as adjectives in English language is one such example of polymorphic form of the category of speech such as Adjective which can be a single word or group of words.

The Object-Oriented approach provides for interaction mechanisms among software systems which are loosely coupled. One example of such interaction is the Template pattern [GoF 1994]. Template pattern is a series of relationships among classes which provides this loosely coupled mechanism of interaction among software systems. The Template pattern is a behavioral pattern. This design pattern helps in creating a framework in which algorithms are tied together and have a collaborative relationship among them.

2.4 Natural Language Processing

Natural Language Processing is an area of domain knowledge dealing with processing of Natural language by machines. This processing of natural language uses numerous types, algorithms, grammars and Turing machines to accomplish certain goals. There are two basic processing components, the syntax processing component and the semantic processing component. These two form the basis of lot of work done in Natural Language Processing. The goals of Natural Language Processing are several and range from understanding the text by a machine to applying the findings to other systems.

The four tasks we discuss in the future sections are tokenization, part of speech tagging, phrase structure chunking and dependency parsing. The tokenization is the task of breaking up the sentences into words. Part of speech tagging is the task of tagging a category to a word. These categories are one of the classes in English language Treebank, e.g., NN is the category assigned to nouns, VBZ to verbs etc. The phrase structure is the task of further categorizing group of words into categories like Noun phrase, Verb phrase etc. Dependency parsing is specifying semantic relations between words or group of words.

Here are the Parts Of speech tags for Penn-Treebank and the dependency hierarchy for the various dependency grammar categories.

The Penn Treebank POS tagset.

1. CC	Coordinating conjunction	25. TO	*to*
2. CD	Cardinal number	26. UH	Interjection
3. DT	Determiner	27. VB	Verb, base form
4. EX	Existential *there*	28. VBD	Verb, past tense
5. FW	Foreign word	29. VBG	Verb, gerund/present participle
6. IN	Preposition/subordinating conjunction	30. VBN	Verb, past participle
7. JJ	Adjective	31. VBP	Verb, non-3rd ps. sing. present
8. JJR	Adjective, comparative	32. VBZ	Verb, 3rd ps. sing. present
9. JJS	Adjective, superlative	33. WDT	*wh*-determiner
10. LS	List item marker	34. WP	*wh*-pronoun
11. MD	Modal	35. WP$	Possessive *wh*-pronoun
12. NN	Noun, singular or mass	36. WRB	*wh*-adverb
13. NNS	Noun, plural	37. #	Pound sign
14. NNP	Proper noun, singular	38. $	Dollar sign
15. NNPS	Proper noun, plural	39. .	Sentence-final punctuation
16. PDT	Predeterminer	40. ,	Comma
17. POS	Possessive ending	41. :	Colon, semi-colon
18. PRP	Personal pronoun	42. (	Left bracket character
19. PP$	Possessive pronoun	43.)	Right bracket character
20. RB	Adverb	44. "	Straight double quote
21. RBR	Adverb, comparative	45. `	Left open single quote
22. RBS	Adverb, superlative	46. ``	Left open double quote
23. RP	Particle	47. '	Right close single quote
24. SYM	Symbol (mathematical or scientific)	48. ''	Right close double quote

[Figure 4]

Here are the phrasal and clausal level categories defined in Penn-Treebank:

Phrase Level

ADJP - Adjective Phrase.

ADVP - Adverb Phrase.

CONJP - Conjunction Phrase.

FRAG - Fragment.

INTJ - Interjection. Corresponds approximately to the part-of-speech tag UH.

LST - List marker. Includes surrounding punctuation.

NAC - Not a Constituent; used to show the scope of certain prenominal modifiers within an NP.

NP - Noun Phrase.

NX - Used within certain complex NPs to mark the head of the NP. Corresponds very roughly to N-bar level but used quite differently.

PP - Prepositional Phrase.

PRN - Parenthetical.

PRT - Particle. Category for words that should be tagged RP.

QP - Quantifier Phrase (i.e. complex measure/amount phrase); used within NP.

RRC - Reduced Relative Clause.

UCP - Unlike Coordinated Phrase.

VP - Verb Phrase.

WHADJP - *Wh*-adjective Phrase. Adjectival phrase containing a *wh*-adverb, as in *how hot*.

WHAVP - *Wh*-adverb Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizer) or lexical, containing a *wh*-adverb such as *how* or *why*.

WHNP - *Wh*-noun Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizer) or lexical, containing some *wh*-word, e.g. *who*, *which book*, *whose daughter*, *none of which*, or *how many leopards*.

WHPP - *Wh*-prepositional Phrase. Prepositional phrase containing a *wh*-noun phrase (such as *of which* or *by whose authority*) that either introduces a PP gap or is contained by a WHNP.

X - Unknown, uncertain, or unbracketable. X is often used for bracketing typos and in bracketing *the...the*-constructions.

Clause Level

S - Simple declarative clause, i.e. one that is not introduced by a (possible empty) subordinating conjunction or a *wh*-word and that does not exhibit subject-verb inversion.

SBAR - Clause introduced by a (possibly empty) subordinating conjunction.

SBARQ - Direct question introduced by a *wh*-word or a *wh*-phrase. Indirect questions and relative clauses should be bracketed as SBAR, not SBARQ.

SINV - Inverted declarative sentence, i.e. one in which the subject follows the tensed verb or modal.

SQ - Inverted yes/no question, or main clause of a *wh*-question, following the *wh*-phrase in SBARQ.

Many software systems use various grammar formalisms to do the tasks talked about above (tokenization, part of speech, phrase structuring, dependency parsing). The Stanford Natural language parser, and Python's NLTK are some of the software we will talk about in future sections. There are representations which help people interact with the systems. These are important in Natural Language Processing as the medium of consumption for users and other systems. There are various frameworks like Jena which are using the XML representations of text to create new type systems to be processed and collaborated with for the next generation of the Web; i.e., the Semantic Web. The Semantic web uses representation forms like RDF (Resource Description Framework)

and OWL (Web Ontology language) which communicate or link to other systems on the World Wide Web. These systems will also be touched upon in the future sections.

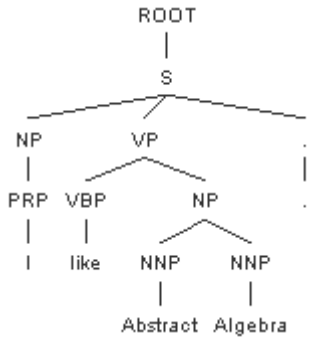
2.4.1 Tokenization

The tokenization of a sentence or a physical line in a text refers to breaking up of a sentence into words which have the intended meaning. A word is defined as "a string of contiguous alphanumeric characters with space on either side; may include hyphens and apostrophes, but no other punctuation marks." [Kucera, Francis 1967]. This definition does classify lot of words but leaves out potential words used in text. One such example in Mathematical text is the sentence "If f is injective and **a . K**, then f(a)=1= f(1), hence a = 1." Here the word **"a.K"** should be considered a word by the tokenization rule given above but sentence demarcation using period affects the tokenization. Here **"a" is considered a word or token.** Defining a unique rule to recognize the context in which this period was used would resolve this issue. The dot (.) here refers to product of a and K. Tokenization of sentences such as "A finite cyclic group generated by a is necessarily Abelian, and can be written as {1,a,a2,...,an-1} where an =1, or in additive notation, {0,a,2a,...,(n-1)a}, with na = 0." in mathematical text has problems identifying the bracketed list (type). This list type can be kept together during tokenization but is tokenized based on the sentence demarcation. The left curly braces is an identified category in Penn-Treebank and so gives us a way to define a rule within the curly braces context.

2.4.2 Tagging

The tagging of a sentence is the process of assigning part of speech to words in a sentence. This tagging process can use various techniques in Natural Language Processing. One such technique is Markov model based which is probabilistic in its approach. The other techniques like transformation based techniques are described at end of this section.

The probabilistic technique basically assigns probability to a word sequence given an already tagged corpus. Consider the Penn-Treebank which has tagged corpora for a number of sentences. This corpus can be used to break up the information in the corpus into chunks which are of word length. Each word is assigned a tag in the corpora by hand. The task of tagging using Markov technique will learn from this tagged corpus and hence predict the tags in an input sentence. Here is a simple example consider the sentence “I like Abstract Algebra” here we have tags of the sequence “PRP VBP NNP NNP”. The training of the Markov network is done based on tag probabilities and on word probability given a previous tag occurrence; e.g., if tag NNP follows tag VBP then there is a probability assigned for this occurrence and there is probability assigned to the occurrence of the word “Abstract” given the probability of tag VBP. Combining these two probabilities gives us probability of occurrence of a tag given a word occurrence.



[Figure 5]

The above description has been derived from [Manning, Schutze '99] equation on page 347 in his book on foundations of Natural language processing.

Another technique for tagging is based on transformation rules. This technique needs a initial tagging available for each word for this we need a dictionary of words with the initial tagging and we need a training corpus [Manning, Schutze '99]. The initial tags assigned are transformed based on the specified rules and hence this tagging is taking into account previous words and tags not just tags and is more contextual.
Consider an example "I like a run in the morning." According to the dictionary run is a verb but when followed by a determiner it can be considered as a noun this is the transformational rule where the source tag is "VB" and the target tag or to be transformed to tag is "NN" when the previous word is "a".

There are other methods of tagging using decision trees [Schmid 1994], neural networks [Benello et al. 1989, k nearest neighbour [Ratnaparkhi 1996]. In this survey we will implement the Markov technique this is viable since the mathematical text has fewer

special cases in terms of language use and the vocabulary is restricted and number of nouns are greater than number of verbs used in the text.

3.0 Methodology

The two main components of this thesis need different methodologies. First dimension is the research methodology for survey portion of the thesis and the second dimension is the software development methodology for developing KSM framework. We use "Algorithm for Discovery" [Paydarfar and Schwartz 2001, Algorithm for Discovery] as our guideline and methodology for survey portion of the thesis. This appeared as an editorial in the Science Magazine. We use Agile methodology to develop software for the KSM framework. In particular we use Extreme Programming [Beck 1999], which is one of the Agile methodologies in use in the industry for software development.

3.1 Algorithm for Discovery

We have followed the method of discovery laid out by Paydarfar and Schwartz [Paydarfar and Schwartz 2001, Algorithm for Discovery] during the survey portion of the thesis. This method presents a strategy learnt from prior discoveries and gives us an all round balanced approach to solving a problem(s). There are five main principles of "Algorithm for Discovery" given by Paydarfar and Schwartz. In the paragraphs below we go over each of them.

1. **Slow down to explore:** We use this method of "Algorithm for Discovery" to explore software and theories that we need to accumulate to create the KSM framework. One of the main aspects of this approach by Paydarfar and Schwartz is to slow down while exploring new areas and not jump to conclusions, till we explore and question anomalies in the solution. We have encountered anomalies in the various tools like Stanford parser which uses PCFG and Dependency grammars. These grammars have their own limitations and use Penn-Treebank which has limited vocabulary and n-gram patterns. We have also encountered tools like COQ which have limitations inherited from inconsistency in Mathematical axiomatic systems. A proof of this is mentioned in the Appendix B.

2. **Read, but not too much:** There is a lot of published material created by researchers on the topics of natural language processing, linguistics, grammars and algorithms etc and this poses a problem of managing the information and focusing on the material which are important to us. Following this "Algorithm of discovery" method we have been able to question published works by experts by testing the software generated by them. This has reduced expert influence in our solution since we are more bound by the results we get running a solution and not influenced only by the opinions of the experts.

3. **Pursue quality for its own sake:** The quality of the approach taken by the solution can be refined and leads to more discoveries. This is one of the important

aspects of “Algorithm for discovery”. Testing of the software available and seeing the inadequate results in certain scenarios opens the gate for further refinement of techniques used and options presented to the user.

4. **Look at the raw data:** The data in use to generate the results needs to be questioned and this is very important aspect of Algorithm for Discovery. Looking at particular scenarios for anomalies in data use generates good alternatives to be given to the user of the system. These alternatives may help user alter the result generated by the system to his satisfaction. This is a good guidance feedback loop which helps in user using the system as a tool. We provide such feedback at syntactical parsing level by capturing 4-best parse trees for a single sentence. This allows user’s to choose the appropriate parses when working with the KSM framework.

5. **Cultivate smart friends:** Discussions with fellow researchers and professors bring out different view-points which have further more discoveries and also set a direction to the next step in the search for a solution. This is the last principle of the “Algorithm of Discovery”

3.2 Extreme Programming

We have decided to follow the Extreme Programming (XP) approach during software development of KSM framework, which is one kind of Agile Methodology. Agile set of methods is defined by following principles: [see 5]

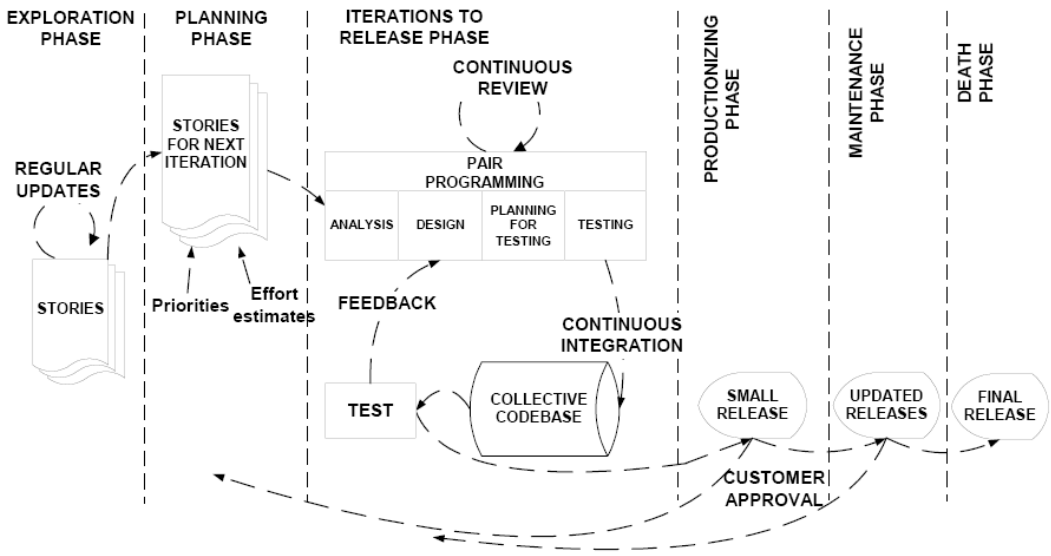
a) Customer satisfaction by rapid, continuous delivery of useful software.

b) Working software is delivered frequently.

c) Working software is the principle measure of progress.

d) Even late changes I requirements are welcome.

Here is the life Cycle of an XP process:



[Figure 6 from Abrahamssom 2002]

We decided to use this methodology since it was very suitable for iterative development and provides for refactoring of artifacts. Since the natural language problem is well segmented it made sense to divide this into various separate sub problems which could be solved and resolved by exploring various stages and domain of problems iteratively.

There are four main phases that we went through during the iterations:

a) Exploration phase

b) Planning phase

c) Iterations to release

d) Productionizing phase

a) Exploration phase

In this phase, story cards are written by the person requesting the software. These cards form the basis of feature specification, during this the solution providers to get up to speed with the tools and domain area of the problem. Prototyping of techniques takes place as part of learning the tools and the domain of the problem. We created a Software requirements specifications or SRS at various stages of software development that was done during exploration of the software project boundaries with the customer. This proved useful in scoping the requirements for the software portion of the thesis.

b) Planning phase

During this phase each feature set is ordered according to its priority. Estimate of effort is done in this phase. A schedule is developed for the delivery of the feature sets.

c) Iterations to release

This phase comprises of several iterations and selected feature sets. The first iteration creates a first draft of architecture of the whole system. Each iteration comprises of feature sets selected for a particular iteration. In the first iteration it is important to include feature sets which will shape the structure of the whole system. The customer runs tests at end of each the iterations. At the end of last iteration the implementation is ready to go to the production phase. It is important to note that at the end of each of the iterations we would get a working system with certain feature sets.

d) Productionizing phase

During this phase integrated testing as well as performance testing is completed. All the future extensions to the current system are documented in this phase. After this phase the system is ready to be delivered.

3.2 Stanford Parser

The Stanford parser consists of many classes as depicted in Appendix C. The processing done by the Stanford parser is shared among various algorithms. Stanford parser gets three categories of input information about the Treebank namely the Lexicon of the Penn-Treebank, the binary grammar and unary grammar inferred from the Penn-Treebank. The unary and binary grammars are inferred from Penn-Treebank, these form the PCFG rules on which Stanford parser operates as it sees input tokens.

The parsing function operates on the lexicon, binary grammar, and unary grammar. The parsing function initializes the P(word|tag) scores which assign the initial probability for parts of speech tags to each word. These words will be further assembled into groups as the parser tries to calculate inside and outside probabilities for each group of words. These groups of words are related to the PCFG rules and so are linked to the tree structures. There can be multiple combinations of these groups of words which are related to the PCFG rules. In order to find the best parse all the group of words combination are assigned inside and outside probabilities and the overall probability of each such combination is calculated using the combination of Inside-outside Algorithm [Manning 1999].

The PCFG parser mentioned above gives the phrase and part of speech information for a given sentence. The dependency parser gives the lexicalized dependency based on semantic categories like subject, modifier, direct object, indirect object. Stanford parser

uses unlexicalized PCFG and this classification between lexicalized PCFG and un-lexicalized PCFG is demarcated by the fact that there is no usage made of lexical class or content words to provide lexical or bi-lexical probabilities [Manning, 2003].

In this iteration of the KSM project we used the Stanford parser as a tool to generate tagging for sentences in a Mathematical Text (Basic abstract Algebra by Robert Ash).

3.3 Python NLTK

Python is an object-oriented language. It has the natural language toolkit which is open source and contains basic natural language manipulation functions and various corpora.

These functions include functions for tokenization, part-of-speech tagging, grammar extraction algorithms, machine translation and various algorithms for top-down and bottom-up parsing. The corpora it contains are from various subjects. Brown corpus which contains one million words of American English texts. Gutenberg corpus is another big collection of English text. Here is a complete list of various corpora.

Name	**Description**
Abcu	Australian Broadcasting Commission 2006
BioCreAtIvE-PPI	BioCreAtIvE Protein-Protein Interaction Corpus
Brown	Brown Corpus
chat80	Chat-80 Database
Cmudict	Carnegie Mellon Pronouncing Dictionary
conll2000	CoNLL 2000 Chunking Corpus
conll2002	CoNLL 2002 NER Corpus
Genesis	Genesis Corpus
Gutenberg	Project Gutenberg Selections

Ieer	NIST 1999 Information Extraction
Inaugural	US Presidential Inaugural Address Corpus
Indian	Indian Language POS-Tagged Corpus
Kimmo	
Names	Names Corpus
Paradigms	Paradigm Corpus
Pil	
Ppattach	PP Attachment Corpus
problem_reports	
Senseval	SENSEVAL 2 Corpus
Shakespeare	Shakespeare XML Corpus Sample
sinica_treebank	Sinica Treebank Corpus Sample
state_union	US Presidential State of the Union Address Corpus
stopwords	Stopwords Corpus
switchboard	
Timit	TIMIT Corpus Sample
toolbox	Toolbox Data Samples
Penn Treebank	Penn Treebank Corpus Sample
Udhr	Universal Declaration of Human Rights Corpus
Web	
wordnet	Wordnet 3.0
words	Wordlist (English)

The algorithms present in the NLTK have various functions in natural language evaluation. The functions in the NLTK have functionality to perform tokenization, generate grammars, parse sentences using the grammars.

The tokenizer in the toolkit can do tokenization specific to Penn-Treebank which expects some encoding for special symbols like brackets. This tokenizer library can also work on S-Expressions which were first introduced in Lisp. These S-Expressions are can represent a context free grammar tree as depicted below:

(((S) (NP) (VP)) ((VP) (V))) depicts the two productions in a context free grammar.

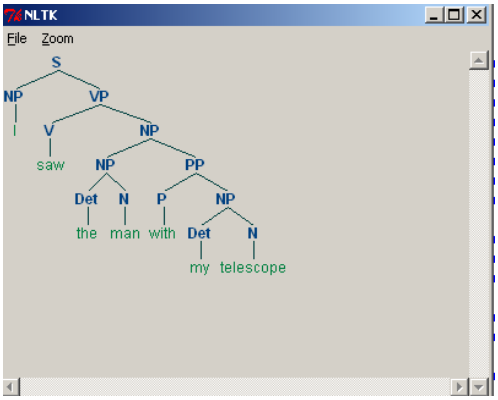
These expressions are recognized in Python tokenizer algorithm in NLTK. The tokenization is also possible in the NLTK using regular expressions to classify a token.

The tagging of words to the part of speech and phrase classification is available in the NLTK. The tnt.py uses statistical tagging to tag the words in sentences in brown corpus. The algorithm trains on one set of sentences in Brown corpus and then tests other sentences. There are taggers also based on N-grams (NgramTagger) which are available to classify the POS of sentences based on words around the target word. Apart from the tagger we have models like hidden Markov model algorithms which map the words to parts of speech tags based on probability. There are also transformational grammar tools

like Brill transformation which assigns tags to the words initially and in the second pass applies transformations given to it by the user, one example of the transformation is NNS -> NN if the text of the preceding word is ‘one’.

Algorithms for stemming using Wordnet corpus, algorithms to look up Wordnet to determine synonyms can be used for spelling corrections, antonyms and other such operations. Regular expressions can also be used for the purpose of stemming there is an implementation of executing this function.

Parsing algorithms which are numerous in the toolkit include Viterbi parser which is used to extract PCFG grammar from corpuses. The Viterbi parser assigns the grammar production rules probability values. The grammar production rules need to be input in the Viterbi algorithm. Here is a sample output from a parse of a sample sentence in this scenario there is a tree output already coded in this algorithm (Viterbi.py).



[Figure 7]

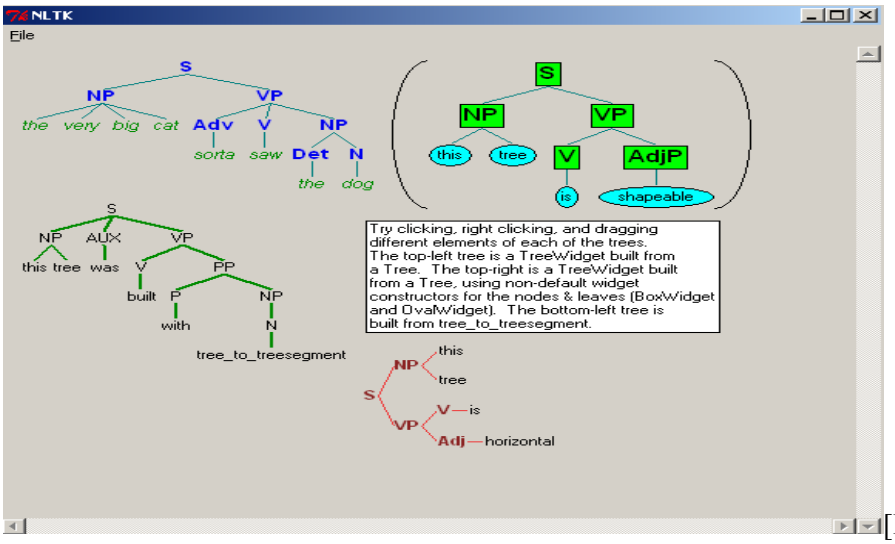
Implementations of shift-reduce and recursive descent parsers are also present in this NLTK. Like Viterbi parser which assigns probabilities to grammar production rules,

another parser which does the same thing is the chart parser. Earley chart parser also incorporates features of nodes while doing the parsing. An example of the result is

```
S[]
  (NP[NUM='sg'] (PropN[NUM='sg'] Kim))
  (VP[NUM='sg', TENSE='pres']
    (TV[NUM='sg', TENSE='pres'] likes)
    (NP[NUM='pl'] (N[NUM='pl'] children))))
```

Here the feature node contains the tense and the cardinality of a word. There are algorithms to calculate the entropy of sentences which is the negative log probability of a word occurrence given the tag occurrence. This is used in other algorithms of part of speech tagging like Viterbi and Earley chart algorithms. There are basic algorithm implementations of Edit distance and accuracy, precision and recall given test and reference lists in the NLTK.

There is implementation of first order theorem provers (tableau.py). There is also interfacing from Python to external theorem provers like “Prover9”. There are also tree drawing algorithms which present the information with different perspectives as in tree.py below:



[Figure 8]

In the user interface realm there is a CFG editor which has interfaces for changing production rules and running the grammar.

Python has many algorithms implemented to read any corpora. TaggedCorpusReader are some of the classes which read POS tagged corpus like Penn-Treebank corpus. The WordNetCorpusReader reads from wordnet and can lookup various synsets. The BracketParseCorpusReader reads corpuses with bracket separated parse trees. The PropbankCorpusReader reads the propbank corpus which adds information about predicate argument structure for every verb.

There are also algorithm implementations for clustering of data. Entity maximization Clustering, Group Average Agglomerative, KMeans clustering are the implementations of Clustering of data given the means, priors, covariance matrices for the vectors.

Implementations of classifiers are found in the NLTK and these are Naïve bayes, decision tree classifier, maximum entropy classifiers. There are interfaces to other machine learning libraries like weka, mallet and megam. Chunking algorithms are implemented using regular expression and there is also an implementation of Named entity parser (NEChunkParserTagger).

Finally there are complete applications with user interface implementations in the toolkit. These are simulators for recursive descent parser, shift-reduce parser, browser for wordnet corpora, and query engine for all corpora. Chunk and chart parser user interfaces for executing chunking or phrase structuring of a sentence.

4.0 Results

4.1 Overview of KSM Framework

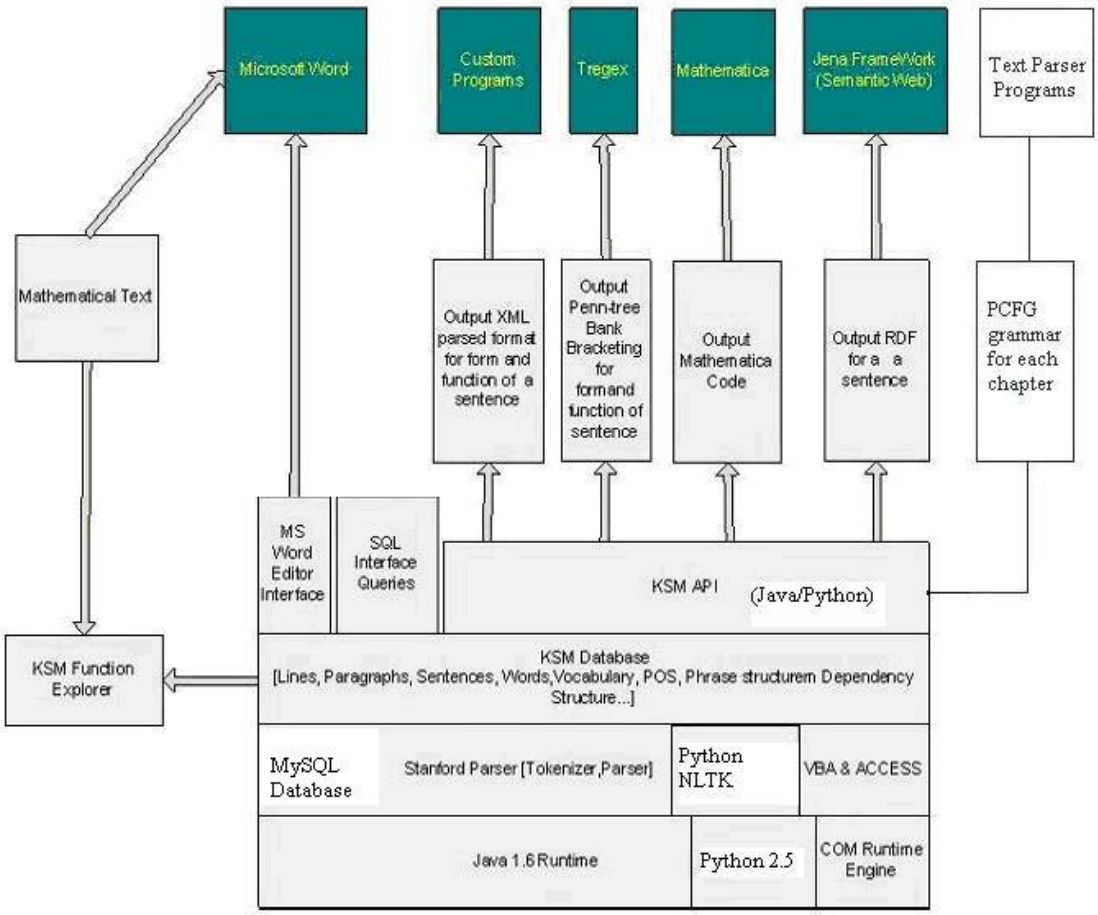
KSM framework implements form and functional layers for navigating through mathematical text. These layers are implemented using various programming languages. Exploration of the KSM framework is done at the form and functional layers. The addressability is provided at both these layers. The exploration of the segmented types like lines, sentence, words, paragraphs, sections, theorems, pages, chapters is provided at form and functional layers.

Interfacing to this framework of form and functional layers can be done via SQL, functional APIs as well as other programming languages and software. The SQL interface can be accessed via MYSQL query browser, JDBC/ODBC database drivers. The function set of KSM framework which is implemented in Java and Python can be used in programming languages. These programming languages can be any software runtime which can interface with Java and Python libraries. The function setin Java is also accessible from Word using the VBA programming language. The interfaces to other type safe systems like Jena can be done using the RDF representation. The Semantic-Web frameworks like Jena make use RDF representation to map to their object models (Subject-Object-Predicate).

Any mathematical text has physical layout(s). This layout is influenced by the author of the text as well as domain of the text being written. The database interfaces have exposed the physical layout of the text using form and functional layers. The form layer exposes the layout and can determine the location of the words, phrases etc. The form layer also determines the unique words, noun and verb phrases etc. The functional layer exposes the dependency structure of a sentence. The functional layer can also be used to cross-reference theorems and their sequence to determine ontology of the concepts covered in the text.

KSM framework uses Stanford PCFG parser to expose the natural language classes in a sentence by tagging “Parts of speech” and the “Phrase structure” of the sentence, it also tags the semantic aspect of a sentence by giving its dependency parse.

4.2 KSM Framework Architecture



[Figure 9]

4.3 Functions of the KSM Framework API

The functions mentioned here address the two problem domain areas i.e. “Creation and querying of the KSM database” (C) and “Parsing tasks of Mathematical text” (P).

Function 1: [Tokenizer.java] **(P)**

I/P: Text file of a Chapter of the book

ArrayList<Object> GetSentencelist (String ptfilename, Integer pirowno)

O/P: Sentence list of lines and sentences in the chapter

Function 2: [Tokenizer.java] **(C)**

I/P: A Physical line in the Text from the previous call or any line user wants to insert

InsertinLineTable (Hashtable<Integer, Object> lines)

O/P: Insertion of the line in the database table Book_line

Function 3: [Tokenizer.java] **(C)**

I/P: An English language sentence in the Text got from Function #1 call or any sentence user wants to insert.

InsertinSentenceTable (Hashtable<String, Object> sentencelist)

O/P: Insertion of the line in the database table Book_sentence.

Function 4: [CorporaParse.java, CallfromWord.java] **(P)**

I/P: Instance of the LexicalizedParser from Stanford and a sentence from Text

-ArrayList<String> ParseSentence (LexicalizedParser lp, String sentence)

O/P: Returns the bracketed form of the sentence which was provided as input

Function 5: [CorporaParse.java] **(P)**

I/P: Instance of the LexicalizedParser from Stanford and a sentence from Text

-ArrayList<String>OutputAllTrees(LexicalizedParser lp, String tsentence)

O/P: Returns multiple bracketed sentence this is basically various ways the sentence could have been parsed using PCFG parser which is using a trained Markov network.

Function 6: [CorporaParse.java] **(C)**

I/P: Assumes the sentence has already been created in the database using function 3. This function requires startrow, startcolumn, endrow, endcolumn as the location parameters to locate the sentence in the database and also supplies the syntax parse and dependency parse trees.

- SaveSentence (String[] tloc, String tParse, String DepParse)

O/P: Saves a Natural language evaluation of the sentence which includes "Parts of Speech", "Phrase structure" and "Dependency parse" information sets.

Function 7: [CorporaParse.java] **(P)**

I/P: Chapter number

-void SavePatternMatches (int ichap)

O/P: Saves the matches to the Tgrex pattern specified in the app.config file to the database table Book_pattern.

Function 8: [Datastore.java] **(C)**

I/P: The table name and the name, value pairs to be inserted in the table in the name fields of the table.

-InsertRowinDB(String tablename,ArrayList<ArrayList<String>> values)

O/P: Saves the given column values in the table.

Function 9: [Datastore.java] **(C)**

I/P: Query string to be executed on the database

-Recordset RunQuery(String tsql)

O/P: Returns the typed dataset after executing the SQL.

Function 10: [Utility.java] **(P)**

I/P: Penn-Treebank type bracketed sentence.

-String CreateMatrixfrombracketing(String t)

O/P: Tranforms the Penn-Treebank bracketing to XML.

Function 11: [Utility.java] **(P)**

I/P: Penn-Treebank type bracketed sentence.

-String CreateMathematicaCodefromBracketing(String tstr, Integer ipos)

O/P: Tranforms the Penn-Treebank bracketing to Mathematica code.

Function 12: [grammar.py] **(P)**

I/P: Read output of the Stanford parser for each chapter from a text file.

-def pcfg_demo

O/P: Generate the PCFG grammar on the screen output inferred from the text file.

4.4 SQL Tables and Types

SQL Tables:

Table 1:

Name: Book_Line

Purpose: Stores the physical form of the book.

Table 2:

Name: Book_Sentence

Purpose: Stores the sentence form of the book. This table also provides various syntax and semantic trees in bracketed form, apart from the sentence content.

Table 3:

Name: Book_tblparagraphmarkers

Purpose: Stores the paragraph boundaries in terms of physical line numbers.

Table 4:

Name: Book_Vocabulary

Purpose: Stores the unique words used in the book.

Table 5:

Name: Book_NP, Book_VP

Purpose: Stores the unique Noun, Verb phrases used in the book.

Table 6:

Name: Book_pattern

Purpose: Stores the chapter number, sentence text, Tgrex search string and matched output rows of the Tgrex search string.

Query1:

Name: Book_ADDParagraphnumbesrforaword

Purpose: Adds rows to Book_Paragraphmarkers. This is a construct to learn where a paragraph ends.

Query2:

Name: Book_ExtractUniqueVocabulary

Purpose: Adds rows to Book_Vocabulary table. This is a construct to learn all the unique words used in the book.

Query3,4,5:

Name(s): Book_TheoremBoundary, Book_CorollaryBoundary,

Book_PageBoundary

Purpose: These queries locate the physical rows in Book_line table where these

boundaries occur.

Query6,7:

Name(s): Book_otherMarkings, Book_Markingsaroundsentence

Purpose: These queries locate the physical rows in Book_line table where section

Headers for (Theorems, Lemma, Definitions etc) are located.

Query8:

Name(s): Book_querysimpleNps

Purpose: This query locates the noun phrases in all sentences with the word “ring” but this can be modified to locate collocation of a word in a noun phrase.

Query9:

Name(s): Book_IntheBookandinPT

Purpose: This query matches the lexicon which is common to the Penn-Treeank and the artifact processed by the KSM framework.

Query10:

Name(s): Book_Getsentencereferencesforword

Purpose: This query matches the word in the artifact text processed by KSM framework.

Query11:

Name(s): Book_ExtractTheoremStart

Purpose: This query matches the word “Theorem” if it is the starting word of the physical line for all lines of text in the artifact processed by KSM framework.

4.5 Integration with Microsoft Word

We want KSM framework to integrate with a widely used editor, so that a user can parse sentences while browsing mathematical text. This would help in the user having online help for reading complex sentences. Word provides a platform for entering and formatting sentences. It has services for looking up various types of information such as spelling, suggestions about grammar, etc. The service of parsing the form and function for any user selected sentence provides user with a tool to plough through dense information. This integration facility offered by the KSM framework is via the CallfromWord.java module. This involves Using VBA script which is resident in a word template which the user installs. **The VBA Functions:** RunParser, GetText, DeleteFile are used to communicate with the KSM framework which is implemented in Java. The user has a trigger key once the text is selected and this triggers the parser which parses the text and writes its output in a file. This file is read by the VBA function GetText.

4.6 Integration with the Semantic Web

RDF is a language which identifies resources on the World Wide Web. The KSM framework needs to be open to interpretation of tools on the Semantic Web. The resources such as mathematical text are physical and conceptual. The physical resources can be identified using RDF by the context markers that we have in KSM such as Chapter name/Theorem/Lemma/Definition, etc. The frameworks, such as Jena can accept and manipulate RDF formats and has a query engine to query the data. This query engine is called reasoner. It can be queried for various properties. In a similar way a relational query would query data, these RDF queries would query the types, e.g. definition of Euler's theorem for its sub types such as noun phrases used in this type. This linking creates knowledge to be linked and queried on the web with other RDF pages.

Schemas in RDF can be of the following format:

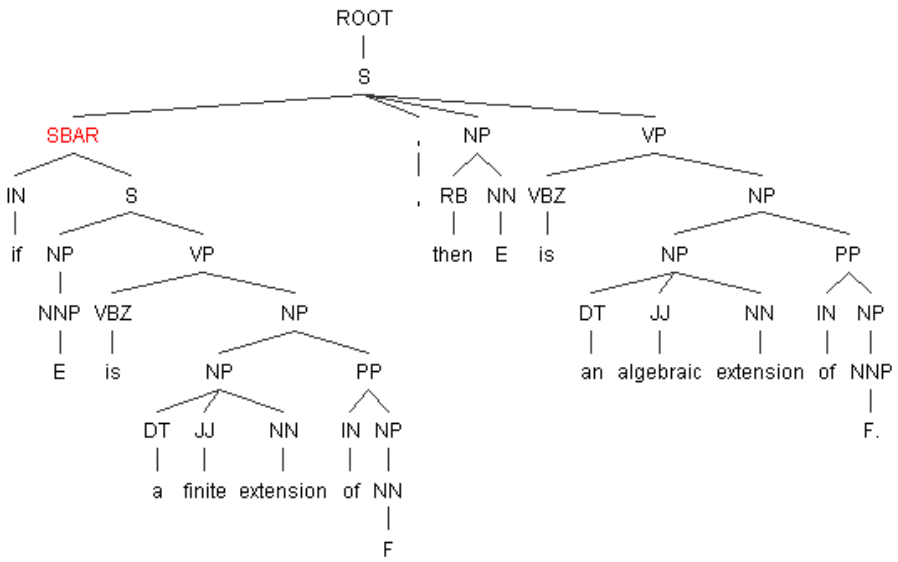
```
<rdf:Description rdf:about="&eg;Theorem">
  <rdfs:range  rdf:resource="&eg;NP"/>
  <rdfs:domain rdf:resource="&eg;VP"/>
  <rdfs:range  rdf:resource="&eg;STAT"/>
</rdf:Description>
```

The above RDF schema for a theorem enables us to query information based on phrases instead of words. This adds context to the search and therefore limits the search area.

5.0 Conclusion

Our survey has exposed various layers of types, algorithms, grammars and Turing machines we need to create the KSM framework. The type of individuals in the KSM framework encountered were sentences, lines, words, paragraphs, sections (like theorems, definitions etc), pages, chapters, book at a physical level and parts of speech, phrase structure, and dependency structure at a linguistic level (syntax and semantic layers). There is also a layer or level (or dimension) exposed at the conceptual level, exposing the semantic level of understanding of the mathematical text. Here are the things KSM Framework addresses at a functional level: Extract unique vocabulary from mathematical text by providing function to save the words it tokenizes. The tokenization uses lexical analyzer in the Stanford parser. Another functionality of the KSM framework is to extract lexical and phrase fragments found in statements in mathematical text, this we do with the help of the Stanford PCFG parser. We also provide interface to search for patterns by providing a way to go over the input text using the function in the tree parser "Tregex" tool from Stanford, to parse through the tree parse of a sentence looking for a user specified pattern. This search helped us match patterns such as SBAR type linguistic clause pattern in the sentence we introduced in the introduction section “if E is a finite extension of F, then E is an algebraic extension of F.” The pattern match is highlighted

below in red. Patterns can be searched using the interface "FindandSavePattern" function of the KSM framework.



[Figure 10]

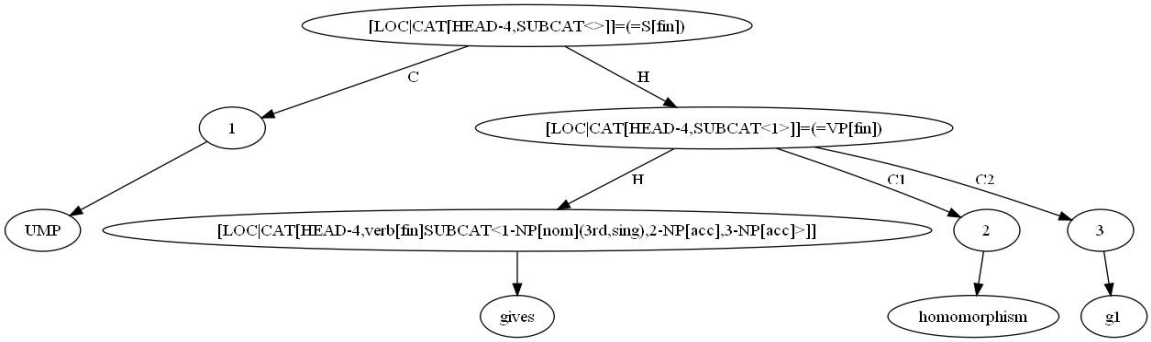
Patterns can be more precisely specified using the Tregex tool, here is an example of a use-case scenario we created during our survey, using the search pattern " (S < 1 SBAR & < 3 NP & < 4 VP)" which signifies that SBAR or clause type is the first child of a sentence and a noun phrase is the third child of a sentence and a verb phrase is the fourth child of a sentence. This pattern matching capabilities were exposed with the help of Stanford's Tregex tool is very useful when mining for patterns in mathematical text. For more information on Tregex patterns refer to http://nlp.stanford.edu [Roger Levy and Galen Andrew. 2006]. The dependencies between words are extracted using Dependency Grammar (DG) analysis, Stanford parser implements "Collins head search" algorithm

during the Dependency Grammar analysis. The KSM-framework parses the sentences in the given mathematical text and stores the noun phrases and verb phrases and stores them in database tables. The parsing of a sentence uses Stanford's PCFG parser. Since the PCFG parsing is probabilistic we extract the best 5 syntactical parses and store them in the database for the user to extract using SQL API. Apart from the syntactical parse the dependency parse results are also stored in the KSM-framework database. The positional data is stored in the sentence table this consists of information like start row and the number of lines in the sentence, the number of carriage returns and their locations in a sentence. This serves as the positional data which may be used by the future systems to extract positional dimension information from the parsed mathematical text.

Database storage of types helped us in specifying and using types as predicates. For example lines were stored as individual records and so a line type had the domain of “varchar" and its range was limited to a string of characters limited by the carriage return character. Similarly, sentences were tokenized using algorithms and these were stored in the database as records with the domain of “varchar” and range was revealed when the line->sentence implication was executed and it was restricted by encountering a period. This helped determine the range of the type “sentence”.

During the work on this thesis, the grammar theories such as HPSG, RRG show Object-Oriented behavior embedded in the definition of terms and principles e.g HPSG consists of types like PHON, SYNSEM where SYNSEM is further divided into CATEGORY, CONTENT and CONTEXT this relationship is similar to association ,the Head feature

principle states that " The HEAD value of any headed phrase is structure-shared with the HEAD value of the HEAD daughter " this as depicted in the diagram below:

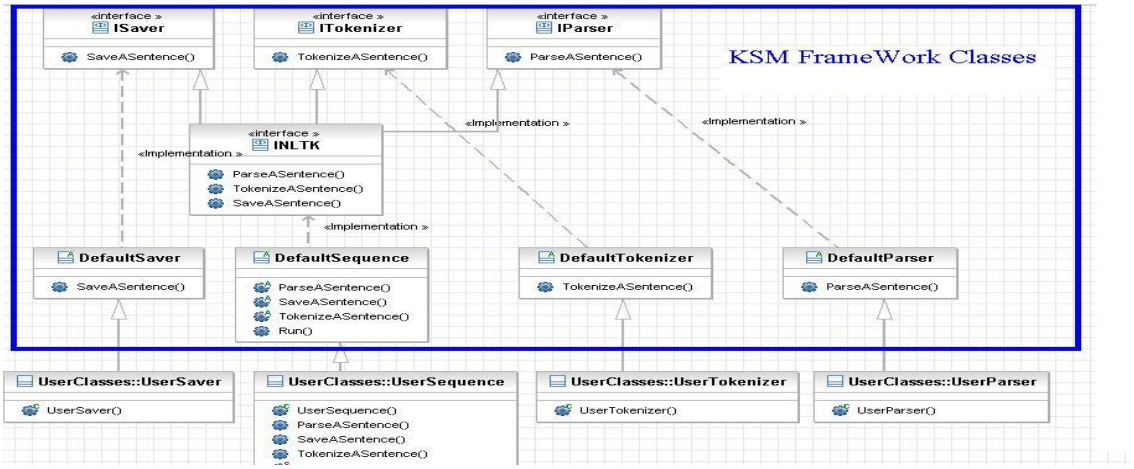


[Figure 11]

This Head feature principle reuses the Head structure and is a example of encapsulation of the Head structure. These are some of the examples of Object-Oriented behavior we saw in the grammatical theories.

Implementation of KSM framework was done primarily using a functional oriented methodology which has discrete classes like CorporaParse, Databasestore, Tokenizer, Utility, these classes are encapsulating functions for parsing, database, tokenization, utility/representation for tagged sentence respectively . A tighter integration with the Stanford parser was evaluated using the Template pattern. We recommend the use of template pattern for the next version of KSM framework for interaction with the Stanford parser. Here is the working of the pattern illustrated using string symbols, "DT", "DP" and "DS" symbolize the default implementation or class instances and "UT", "UP" and

"US" symbolize the overloaded class instances. Here DT stands for "DefaultTokenizer", DP stands for "DefaultParser" and DS stands for "DefaultSaver" classes. Similarly the UT stands for "UserTokenizer", UP stands for "UserParser" and US stands for "UserSaver" classes.



[Figure 12]

UQ.main((DT.T)(DP.P)(DS.S))-> UQ.main((UT.T)(DP.P)(DS.S)) or

UQ.main((DT.T)(DP.P)(DS.S) ->UQ.main((UT.T)(UP.P)(DS.S)) or

UQ.main((DT.T)(DP.P)(DS.S) -> UQ.main((UT.T)(UP.P)(US.S)) [full NP-form].

Over the course of the thesis the Stanford parser and Python's Natural Language Toolkit are considered as Turing machine repositories for the KSM framework on which the framework depends for various functions. MySQL Database serves as a repository of information and semantic frameworks like Jena can be used over the KSM framework to represent phrase level information on the Semantic web. The PCFG grammar generated by the KSM framework is one of the important outputs of the KSM framework, this helps us look at information in the Mathematical text in a BNF format.

The thesis has helped us appreciate linguistic, computer science and mathematical theories and tools currently in use. We have explored CFG, PCFG, Dependency grammar, HPSG, RRG grammars during this iteration of KSM framework. Future implementation of the KSM framework would use HPSG, RRG grammars so that we can explore the semantic dimension in more detail (lexical level). The interfacing to Coq syntactical engine would also be one more dimension to explore as an extension to the current KSM framework. A good Graphical visualization and interaction framework can serve as an important extension for future work on the KSM framework. This would provide the user more intuitive interfaces and also add ways to add supervised knowledge to KSM database.

I would like to thank Dr. Christopher Morgan for guiding me throughout this thesis, without his help this thesis would not be possible.

6.0 References

Allen, Bryant (1995). Unification-based Adaptive Parser. ACM

Ash (2007). Basic Abstract Algebra. Dover.

Bernardo (2000). Bayesian Theory. Wiley.

Booch (1994). Object-oriented analysis and design with applications. Redwood City, Calif: Benjamin/Cummings Pub. Co., 1994.

Brown and Guest (2002). Using Role and Reference grammar to support computer assisted assessment of free-text answers, The Higher Education Academy.

Chirstodaulakis (Ed.) (2000). Natural Language Processing. In Proceedings of 2nd International Conference, Patras Greece.

Choi, Son (1991). Unification in Unification based grammar. In proceedings of 6th

Japenese-Korean Conference on Formal Lingusitics.

Chomsky (1969). Aspects Of The Theory of Syntax. MIT Press.

Coad (1991). Object-oriented design. Englewood Cliffs, N.J: Yourdon, 1991.

Collins (1999). Head-driven Statistical Model for Natural Language Parsing. Phd. Thesis.

Collins (2003). Head-driven Statistical Models for Natural Language Parsing. ACM.

Coq (2008). Theorem-Proving Tool's Specification Language. http://coq.inria.fr/doc/main.html

Culicover (1997). An Introduction to Syntactic Theory. Oxford.

Dawson (2007). Guide to Python Programming. Course Technology.

Dean (2003). Chomsky A Beginner's Guide. Hodder and Stoughton.

Feller (1962). An Introduction to Probability Theory and its Applications Vol I. John Wiley and Sons.

Fulton (1998). Intersection Theory. New York: Springer, 1998.

Golfarb (1999). The Xml Handbook. Prentice Hall.

Grädel and Thomas, Wilke (Eds.) (1998). Automata Logics and Infinite Games. Springer.

Iwanska, Stuart (2000). Natural Language Processing and Knowledge Representation. AAAI/MIT Press.

Jurafsky (2008). Speech and Language Processing (2nd edition). Pearson Prentice Hall.

Kornai (2008). Mathematical Linguistics. Springer.

Krantz, Luce, Suppes and Tversky (2007). Foundations of Measurement Vol I. Dover

Lakoff (1990). Women, Fire And Dangerous Things. University of Chicago Press.

Lehnert and Martin (1982). Strategies for Natural Language Processing. Lawerence Erlbaum

Leithanser (1988). Exploring Natural Language Processing. WindCrest Books

Madnani (2007). "Getting started on natural language processing with Python." Crossroads 13.4 (2007): 5-5.

Manning and Schutze (1999). Foundations of Statistical Natural Language Processing. The MIT Press.

Megill (2007). Metamath, A Computer Language for Pure Mathematics. Lulu Press.

Miyao and Tsujii (2002). Maximum Entropy Estimation for Feature Forests**.** In Proceedings of HLT 2002.

Miyao and Tsujii (2003). Probabilistic modeling of argument structures including non-local dependencies.

In Proceedings of the Conference on Recent Advances in Natural Language Processing (RANLP) 2003, pp. 285-291

Miyao, Ninomiya, and Tsujii (2004). Corpus-oriented Grammar Development for Acquiring a Head-driven Phrase Structure Grammar from the Penn-Treebank

Miyao and Tsujii. 2005. Probabilistic Disambiguation Models for Wide- Coverage HPSG Parsing. In Proceedings of ACL-2005, pp. 83-90.

Münster Compiler (2007). The Münster Compiler. http://danae.uni-uenster.de/~lux/curry/

Neapolitan (2004). Learning Bayesian Networks. Pearson Prentice Hall.

Ninomiya, Matsuzaki, Tsuruoka, Miyao and Tsujii (2006). Extremely Lexicalized Models for Accurate and Fast HPSG Parsing. In Proceedings of EMNLP 2006 Ninomiya, Matsuzaki, Miyao, and Tsujii. 2007. A log-linear model with an n-gram reference distribution for accurate HPSG parsing. In Proceedings of IWPT 2007.

Norvig and Stuart (1995). Artifical Intelligence. Prentice Hall.

Pollard, Sag (1994). Head-Driven Phrase Structure Grammar. University of Chicago Press.

Quine (1964). Word and Object. MIT Press.

Levy and Andrew (2006). Tregex and Tsurgeon: tools for querying and manipulating tree data structures. 5th International Conference on Language Resources and Evaluation.

Rudin (1966). Real and Complex Analysis. McGraw-Hill.

Sag, Wasow and Bender (2003). Syntactic Theory 2nd edition. CSLI Publications.

Sandford (1980). Using sophisticated Models in Resolution Theorem Proving. Springer.

Schubert, Windley, Foss(Eds.) (1995). Higher Order Logic Theorem Proving and its Applications. In Proceedings of 8th International Workshop, Aspen UT.

Scott (2006). Programming language Pragmatics 2nd edition. Morgan Kaufmann.

Sheard (2005). Putting Curry-Howard to work. Proceedings of the 2005 ACM SIGPLAN workshop on Haskell

Simon (1991). Type theory and Functional Programming. Addison-Wesley 1991.

Sipser (2005). Introduction to the Theory of Computation, Second Edition. Boston Course Technology, 2005.

Stanford (2008). Head-Driven Phrase Structure Grammar. http://hpsg.stanford.edu

Studer (Ed.) (1989). Natural Language and Logic. In proceedings of the International Scientific Symposium, Hamburg.

Talmy (2003). Toward a Cognitive Semantics Vol I, II. MIT Press.

Theidorids, Koutroumbas (2006). Pattern Recognition. Academic Press.

University of Berlin (2008). The HPSG Bibliography. http://hpsg.fu-berlin.de/HPSG-Bib/hpsg_bib.html

Van Valin (2002). An Overview of Role and Reference Grammar.

APPENDIX A

Software Requirement Specification for KSM Framework

Purpose:

The main objective of the system is to help in understanding the multi-dimensional aspects of the information in a specialized Technical text. The dimensions used could be linguistic dimension such as part of speech or phrase information or vocabulary information. The dimensions could also be mathematical inferences derived from the knowledgebase about chapter, page, paragraph, sentences, Theorems. The system will have functionality which is extendable and callable from other software systems. The two main layers the system is based on are that of form and function. The form dimension deals with syntactical data and addressability of this data using types e.g. words, sentences, paragraphs, pages, chapters. The function dimension deals with semantic data and addressability of this data using types like phrase structure, dependency structure, grammar structure.

Scope:

The classification of information will help domain experts and students in harvesting the classified knowledge. The Object framework this system is built upon will be reusable and so can also be harvested in building other systems. This project is part of the “Software Proofs” portion of the thesis. This project will be done under the guidance of Dr. Christopher Morgan and will be part of the thesis fulfillment objective of the Master of Computer Science degree.The software system will provide a bird’s eye view on what level of understanding is possible by using the techniques and tools available in computer science, linguistics, mathematics to help technical text get to a another level of understanding. The proposed system will work on a text formatted copy of a technical book. It will help the user capture various dimensions of the written technical text. Vocabulary, parts of speech, phrase information are some of the linguistic dimensions which can be explored using the system. There is also physical layout dimensions like chapter, page, paragraph, sentences, theorems which can be explored using the system. The vocabulary information will contain unique words The physical layout dimension here refers to segments of textual layout boundaries like chapter, page, paragraph, theorem. There will be parsing of the linguistic dimension which will generate possible parse trees. These parse trees will contain information about parts of speech as well as phrase information. This version of software will provide a set of unique noun phrases. The system is focused on mining the captured information. The mining of vocabulary and physical layout information will be done using using relational queries. The system also provide other ways of mining by exporting the parse trees to tree mining systems like Grep2 and Tgrex. The system will be built on an Object Framework and the functions will be exposed via class Interfaces these interfaces can be called from other systems. This will permit exposing the interfaces of all the use cases to other software systems

Sr #	*Description*	*Intent*
1	Terms	Vocabularies of interest in a particular text that helps build layers of knowledge above corpora based English.
2	Context	Context can be defined as physical fragments of text or non-physical views of information in the text. The physical context are chapters, pages, paragraphs, sentences, words. The non-physical contexts are noun phrases.
3	Part of speech	Classification classes of words in terms of their use in a sentence. There are 8 major parts of speech (verb, noun, pronoun, adjective, adverb, preposition, conjunction, interjection)
4	Phrase Information	Classification of a group of words in terms of their function in a sentence there are 5 main phrase types (verb, noun, adjective, adverb, adjective, preposition phrase)
5	Dimension and Fact tables	The dimension table holds data with one key information e.g. Chapter table. The fact table will hold a multi-dimensional data like Noun phrase table.
6	Data mining queries	These will be relational queries based on stored contextual and vocabulary information. The information will be stored in star schema to expose multi-dimensional views of information. .

#	Name	Description
1	Term user	A “Term User” can use “Insertinline, InsertinSentence, InsertRowinDB” functions of the Knowledge base(KB) system.
2	Query user	A “Query user” can Query the Knowledge base(KB
3	Parse Tree user	A “Parse Tree” user is a user expecting to parse the information loaded by the term user.
4	Update user	A “Update user” is the user expecting to update the data in the KB. This user can use the Save*, Insert* functions.

#	Name	Description
1	Insertin [Line/Sentence/anyothertable]	Load the unprocessed information in database.
2	ParseSentence or OutputAlltrees [Sentence]	Parse using the natural language parser.
3	RunQuery(“Select * from Book_Vocabulary”)	The user or the a program can get the Vocabulary used in the book
4	RunQuery(BookFindANounphrasereference)	The user or the a program using this function can query this system for information regarding a Chapter e.g. what noun phrases were used, what is the count of a particular word or POS etc.
5	RunQuery(BookFindPage)	The user or the a program using this function can query this system for information regarding a Page e.g. what noun phrases were used, what is the count of a particular word or POS etc.
6	RunQuery(BookCountofWord)	The user or the a program using this function can query this system for information about count of a particular word or POS etc.
7	RunQuery(Book_theoremBoundary, Book_otherMarkings, Book_Markingsaroundsentence, Book_FindANounphrase)	The user or the a program using this function can query this system for information regarding a Theorem e.g. what noun phrases were used, what is the count of a particular word or POS etc.
8	ParseSentence	A Sentence can be parsed and the bracketed text of the parse will be returned.
9	FindAndSavepattern	These will execute internal existential inferences will be based on various lexical patterns provided by the user/caller e.g. “NP such as NP” or “NP subset of NP”
10	SaveSentence	Store in the dimension and fact tables in the knowledge base.

11	pcfg_demo()	Use Python to generate the grammar from a given chapter

APPENDIX B [Independent Study Results]

COQ Supplements

The Constructive Coq Repository at Nijmegen, C-CoRN, aims at building a computer based library of constructive mathematics, formalized in the theorem prover Coq. Coq has the language to generate an **axiomatic formalization of the most common algebraic structures**, including setoids, monoids, groups, rings, fields, ordered fields, rings of polynomials, real and complex numbers.

Coq was used as a reference system to study how the formalism is helping Coq parse out Abstract Algebra theorems. Coq is based on "Calculus of Inductive Constructions"

Gallina is the specification language used by Coq. It is a context free grammar. The following keyword exists in Gallina Set, if, Cofix, type, with, else.
Gallina has 3 types of specifiers Prop, Set and Type. Prop is used for Logical propositions, Set is used for specifying mathematical construction. Type is used to specify abstract types.

Here is a breakdown of a sentence in the Gallina grammar [http://coq.inria.fr/]:

Sentence ::= *Declaration*
| *definition*
| *inductive*
| *fixpoint*
| *statement [proof]*

statement_keywordd ::= *Theorem | Lemma | Definition*

statement ::= *statement_keyword ident [binderlet … binderlet]* : *term*

The sentence is derived in the Physical ontology in the text we considered into phrases. The mapping to Coq from phrases needs a classification into high level constructs like declaration, definition, inductive, statement. Declarations in the text are done using certain keywords like “Definition”. In Gallina the “declaration” is further broken down into Axiom, Conjecture, Parameter, Variable and Hypothesis. These in natural language text are phrases and need further refinement in order to extract the 5 kinds of declarations. The “statement“ in Gallina is mappable to

"Theorems" in the concept ontology mentioned earlier. The "term" in Gallina encapsulates logical constructs. Mapping of logical constructs from natural language to Gallina syntax would require marking the start and end of a self contained logical statement for example "if" could be a natural language statement we want to map to Gallina. This would require us to demarcate the start and end of the "if" statement. We would need to extract the variables in the if clause e.g. "if alpha and beta are roots of the irreducible polynomial f element of F[X] in an extension Eof F, then F(alpha) is isomorphic to F[beta] via an isomorphism that carries alpha into beta and is the identity of F". This would also involve declaring functions, initializing variables, terms in the Gallina syntax.

Term ::= `forall` *binderlist* `,` *term*

| `fun` *binderlist* `=>` *term*

| `fix` *fix_bodies*

| `cofix` *cofix_bodies*

| `let` *ident_with_params* `:=` *term* `in` *term*

| `let fix` *fix_body* `in` *term*

| `let cofix` *cofix_body* `in` *term*

| `let (` *[name , ... , name]* `)` *[dep_ret_type]* `:=` *term* `in` *term*

| `if` *term [dep_ret_type]* `then` *term* `else` *term*

Coq provides mathematical expressions syntax which can be resolved in the coq framework. For example in Coq we can write f:X->Y as f:Morphism ((X ==> Y). This is for instance an assumption in stating the Grothendieck-Riemann-Roch theorem which states that for a proper morphism F:X->Y of non-singular varieties,

Ch(f*alpha).td(Ty)-f*(ch(alpha).td(Tx))

Coq can help in processing the opaque formulae to natural language processing.

On the existence of Evil Types of things in COQ

(An essay by Dr. Christopher Morgan)

Coq is a program developed in France to model logic. It is a proof verification system that allows users to interactively develop and verify formal logical systems. A web site describes how it has been applied to many areas of mathematics and computer science, including set theory, algebra, and program verification. It uses type theory to resolve problems of self reference so that it can model inductive reasoning. It takes a constructive approach to logic, in that double negations of propositions are not necessarily equivalent to themselves, and that proof is the goal, rather than some absolute notion of truth. However, it does include True and False as entities of type 'Prop" which stands for a built-in type that models the notion of proposition.

In the following discussion I will describe the result of an experiment with this system that shows that one can introduce some plausible assumptions in a Coq session that will result in having the system verify such things as "forall P:Prop,P". This can be interpreted as: all propositions are provable. If one takes the approach that provable implies true, then the conclusion is everything is true. Indeed, within the same session, it is possible to verify statements such as "True=False" and "0-1".

The session begins with the introduction of an entity that behaves like False and can be introduced into a session in the same way as False is defined. I call this entity Evil:

```
Coq > Inductive Evil: Prop :=.
```

This is modeled after the definition of False that is already assumed upon normal startup. The Print command can be used to display entities that are currently in the system:

```
Coq > Print False.
Inductive False: Prop :=.
```

Let's not try to understand what this means or even how to interpret it. Let's just allow there to be two entities of type "Prop" that are defined in the same way. Coq attempts to model induction with the Inductive command. Induction is notoriously problematic in logic. However, it is fundamental to mathematics. For example, the axioms for natural numbers involve induction. Perhaps one can say it is a necessary evil.

Now that we have introduced "Evil", let us simply assume that there are entities of that type:

```
Coq > Axiom trouble:Evil.
```

The command introducing "Evil" has a side effect of creating several entities. Among them is "Evil_ind" that I now use it to verify a theorem that states "False". I start the theorem

```
Coq > Theorem th1: False.
```

The program responds:

```
1 subgoal

  ============================
   False
```

I respond:

```
th1 > apply Evil_ind; apply trouble.
```

The program responds with
```
Proof completed.
```

I complete the proof:

```
Coq > Qed.
```

The system responds favorably. This exercise serves as a model for how to verify any proposition in this system. For example.

```
Welcome to Coq 8.1pl2 (Oct. 2007)

Coq < Inductive Evil:Prop :=.
Evil is defined
Evil_rect is defined
Evil_ind is defined
Evil_rec is defined

Coq < Axiom trouble:Evil.
trouble is assumed

Coq < Theorem th2: forall P:Prop, P.
1 subgoal

  ============================
   forall P : Prop, P

th2 < apply Evil_ind.
1 subgoal
```

```
  ============================
   Evil

th2 < apply trouble.
Proof completed.

th2 < Qed.
apply Evil_ind.
apply trouble.
th2 is defined
```

Abstract Algebra Vocabulary

The process of classifying vocabulary in a text is a daunting one. Parts of speech have been used as starting points in linguistic theory. Here we have a text based on abstract algebra. We have python script lookup Brown corpora and assign the part of speech to the words in the text.

These words can help us take the position of the observer in a phrase to create rules which use them as terminal symbols, to help create the grammar existing around them In terms of creating a fixed "CAT", "CONTENT", "CONTEXT" tags using the HPSG formalism. Hope is that these will turn out to be much simpler AVMs to predict, and can be reused in different sentences without much change. The clarification of context can be done using the N-grams which predicts the 3 word combination occurrences uniquely across the Abstract Algebra text.

Here are the 18 words used in decreasing order:

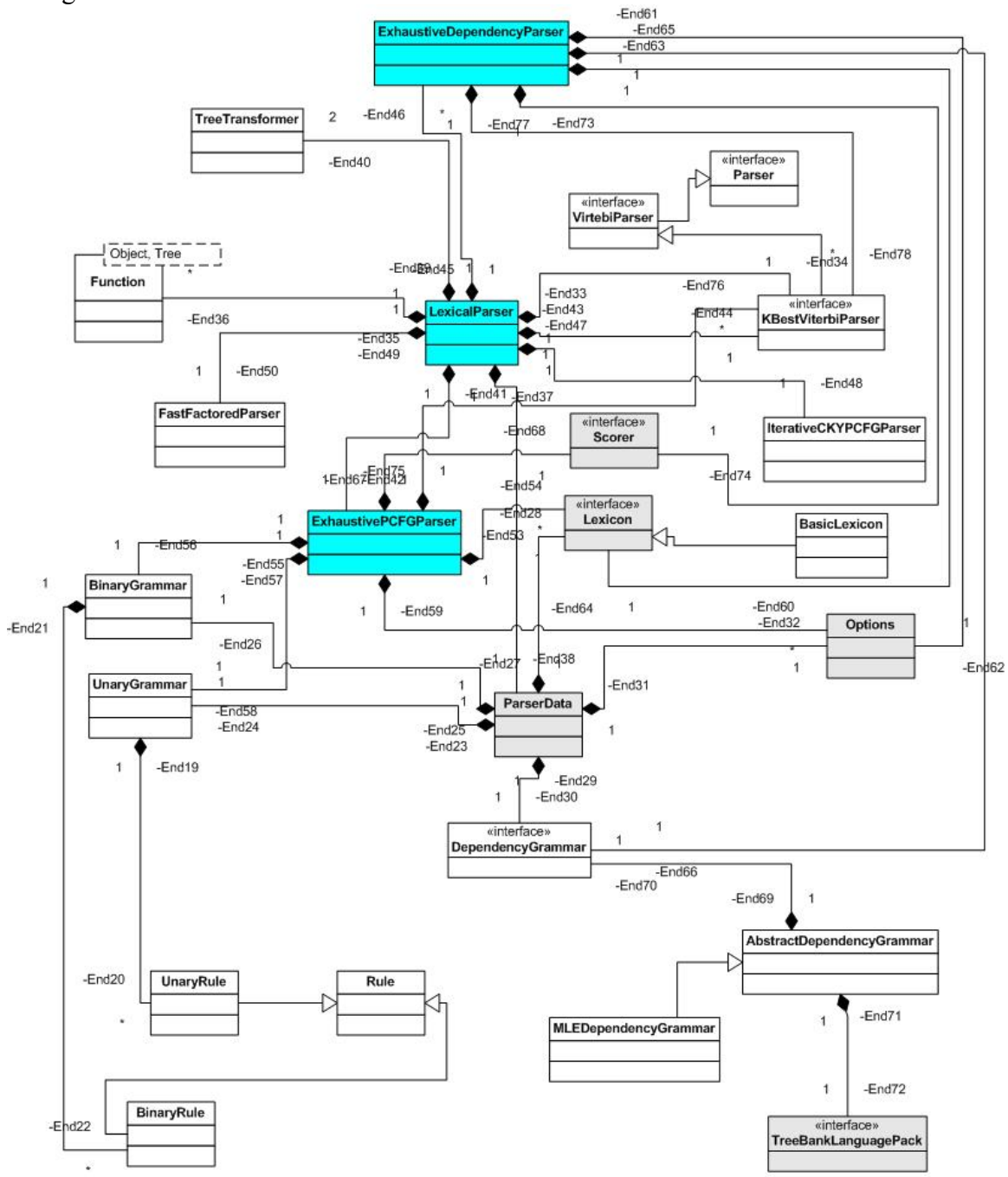
Is	4264	0.052031727
Of	4288	0.052324588
If	1268	0.015472849
Group	434	0.005295912
Ring	315	0.003843807
Field	309	0.003770592
On	309	0.003770592
Polynomial	232	0.002830995
Element	205	0.002501525
Subgroup	175	0.002135448
Theorem	170	0.002074436
Normal	159	0.001940207
Integral	146	0.001781574
Abelian	131	0.001598536
Basis	124	0.001513118
Homomorphism	119	0.001452105
Isomorphism	102	0.001244661
Free	101	0.001232459

In these frequent occurring words(freq >100) we have only 2 verbs "is" and "basis". There are 3 conjunctions "of", "if", "on", rest are all nouns. In the whole vocabulary there are around 2500 unique words.

Category for the above words is apparent. "CONTENT" for these words can be expressed according to the role (of a lexical head or non lexical head) of these in a phrase or sentence.

APPENDIX C [Stanford Parser Class Diagram]

This high level class diagram has been derived from the code of the Stanford parser. This lists the main classes responsible for processing the requests of the PCFG parser. These highlighted classes in blue are the focus of the functionality used in the Stanford parser during this thesis.



[Figure 13]

APPENDIX D [PCFG Grammar of Mathematical Text]

The following was the output generated using the python NLTK. We used the conversion functions in Python NLTK (grammar.py) to convert the parsed sentences of mathematical text to PCFG grammar. This involved converting the n-ary trees to Chomsky normal form and then calculating the probability of each production. The algorithm induce_pcfg (Python NLTK) calculates probability of each production of form A -> B C (Chomsky normal form) is calculated using ratio of instance count of A -> B C instances of sequences in the n-ary trees to the count of instances of form A->*. The n-ary trees were generated using the Stanford parser.
Grammar with 436 productions (start state = S)

JJ -> 'prime' [0.0512820512821]
LST -> LS [0.25]
VP -> VBP ADJP [0.0652173913043]
DT -> 'A' [0.0869565217391]
NN -> 'bc' [0.0153846153846]
MD -> 'might' [0.5]
NP -> NN [0.0855263157895]
VP -> VB SBAR+S [0.0217391304348]
NNS -> 'people' [0.0588235294118]
NP|<NP-,> -> NP NP|<,-CC> [0.2]
NNP -> 'Zm' [0.0454545454545]
NNP -> 'Fundamentals' [0.0454545454545]
S+VP|<PP-PP> -> PP PP [1.0]
CC -> 'or' [0.142857142857]
JJ -> 'commutative' [0.025641025641]
LST -> -LRB- LST|<DT--RRB-> [0.25]
NN -> 'C' [0.0153846153846]
NP|<CC-NP> -> CC NP [1.0]
IN -> 'If' [0.102564102564]
S -> NP S|<VP-.> [0.176470588235]
VP|<NP-ADVP> -> NP ADVP [1.0]
S -> VP : [0.0294117647059]
NNP -> 'Group' [0.0454545454545]
PRN -> -LRB- PRN|<NP--RRB-> [0.375]
JJ -> '1a' [0.025641025641]
JJ -> 'finite' [0.025641025641]
NN -> 'proof' [0.0153846153846]
MD -> 'can' [0.25]
NP -> DT NP|<CC-NN> [0.00657894736842]
DT -> 'a' [0.260869565217]
SBAR -> IN S [0.6]
VP -> VB VP [0.0217391304348]
NP -> NP NP|<CC-NP> [0.0197368421053]
TO -> 'to' [1.0]
DT -> 'The' [0.0434782608696]
NP|<,-PP> -> , NP|<PP-,> [0.5]
NP|<NNS--RRB-> -> NNS -RRB- [1.0]
NN -> 'section' [0.0153846153846]
NP -> SYM [0.0789473684211]
NP|<CONJP-NP> -> CONJP NP [1.0]
NP|<,-PP> -> , PP [0.5]

LST|<DT--RRB-> -> DT -RRB- [1.0]
WDT -> 'which' [0.5]
NP|<FW-FW> -> FW FW [1.0]
S+VP|<ADJP-SBAR> -> ADJP SBAR [1.0]
NP -> CD NN [0.00657894736842]
NN -> 'cde' [0.0153846153846]
PRN -> -LRB- PRN|<FRAG+NP--RRB-> [0.125]
VP|<ADVP-VP> -> ADVP VP [1.0]
NP -> NNP NNS [0.00657894736842]
S+VP -> SYM NP [0.166666666667]
IN -> 'if' [0.025641025641]
RBR -> 'less' [1.0]
NP|<JJ-NNP> -> JJ NP|<NNP-NN> [1.0]
PDT -> 'all' [0.75]
DT -> 'this' [0.0434782608696]
JJ -> 'abelian' [0.025641025641]
S -> ADVP S|<,-NP> [0.0294117647059]
NNS -> 'numers' [0.0588235294118]
NN -> 'Z' [0.0153846153846]
NN -> 'bcd' [0.0153846153846]
NNS -> 'properties' [0.0588235294118]
NP -> NP NP|<,-NP> [0.0263157894737]
S+VP|<S+VP-PP> -> S+VP PP [1.0]
VP -> VB VP|<NP-ADVP> [0.0217391304348]
JJ -> 'other' [0.025641025641]
NNS -> 'conditions' [0.0588235294118]
NP|<DT-JJ> -> DT NP|<JJ-NN> [1.0]
NP|<PP-,> -> PP , [1.0]
WHNP -> NP PP [0.333333333333]
IN -> 'in' [0.205128205128]
NP|<NNP-NN> -> NNP NP|<NN-NN> [1.0]
SYM -> 'b' [0.368421052632]
VP -> VP VP|<CC-VP> [0.0217391304348]
NP -> LST NP|<DT-JJS> [0.00657894736842]
NP -> NNP NP|<NNP-NNP> [0.00657894736842]
FRAG+NP|<FW-NP> -> FW NP [1.0]
NNP -> 'Inverse' [0.0454545454545]
NP -> JJ DT [0.00657894736842]
NP|<ADJP-NNS> -> ADJP NNS [1.0]
NP|<NP-,> -> NP NP|<,-X> [0.2]
JJ -> 'different' [0.025641025641]
NP|<,-NP> -> , NP|<NP-,> [0.571428571429]
LST -> -LRB- LST|<NN--RRB-> [0.25]
VP -> VB S+VP [0.0217391304348]
PP -> X PP|<IN-NP> [0.030303030303]
NP -> DT NP|<ADJP-NN> [0.00657894736842]
S+VP|<S-PP> -> S S+VP|<PP-PP> [1.0]
VP|<NP-PRN> -> NP PRN [1.0]
NP -> NNS NP|<NN-NN> [0.00657894736842]
S|<,-NP> -> , S|<NP-VP> [1.0]
FW -> 'a' [0.5]
JJ -> 'cyclic' [0.025641025641]
NP -> DT NP|<JJ-NN> [0.0789473684211]
NNS -> 'generalizes' [0.0588235294118]
VB -> 'be' [0.25]
NN -> 'B' [0.0307692307692]

JJ -> 'rationals' [0.025641025641]
NP -> JJ NNS [0.0197368421053]
NP -> PDT NP|<DT-NN> [0.00657894736842]
NP|<JJ-NNS> -> JJ NP|<NNS-NN> [0.666666666667]
NN -> 'n.' [0.0153846153846]
S -> S S|<,-CC> [0.0294117647059]
S|<,-S+VP> -> , S+VP [1.0]
: -> ':' [0.555555555556]
NN -> 'rs' [0.0153846153846]
VP -> SYM NP [0.0652173913043]
JJ -> 'following' [0.0512820512821]
FRAG+NP -> -LRB- FRAG+NP|<NP--RRB-> [1.0]
NNP -> 'Closure' [0.0454545454545]
S|<,-ADVP> -> , S|<ADVP-NP> [1.0]
VP -> VBZ PP [0.0217391304348]
VP -> MD VP [0.0869565217391]
S|<CC-S> -> CC S|<S-.> [1.0]
NP -> DT NP|<JJ-NNP> [0.00657894736842]
VBN -> 'ab' [0.111111111111]
S -> LST S|<VP-.> [0.0294117647059]
NP|<DT-JJS> -> DT NP|<JJS-NN> [1.0]
NNS -> 'numbers' [0.0588235294118]
VBG -> 'satisfying' [0.333333333333]
JJ -> 'binary' [0.0769230769231]
NP|<:-NP> -> : NP [0.75]
NN -> 'group' [0.0769230769231]
JJ -> 'real' [0.025641025641]
NN -> 'c' [0.0153846153846]
JJ -> 'familiar' [0.025641025641]
NN -> 'identity' [0.0153846153846]
SBAR -> WHNP S [0.2]
NP|<NP-,> -> NP NP|<,-NP> [0.6]
NP|<NNP-NNP> -> NNP NNP [1.0]
NP -> NN NNS [0.00657894736842]
RB -> 'additively' [0.0833333333333]
ADJP -> JJ PP [0.111111111111]
SYM -> '=' [0.210526315789]
NP|<VBG-NNS> -> VBG NNS [1.0]
VB -> '=' [0.125]
DT -> 'that' [0.0217391304348]
S|<:-S> -> : S|<S-.> [1.0]
VBZ -> 'has' [0.0714285714286]
JJ -> 'inverse' [0.025641025641]
VP -> JJ VP|<NP-PP> [0.0217391304348]
NP -> NP NP|<PP-SBAR+S> [0.00657894736842]
S|<NP-VP> -> NP VP [0.777777777778]
NP -> NP VP [0.0131578947368]
SBAR+S -> NP VP [1.0]
NNS -> 'groups' [0.117647058824]
S -> S S|<,-NP> [0.0294117647059]
NP -> NP NP|<``-S> [0.00657894736842]
NN -> 'G' [0.0153846153846]
EX -> 'There' [0.5]
DT -> 'the' [0.347826086957]
NP|<PP-SBAR+S> -> PP SBAR+S [1.0]
VP|<NP-PP> -> NP PP [1.0]

NNP -> 'Nonabelian' [0.0454545454545]
NN -> 'order' [0.0307692307692]
-RRB- -> '-RRB-' [1.0]
CD -> '0' [0.0833333333333]
NP|<,-SBAR> -> , NP|<SBAR-,> [1.0]
S -> LST S|<NP-VP> [0.0588235294118]
NP|<NN-CD> -> NN CD [1.0]
JJ -> 'next' [0.025641025641]
PRN|<FRAG+NP--RRB-> -> FRAG+NP -RRB- [1.0]
PDT -> 'such' [0.25]
JJ -> 'Gis' [0.025641025641]
NN -> 'm.' [0.0153846153846]
JJ -> 'complex' [0.025641025641]
S|<,-CC> -> , S|<CC-S> [1.0]
NP -> DT NP|<NN-NN> [0.00657894736842]
VBN -> 'set' [0.222222222222]
S -> PP S|<NP-VP> [0.0294117647059]
S|<FW-,> -> FW S|<,-S+VP> [1.0]
NP|<VBN-NN> -> VBN NN [1.0]
S -> S S|<:-S> [0.0294117647059]
NN -> 'form' [0.0153846153846]
WHPP -> IN WHNP [1.0]
NP -> RB NN [0.00657894736842]
PRN -> -LRB- PRN|<VP--RRB-> [0.25]
NP|<JJS-NN> -> JJS NP|<NN-PRN> [1.0]
NN -> 'Q' [0.0153846153846]
X -> SYM [1.0]
NP|<``-S> -> `` NP|<S-,> [1.0]
JJ -> 'a-1' [0.025641025641]
S|<S-.> -> S . [1.0]
PRN|<NP--RRB-> -> NP -RRB- [1.0]
VP -> VBN VP|<NP-S+VP> [0.0217391304348]
PP|<IN-NP> -> IN NP [1.0]
NNP -> 'Associativity' [0.0454545454545]
NNP -> 'Subgroups' [0.0454545454545]
ROOT -> FRAG [0.142857142857]
NP -> PRP [0.00657894736842]
NNS -> 'elements' [0.0588235294118]
IN -> 'as' [0.025641025641]
NNS -> 'Groups' [0.0588235294118]
S+VP -> TO VP [0.166666666667]
VB -> 'begin' [0.125]
NP -> NP PRN [0.0263157894737]
VBP -> 'have' [0.142857142857]
PRP -> 'We' [1.0]
NP -> EX [0.0263157894737]
FRAG+NP|<NP--RRB-> -> NP FRAG+NP|<-RRB--FW> [1.0]
VB -> 'look' [0.125]
NP|<JJ-NNS> -> JJ NNS [0.333333333333]
RB -> 'then' [0.166666666667]
VBZ -> 'is' [0.785714285714]
RB -> 'namely' [0.0833333333333]
IN -> 'by' [0.102564102564]
S -> NP VP [0.382352941176]
NP|<NN-NN> -> NN NN [0.8]
RB -> 'often' [0.0833333333333]

VP -> JJ S [0.0217391304348]
NN -> 'number' [0.0307692307692]
NN -> 'law' [0.0153846153846]
VB -> 'appear' [0.125]
NP -> NNP [0.0855263157895]
WHNP -> WDT [0.666666666667]
NP -> DT NP|<JJ-NNS> [0.0197368421053]
NP -> CD [0.0197368421053]
: -> ';' [0.444444444444]
S+VP -> VBN S+VP|<S+VP-PP> [0.166666666667]
NP|<,-CC> -> , NP|<CC-NP> [1.0]
VP|<ADVP-PRN> -> ADVP PRN [1.0]
LST|<LS--RRB-> -> LS -RRB- [1.0]
VBN -> 'performed' [0.111111111111]
SBAR|<IN-S> -> IN S [1.0]
NN -> 'Un' [0.0153846153846]
JJ -> '1mod' [0.025641025641]
NP|<NNS-NN> -> NNS NN [1.0]
NP|<S-,> -> S NP|<,-SBAR> [1.0]
ADVP -> RB [0.833333333333]
NP|<JJ-NN> -> JJ NP|<NN-NN> [0.0625]
NP -> NP NP [0.0263157894737]
LS -> 'r' [0.5]
NN -> 'n' [0.107692307692]
IN -> 'for' [0.102564102564]
JJ -> '=' [0.0769230769231]
VBN -> 'given' [0.111111111111]
IN -> 'on' [0.025641025641]
CC -> 'and' [0.857142857143]
NP -> NNP JJ [0.00657894736842]
JJ -> 'associative' [0.025641025641]
PP -> TO NP [0.121212121212]
NP -> JJ NP|<JJ-NN> [0.00657894736842]
RB -> 'nonempty' [0.0833333333333]
ADJP -> JJ SBAR [0.111111111111]
DT -> 'any' [0.0217391304348]
VP|<CC-VP> -> CC VP [1.0]
PP -> IN NP [0.848484848485]
NP -> DT NP|<-LRB--NN> [0.0131578947368]
NP|<JJ-NN> -> JJ NN [0.75]
NP|<PRN-NN> -> PRN NN [1.0]
VBG -> 'following' [0.333333333333]
FRAG|<S-.> -> S . [1.0]
NN -> 'R' [0.0153846153846]
NN -> 'multiplication' [0.0461538461538]
S|<NP-VP> -> NP S|<VP-.> [0.222222222222]
LST -> -LRB- LST|<LS--RRB-> [0.25]
VP -> VBP NP [0.0652173913043]
ROOT -> S [0.857142857143]
NN -> 'element' [0.0307692307692]
JJ -> 'a1' [0.0512820512821]
ADJP -> RB ADJP|<JJ-PP> [0.111111111111]
NN -> 'abelian' [0.0153846153846]
JJ -> 'last' [0.0512820512821]
SBAR -> WHPP S [0.1]
NP|<JJ-NN> -> JJ NP|<NN-PRN> [0.0625]

NN -> 'case' [0.0153846153846]
PRN|<X--RRB-> -> X -RRB- [1.0]
NP -> -LRB- NP|<NNS--RRB-> [0.00657894736842]
VP -> VBN VP|<ADVP-PRN> [0.0217391304348]
NP|<NN-NNS> -> NN NNS [1.0]
CD -> '1' [0.333333333333]
-LRB- -> '-LRB-' [1.0]
S -> SBAR S|<,-NP> [0.0882352941176]
NP|<NN--RRB-> -> NN -RRB- [1.0]
NP|<QP--RRB-> -> QP -RRB- [1.0]
NN -> 'result' [0.0153846153846]
NP -> -LRB- NP|<JJ-NN> [0.00657894736842]
IN -> 'like' [0.025641025641]
NP -> PDT DT [0.0197368421053]
NP|<JJ-NN> -> JJ NP|<NN-NNS> [0.0625]
NP|<NN-NN> -> NN NP|<NN-NN> [0.2]
NN -> 'a-1a' [0.0153846153846]
JJ -> 'formal' [0.025641025641]
VP -> VB PP [0.0652173913043]
VP -> VBZ VP [0.0217391304348]
NNP -> '1.1.1' [0.0454545454545]
FRAG|<:-S> -> : FRAG|<S-.> [1.0]
NN -> 'a1' [0.0153846153846]
NP|<-LRB--NN> -> -LRB- NP|<NN--RRB-> [1.0]
LS -> 'n' [0.5]
JJS -> '=' [1.0]
NP|<SBAR-,> -> SBAR , [1.0]
NNP -> 'Identity' [0.0454545454545]
NP -> -LRB- NP|<QP--RRB-> [0.00657894736842]
NNS -> 'units' [0.0588235294118]
S|<ADVP-NP> -> ADVP S|<NP-VP> [1.0]
VBP -> 'belong' [0.142857142857]
NP|<NN-PRN> -> NN NP|<PRN-NN> [1.0]
NNP -> 'Definition' [0.0454545454545]
NP -> RB NP|<DT-JJ> [0.00657894736842]
RB -> 'also' [0.0833333333333]
VP -> VBZ ADJP [0.0434782608696]
VP -> VB NP [0.0217391304348]
ADVP -> RBR IN [0.166666666667]
RB -> 'rather' [0.0833333333333]
NNP -> 'Chapter' [0.0454545454545]
JJ -> 'aa-1' [0.025641025641]
CD -> '1.1.5' [0.0833333333333]
FW -> 'i.e.' [0.25]
NN -> 'unit' [0.0153846153846]
NP -> JJ NP|<DT-JJ> [0.00657894736842]
SBAR -> NP SBAR|<IN-S> [0.1]
JJ -> 'equivalent' [0.025641025641]
NNS -> 'ab' [0.0588235294118]
VBP -> 'ai' [0.142857142857]
NP -> CD DT [0.0131578947368]
NNS -> 'products' [0.0588235294118]
S|<,-FW> -> , S|<FW-,> [1.0]
VBN -> 'written' [0.222222222222]
VP|<NP-S+VP> -> NP S+VP [1.0]
S+VP -> VBG S+VP|<S-PP> [0.166666666667]

S -> SBAR S|<,-ADVP> [0.0294117647059]
RB -> 'very' [0.0833333333333]
IN -> 'that' [0.0512820512821]
EX -> 'there' [0.5]
`` -> 'G' [1.0]
S -> S+VP S|<:-S> [0.0294117647059]
NP|<CD-NNP> -> CD NP|<NNP-NNP> [1.0]
NP|<DT-NN> -> DT NN [1.0]
NN -> 'ab' [0.0461538461538]
VP -> VBP PP [0.0217391304348]
RB -> 'Furthermore' [0.0833333333333]
CD -> '+1' [0.166666666667]
NP|<JJ-NN> -> JJ NP|<NN--RRB-> [0.0625]
NP -> JJ NP|<FW-FW> [0.00657894736842]
NP -> DT NP|<VBN-NN> [0.00657894736842]
NP|<,-X> -> , X [1.0]
SYM -> 'c' [0.105263157895]
IN -> 'with' [0.025641025641]
CD -> 'two' [0.0833333333333]
ADJP -> RB JJ [0.222222222222]
NNS -> 'words' [0.0588235294118]
S+VP -> VBG NP [0.166666666667]
NN -> 'ba' [0.0153846153846]
NP -> NNS [0.0131578947368]
FRAG -> NP FRAG|<:-S> [1.0]
, -> ',' [1.0]
NP -> NP NP|<:-NP> [0.0263157894737]
LST|<NN--RRB-> -> NN -RRB- [1.0]
VBG -> 'G' [0.333333333333]
VBN -> 'aj' [0.111111111111]
SYM -> 'r' [0.105263157895]
S+VP -> VBZ S+VP|<ADJP-SBAR> [0.166666666667]
IN -> 'of' [0.179487179487]
ADJP -> RB VBN [0.111111111111]
VB -> 'equal' [0.125]
NN -> 'example' [0.0153846153846]
NP -> DT NP|<ADJP-NNS> [0.00657894736842]
NNP -> 'A' [0.0454545454545]
NNP -> 'G' [0.272727272727]
NP -> NP PP [0.0723684210526]
NP -> NP SBAR [0.0131578947368]
VP -> VBZ NP [0.0869565217391]
NN -> 'operation' [0.0461538461538]
CD -> 'ai' [0.0833333333333]
NP -> DT NP|<VBG-NNS> [0.00657894736842]
NNP -> 'a' [0.0454545454545]
PRN|<S--RRB-> -> S -RRB- [1.0]
PRN -> -LRB- PRN|<X--RRB-> [0.125]
DT -> 'some' [0.0217391304348]
FW -> '=' [0.25]
VBP -> 'are' [0.571428571429]
CONJP -> RB IN [1.0]
JJ -> 'positive' [0.025641025641]
NNS -> 'integers' [0.0588235294118]
VP -> VBN NP [0.0217391304348]
NN -> 'r' [0.0153846153846]

RB -> 'relatively' [0.166666666667]
VP -> VBZ VP|<ADVP-VP> [0.0217391304348]
NP -> JJ NN [0.00657894736842]
VP -> VBD PP [0.0217391304348]
CD -> 'a1' [0.0833333333333]
JJ -> 'such' [0.0512820512821]
NP -> CD NNS [0.0131578947368]
. -> '.' [1.0]
VP -> NN NP [0.0217391304348]
JJ -> 'integers' [0.0512820512821]
PRN|<VP--RRB-> -> VP -RRB- [1.0]
IN -> 'than' [0.0512820512821]
DT -> 'an' [0.152173913043]
NP -> DT NN [0.0657894736842]
IN -> 'In' [0.025641025641]
VP|<ADVP-PP> -> ADVP PP [1.0]
NP|<ADJP-SBAR> -> ADJP SBAR [1.0]
NP|<CC-NN> -> CC NN [1.0]
ADJP -> JJ [0.333333333333]
NN -> 'mod' [0.0461538461538]
VBZ -> 's' [0.0714285714286]
NN -> 'addition' [0.0153846153846]
NNP -> '+' [0.0454545454545]
NNS -> 'examples' [0.0588235294118]
VP -> VBZ VP|<NP-PP> [0.0217391304348]
CD -> '1.1' [0.0833333333333]
NNS -> 'ways' [0.0588235294118]
S|<VP-.> -> VP . [1.0]
NN -> 'Proposition' [0.0153846153846]
NN -> 'integer' [0.0153846153846]
NP|<,-NP> -> , NP [0.428571428571]
VB -> 'e.' [0.125]
S -> S+VP S|<,-FW> [0.0294117647059]
VBN -> 'defined' [0.111111111111]
NN -> 'induction' [0.0153846153846]
NP|<ADJP-NN> -> ADJP NN [1.0]
NN -> 'aj' [0.0153846153846]
VP -> VBZ VP|<NP-PRN> [0.0217391304348]
SYM -> 'a' [0.210526315789]
QP -> CD CD [1.0]
NNP -> 'G.' [0.0454545454545]
FRAG+NP|<-RRB--FW> -> -RRB- FRAG+NP|<FW-NP> [1.0]
VP -> VBN PP [0.0652173913043]
NP -> NP NP|<ADJP-SBAR> [0.00657894736842]
MD -> 'will' [0.25]
VBD -> 'ngenerated' [1.0]
NP -> DT NP|<NN-CD> [0.00657894736842]
VP -> VBZ VP|<ADVP-PP> [0.0217391304348]
NN -> 'modulo' [0.0153846153846]
WDT -> 'that' [0.5]
NP -> NP NP|<,-PP> [0.0131578947368]
VP -> VBZ [0.0217391304348]
NP -> NNP NP|<CD-NNP> [0.00657894736842]
NP|<:-NP> -> : NP|<NP-,> [0.25]
NP -> NP NP|<CONJP-NP> [0.00657894736842]
VBZ -> 'forms' [0.0714285714286]

IN -> 'under' [0.0512820512821]
ADJP|<JJ-PP> -> JJ PP [1.0]
PRN -> -LRB- PRN|<S--RRB-> [0.125]

APPENDIX E [Stanford Parser Output]

This output was generated using the PCFG parser provided by the Stanford Natural language processing group. This is a Java parser and has the classes mentioned in the class diagram inferred from the code in Appendix C.

```
(ROOT
 (S
  (NP
   (NP
    (NP (NNP Chapter) (CD 1) (NNP Group) (NNP Fundamentals))
    (NP (CD 1.1) (NNS Groups)))
   (CC and)
   (NP
    (NP (NNP Subgroups) (NNP 1.1.1) (NNP Definition))
    (NP (DT A) (NN group))))
  (VP (VBZ is)
   (NP
    (NP (DT a)
     (ADJP (RB nonempty) (VBN set))
     (NN G))
    (SBAR
     (WHPP (IN on)
      (WHNP (WDT which)))
     (S
      (NP (EX there))
      (VP (VBZ is)
       (VP (VBN defined)
        (NP (DT a) (JJ binary) (NN operation)
         (PRN (-LRB- -LRB-)
          (NP
           (NP (SYM a))
           (, ,)
           (NP (SYM b)))
          (-RRB- -RRB-))
         (NN ab))
        (S
         (VP (VBG satisfying)
          (NP (DT the) (VBG following) (NNS properties)))))))))
  (. .)))
```

```
(ROOT
 (FRAG
```

```
    (NP (NNP Closure))
    (: :)
    (S
      (SBAR (IN If)
        (S
          (NP
            (NP (SYM a))
            (CC and)
            (NP (SYM b)))
          (VP (VBP belong)
            (PP (TO to)
              (NP (NNP G))))))
      (, ,)
      (NP (RB then) (NN ab))
      (VP (VBZ is)
        (ADVP (RB also))
        (PP (IN in)
          (NP
            (NP (NNP G))
            (: ;)
            (NP
              (NP (NNP Associativity))
              (: :)
              (NP
                (NP (DT a) (-LRB- -LRB-) (NN bc) (-RRB- -RRB-))
                (VP (JJ =)
                  (NP (-LRB- -LRB-) (NNS ab) (-RRB- -RRB-))
                  (PP
                    (X (SYM c))
                    (IN for)
                    (NP
                      (NP (PDT all) (DT a))
                      (, ,)
                      (NP (SYM b)))))))
            (, ,)
            (X (SYM c))))))
    (. .)))


(ROOT
  (FRAG
    (NP
      (NP (NNP G))
      (: ;)
      (NP (NNP Identity)))
    (: :)
```

```
  (S
    (NP (EX There))
    (VP (VBZ is)
      (NP (DT an) (NN element) (CD 1))))
  (. .)))

(ROOT
  (S
    (S
      (VP (VBG G)
        (S
          (NP (PDT such) (DT that) (NN a1))
          (VP (SYM =)
            (NP (JJ 1a) (FW =) (FW a))))
        (PP (IN for)
          (NP (PDT all) (DT a)))
        (PP (IN in)
          (NP
            (NP (NNP G))
            (: ;)
            (NP (NNP Inverse))))))
    (: :)
    (S
      (SBAR (IN If)
        (S
          (NP (SYM a))
          (VP (VBZ is)
            (PP (IN in)
              (NP (NNP G))))))
      (, ,)
      (ADVP (RB then))
      (NP (EX there))
      (VP (VBZ is)
        (NP
          (NP (DT an) (NN element))
          (ADJP (JJ a-1)
            (PP (IN in)
              (NP (NNP G) (JJ such))))
          (SBAR (IN that)
            (S
              (NP (JJ aa-1) (JJ =) (NN a-1a))
              (VP (SYM =)
                (NP (CD 1)))))))))
    (. .)))
```

```
(ROOT
  (S
    (NP
      (NP
        (NP (DT A) (NN group))
        (`` G)
        (S
          (S
            (VP (VBZ is)
              (ADJP (JJ abelian))
              (SBAR (IN if)
                (S
                  (NP (DT the) (JJ binary) (NN operation))
                  (VP (VBZ is)
                    (ADJP (JJ commutative)))))))
          (, ,) (FW i.e.) (, ,)
          (S
            (VP (VBN ab)
              (S
                (VP (SYM =)
                  (NP (NN ba))))
              (PP (IN for)
                (NP (PDT all) (DT a))))))
        (, ,)
        (SBAR
          (WHNP
            (NP (SYM b))
            (PP (IN in)
              (NP (NNP G.))))
          (S
            (PP (IN In)
              (NP (DT this) (NN case)))
            (NP (DT the) (JJ binary) (NN operation))
            (VP (VBZ is)
              (ADVP (RB often))
              (VP (VBN written)
                (ADVP (RB additively))
                (PRN (-LRB- -LRB-)
                  (FRAG
                    (NP (-LRB- -LRB-)
                      (NP
                        (NP (SYM a))
                        (, ,)
                        (NP (SYM b)))
                      (-RRB- -RRB-) (FW a)
                      (NP
```

```
                (NP (NNP +))
                (NP (SYM b)))))
            (-RRB- -RRB-))))))
      (, ,))
     (PP (IN with)
      (NP (DT the) (NN identity))))
    (VP (VBN written)
     (PP (IN as)
      (NP
       (NP (CD 0))
       (CONJP (RB rather) (IN than))
       (NP (CD 1)))))
    (. .)))


(ROOT
 (S
  (NP (EX There))
  (VP (VBP are)
   (NP
    (NP (DT some)
     (ADJP (RB very) (JJ familiar))
     (NNS examples))
    (PP (IN of)
     (NP
      (NP (NN abelian) (NNS groups))
      (PP (IN under)
       (NP
        (NP (NN addition))
        (, ,)
        (NP (RB namely) (DT the) (JJ integers) (NN Z))
        (, ,)
        (NP (DT the) (JJ rationals) (NN Q))
        (, ,)
        (NP (DT the) (JJ real) (NNS numbers) (NN R))
        (, ,)
        (NP (DT the) (JJ complex) (NNS numers) (NN C))
        (, ,)
        (CC and)
        (NP (DT the) (JJ integers) (NNP Zm) (NN modulo) (NN m.))))))
    (SBAR
     (S
      (NP (NNP Nonabelian) (NNS groups))
      (VP (MD will)
       (VP (VB begin)
        (S
```

```
            (VP (TO to)
              (VP (VB appear)
                (PP (IN in)
                  (NP (DT the) (JJ next) (NN section))))))))))))
  (. .)))


(ROOT
  (S
    (NP
      (NP (DT The) (JJ associative) (NN law) (NNS generalizes))
      (PP (TO to)
        (NP
          (NP (NNS products))
          (PP (IN of)
            (NP
              (NP (DT any) (JJ finite) (NN number))
              (PP (IN of)
                (NP
                  (NP
                    (NP
                      (NP (NNS elements))
                      (, ,)
                      (PP (IN for)
                        (NP (NN example)))
                      (, ,))
                    (PRN (-LRB- -LRB-)
                      (NP (NN ab))
                      (-RRB- -RRB-)))
                  (PRN (-LRB- -LRB-)
                    (NP (NN cde))
                    (-RRB- -RRB-)))))))))
    (VP (JJ =)
      (S
        (NP (DT a) (-LRB- -LRB-) (NN bcd) (-RRB- -RRB-))
        (VP (VB e.)
          (SBAR
            (S
              (NP (DT A) (JJ formal) (NN proof))
              (VP (MD can)
                (VP (VB be)
                  (VP (VBN given)
                    (PP (IN by)
                      (NP (NN induction))))))))))
    (. .)))
```

```
(ROOT
  (S
    (S
      (SBAR (IN If)
        (S
          (NP
            (NP (CD two) (NNS people))
            (NP (DT A)
              (CC and)
              (NN B)))
          (VP (NN form)
            (NP
              (NP (JJ a1) (DT an))
              (PP (IN in)
                (NP (JJ different) (NNS ways)))))))
      (, ,)
      (NP
        (NP (DT the) (JJ last) (NN multiplication))
        (VP (VBN performed)
          (PP (IN by)
            (NP (NNP A)))))
      (VP (MD might)
        (VP (VB look)
          (PP (IN like)
            (NP
              (NP (-LRB- -LRB-)
                (QP (CD a1) (CD ai))
                (-RRB- -RRB-))
              (PRN (-LRB- -LRB-)
                (VP (VBP ai)
                  (NP (CD +1) (DT an)))
                (-RRB- -RRB-)))))))
    (, ,)
    (CC and)
    (S
      (NP
        (NP (DT the) (JJ last) (NN multiplication))
        (PP (IN by)
          (NP (NN B))))
      (VP (MD might)
        (VP (VB be)
          (NP
            (NP (-LRB- -LRB-) (JJ a1) (NN aj) (-RRB- -RRB-))
            (PRN (-LRB- -LRB-)
              (VP (VBN aj)
```

```
            (NP (CD +1) (DT an)))
           (-RRB- -RRB-))))))
   (. .)))


(ROOT
  (S
   (NP (PRP We))
   (VP (VBP have)
    (NP (DT the) (JJ following) (NN result)))
   (. .)))

(ROOT
  (S
   (S
    (SBAR
     (NP (CD 1.1.5) (NN Proposition))
     (IN If)
     (S
      (NP
       (NP (JJ Gis) (DT a) (JJ cyclic) (NN group))
       (PP (IN of)
        (NP (NN order))))
      (VP (VBD ngenerated)
       (PP (IN by)
        (NP (NNP a))))))
    (, ,)
    (NP (DT the) (JJ following) (NNS conditions))
    (VP (VBP are)
     (ADJP (JJ equivalent))))
   (: :)
   (S
    (LST (LS r))
    (NP
     (NP
      (LST (-LRB- -LRB-) (DT a) (-RRB- -RRB-))
      (DT a) (JJS =) (NN n.)
      (PRN (-LRB- -LRB-)
       (X (SYM b))
       (-RRB- -RRB-))
      (NN r))
     (CC and)
     (NP (NN n)))
    (VP (VBP are)
     (ADJP (RB relatively) (JJ prime))))
   (. .)))
```

```
(ROOT
  (S
    (S
      (LST (-LRB- -LRB-) (NN c) (-RRB- -RRB-))
      (NP (SYM r))
      (VP (VBZ is)
        (NP
          (NP (DT a) (NN unit) (NN mod) (NN n))
          (, ,)
          (PP (IN in)
            (NP (JJ other) (NNS words))))))
    (, ,)
    (NP (SYM r))
    (VP (VBZ has)
      (NP (DT an) (JJ inverse) (NN mod) (NN n))
      (PRN (-LRB- -LRB-)
        (S
          (NP (DT an) (NN integer))
          (VP (VBZ s)
            (ADJP (JJ such)
              (SBAR (IN that)
                (S
                  (NP (NN rs))
                  (VP (SYM =)
                    (NP (JJ 1mod) (NN n))))))))
        (-RRB- -RRB-)))
    (. .)))

(ROOT
  (S
    (ADVP (RB Furthermore))
    (, ,)
    (NP
      (NP (DT the) (VBN set) (NN Un))
      (PP (IN of)
        (NP (NNS units) (NN mod) (NN n))))
    (VP (VBZ forms)
      (NP (DT a) (NN group))
      (PP (IN under)
        (NP (NN multiplication))))
    (. .)))

(ROOT
  (S
    (NP
```

```
    (NP (DT The) (NN order))
    (PP (IN of)
      (NP (DT this) (NN group))))
  (VP (VBZ is))
  (. .)))

(ROOT
 (S
  (LST (-LRB- -LRB-) (LS n) (-RRB- -RRB-))
  (VP
   (VP (VB =)
    (NP
     (NP (DT the) (NN number))
     (PP (IN of)
      (NP (JJ positive) (NNS integers))))
    (ADVP (RBR less) (IN than)))
   (CC or)
   (VP (VB equal)
    (PP (TO to)
     (NP
      (NP (NN n))
      (SBAR
       (WHNP (WDT that))
       (S
        (VP (VBP are)
         (ADJP (RB relatively) (JJ prime)
          (PP (TO to)
           (NP (NN n)))))
        (: ;)))))))
  (. .)))
```

APPENDIX F [Noun Phrases Extracted]

Using the sql to sort through the various noun phrases extracted for the two chapters we encounter interesting areas of conceptual areas picked up by noun phrases. Since the chapter covered rings in detail some interesting features of rings are revealed using the SQL:

```
SELECT tphrase
FROM Book_NP
WHERE (((Len(tphrase))<50) and Len(tphrase) > 20 and Instr(1,tphrase,"ring") > 0)
GROUP BY tphrase
ORDER BY Book_NP.Tphrase;
```

We had 38 hits out of 8232 noun phrases here are the 38 hits for rings in noun phrases:

QuerysimpleNps
tphrase
(NP (DT a) (JJ commutative) (NN ring))
(NP (DT a) (JJ noncommutative) (NN ring))
(NP (DT a) (JJ quotient) (NN ring))
(NP (DT a) (NN division) (NN ring))
(NP (DT a) (NN ring) (NN ho))
(NP (DT a) (NN ring) (NN homomorphism))
(NP (DT a) (NN ring) (NN isomorphism))
(NP (DT a) (NN ring) (NN R) (NN))
(NP (DT A) (NN ring) (NN R))
(NP (DT a) (NN ring))
(NP (DT A) (NN subring))
(NP (DT all) (NNS subrings))
(NP (DT an) (JJ arbitrary) (NN ring))
(NP (DT Any) (NN ring) (NN homomorphism))
(NP (DT any) (NN ring))
(NP (DT every) (JJ) (NN ring))
(NP (DT every) (NN ring))
(NP (DT the) (JJ) (NN ring) (NN R))
(NP (DT the) (NN ring) (NN R))
(NP (DT the) (NN ring) (NN Ris))
(NP (DT the) (NN ring) (NN Z))
(NP (DT the) (NN ring) (NN Zn))
(NP (DT the) (NN ring) (NNP R.))
(NP (DT the) (NN ring) (NNS properties))

QuerysimpleNps
tphrase
(NP (DT the) (NN ring))
(NP (JJ) (JJ polynomial) (NNS rings))
(NP (JJ) (NNS rings))
(NP (JJ commutative) (NNS rings))
(NP (NN ring) (NN homomorphism))
(NP (NN ring) (NNS isomorphisms))
(NP (NN subring) (CC and) (NN ideal))
(NP (NNP QUOTIENT) (NNP RINGS))
(NP (NNP Quotient) (NNPS Rings) (NNP))
(NP (NNP RING) (NNP FUNDAMENTALS) (NNP))
(NP (NNP RING) (NNP FUNDAMENTALS))
(NP (NNP RINGS) (CD 9))
(NP (NNP Rings) (NNP))
(NP (NNS subrings))

Here are some other noun phrases manually mined from the Book_NP table using query:

```
SELECT tphrase
FROM Book_NP
WHERE (((Len(tphrase))<50) and Len(tphrase) > 20)
GROUP BY tphrase
ORDER BY Book_NP.Tphrase;
```

QuerysimpleNps
tphrase
(NP (CD two) (JJ basic) (NNS operations))
(NP (CD two) (JJ even) (NNS permutations))
(NP (CD two) (JJ opposite) (NNS sides))
(NP (CD two) (JJ polynomials) (NN A))
(NP (CD two) (JJ proper) (NNS subgroups))
(NP (CD two) (NN disjoint))
(NP (CD two) (NN nonzero) (NNS matrices))
(NP (CD two) (NNS cosets))
(NP (CD two) (NNS ideals))
(NP (CD two) (NNS integers))
(NP (CD two) (NNS people))
(NP (CD two) (NNS stages))
(NP (CD two) (NNS vertices))
(NP (CD zero) (NNS divisors))
QuerysimpleNps
tphrase
(NP (DT a) (JJ linear) (NN combination))
(NP (DT a) (JJ maximal) (NN element))
(NP (DT a) (JJ maximal) (NN ideal))
(NP (DT a) (JJ noncommutative) (NN ring))
(NP (DT a) (JJ nonconstant) (NN element))
(NP (DT a) (JJ nonempty) (NN subset))
(NP (DT a) (JJ nonzero) (NN element))
(NP (DT a) (JJ nonzero) (NN matrix))
(NP (DT a) (JJ particular) (NN element))
(NP (DT a) (JJ positive) (NN integer))
(NP (DT A) (JJ prime) (NN ideal))
(NP (DT a) (JJ prime) (NN number))
(NP (DT a) (JJ principal) (NN ideal))

QuerysimpleNps
tphrase
(NP (DT a) (JJ proper) (JJ ideal) (NN P))
(NP (DT a) (JJ proper) (NN ideal))
(NP (DT a) (JJ proper) (NN subgroup))
(NP (DT a) (JJ quotient) (NN ring))
(NP (DT a) (JJ regular) (NN pentagon))
(NP (DT a) (JJ regular) (NN polygon))
(NP (DT a) (JJ right) (NN ideal))
(NP (DT a) (JJ rigid) (NN motion))
(NP (DT a) (JJ similar) (NN argument))
(NP (DT a) (JJ single) (NN element))
QuerysimpleNps
tphrase
(NP (DT a) (NN group) (NN homomorphism))
(NP (DT A) (NN group))
(NP (DT a) (NN homomorphism))
(NP (DT a) (NN left))
(NP (DT a) (NN line))
(NP (DT a) (NN mod) (NN m))
(NP (DT a) (NN mod) (NN mn))
(NP (DT A) (NN monoid))
(NP (DT a) (NN monomorphism))
(NP (DT a) (NN multiple))
(NP (DT a) (NN n) (NN b))
(NP (DT a) (NN nonempty))
(NP (DT A) (NN nonzero) (NN polynomial))
(NP (DT A) (NN permutation) (NN p))
(NP (DT A) (NN permutation))
(NP (DT a) (NN product))
(NP (DT a) (NN quaternion) (NN h))
(NP (DT a) (NN ring) (NN ho))
(NP (DT a) (NN ring) (NN homomorphism))
(NP (DT a) (NN ring) (NN isomorphism))
(NP (DT a) (NN ring) (NN R) (NN))
(NP (DT A) (NN ring) (NN R))
(NP (DT a) (NN ring))
(NP (DT a) (NN semigroup) (NN))
(NP (DT A) (NN semigroup))

QuerysimpleNps
tphrase
QuerysimpleNps
tphrase
(NP (DT an) (NN abelian) (NN group))
(NP (DT an) (NN appeal))
(NP (DT an) (NN elem))
(NP (DT an) (NN element) (CD 1))
(NP (DT an) (NN element) (NN))
(NP (DT an) (NN element) (NN g) (JJ such))
(NP (DT an) (NN element))
(NP (DT an) (NN endomorphism))
(NP (DT an) (NN epimorphism) (: ;) (JJ))
(NP (DT an) (NN epimorphism))
(NP (DT an) (NN example))
(NP (DT An) (NN ideal))
(NP (DT an) (NN integer))
(NP (DT an) (NN isomorphism))
(NP (DT an) (NN object))
(NP (DT an) (NN -RRB-))
(NP (DT an) (RB even))
(NP (DT an) (-RRB- -RRB-) (NNP +))
(NP (DT another) (JJ) (NN class))
(NP (DT another) (NN corollary))
(NP (DT another) (NN element))
(NP (DT any) (CD two) (NNS elements))
(NP (DT any) (JJ finite) (NN number))
(NP (DT any) (JJ major) (NN theorem))
(NP (DT any) (NN element) (NN x.) (NNP S))
(NP (DT any) (NN field))
(NP (DT any) (NN ideal))
(NP (DT any) (NN matrix))
(NP (DT any) (NN permutation) (NN p))
(NP (DT any) (NN permutation))
(NP (DT any) (NN product))
(NP (DT any) (NN r))
(NP (DT Any) (NN ring) (NN homomorphism))
(NP (DT any) (NN ring))
(NP (DT any) (NN time))

QuerysimpleNps
tphrase
(NP (DT any) (NN transposition))
(NP (DT any) (VBN given) (NN) (NN group))
(NP (DT both) (JJ odd))
(NP (DT Both) (NNP X.))
(NP (DT both) (NNS sides))
(NP (DT each) (FW i))
(NP (DT each) (NN case))
(NP (DT each) (NN coset))
(NP (DT each) (NN element))
(NP (DT each) (NN Ij))
(NP (DT each) (NN reflection))
(NP (DT every) (JJ) (NN ring))
(NP (DT every) (JJ nonzero) (NN element))
(NP (DT every) (JJ positive) (NN integer))
(NP (DT every) (JJ proper) (NN ideal))
(NP (DT Every) (NN chain))
(NP (DT every) (NN ideal))
(NP (DT every) (NN matr))
(NP (DT every) (NN matrix))
(NP (DT every) (NN permutation) (NN p))
(NP (DT every) (NN r))
(NP (DT every) (NN ring))
(NP (DT Neither))
(NP (DT no) (CD zero) (NNS divisors))
(NP (DT no) (JJ constant) (NN term))
(NP (DT no) (JJ nontrivial) (NN left))
(NP (DT no) (NN difference))
(NP (DT no) (NN multiple))
(NP (DT no) (NNS cycles))
(NP (DT no) (NNS ideals))
(NP (DT no) (NNS repetitions))
(NP (DT some) (NN j))
(NP (DT some) (NN t))
(NP (DT some) (NN x.) (NN) (NN))
(NP (DT that) (NN))
(NP (DT that) (NN A))
(NP (DT that) (NN ab))

QuerysimpleNps
tphrase
(NP (DT that) (NN ax))
(NP (DT that) (NN bj) (NN +) (NN cj))
(NP (DT that) (NN n1))
(NP (DT that) (NN R))
(NP (DT that) (NN r.))
(NP (DT that) (NN R\/R))
(NP (DT that) (NN xr))
(NP (DT the) (CD four) (NN group))
(NP (DT The) (CD zero) (NN element))
(NP (DT the) (JJ) (NN collection))
(NP (DT the) (JJ) (NN corresponding))
(NP (DT the) (JJ) (NN intersection))
(NP (DT the) (JJ) (NN inverse))
(NP (DT the) (JJ) (NN kernel))
(NP (DT the) (JJ) (NN order))
(NP (DT the) (JJ) (NN pentagon))
(NP (DT the) (JJ) (NN result))
(NP (DT the) (JJ) (NN ring) (NN R))
(NP (DT the) (JJ) (NN subset))
(NP (DT the) (JJ) (NN technique))
(NP (DT the) (JJ) (NNS cosets))
(NP (DT the) (JJ) (NNS cycles))
(NP (DT the) (JJ) (NNS integers) (NN Z))
(NP (DT the) (JJ above) (CD) (NNS rules))
(NP (DT the) (JJ above) (NN example))
(NP (DT the) (JJ additive) (NN group))
(NP (DT the) (JJ additive) (NN identity))
(NP (DT the) (JJ alternating) (NN group))
(NP (DT the) (JJ associative) (NN law))
(NP (DT the) (JJ binary) (NN operation))
(NP (DT the) (JJ binomial) (NN theorem))
(NP (DT the) (JJ canonical) (JJ) (NN))
(NP (DT the) (JJ Chinese) (NN remainder))
(NP (DT The) (JJ concrete) (NN version))
(NP (DT the) (JJ coset) (NN rs))
(NP (DT the) (JJ dihedral) (NN group))
(NP (DT the) (JJ direct) (NN product))

QuerysimpleNps
tphrase
(NP (DT the) (JJ distributive) (NNS laws))
(NP (DT the) (JJ equivalence) (NN class))
(NP (DT the) (JJ familiar) (NN Euler))
(NP (DT the) (JJ finite) (NNS sums))
(NP (DT the) (JJ first) (NN vertex))
(NP (DT the) (JJ first))
(NP (DT the) (JJ following) (NN result))
(NP (DT the) (JJ free) (NN group))
(NP (DT the) (JJ ideal) (NN I))
(NP (DT the) (JJ ideal) (NN I0))
(NP (DT the) (JJ ideal) (NN R1) (NN ×))
(NP (DT the) (JJ indeterminate) (NN X))
(NP (DT the) (JJ integer) (CD 1))
(NP (DT The) (JJ integers) (DT a))
(NP (DT the) (JJ integers) (NN mi))
(NP (DT the) (JJ integers) (NN Z))
(NP (DT the) (JJ last) (NN equality))
(NP (DT the) (JJ last) (NN multiplication))
(NP (DT the) (JJ left) (JJ ideal) (NN I))
(NP (DT the) (JJ left) (NN coset))
(NP (DT the) (JJ left) (NN side))
(NP (DT the) (JJ left) (NNS cosets))
(NP (DT the) (JJ matrix) (NN))
(NP (DT the) (JJ natural) (NN map))
(NP (DT the) (JJ natural) (NN way))
(NP (DT the) (JJ normal) (NN subgroup))
(NP (DT the) (JJ only) (NNS ideals))
(NP (DT the) (JJ opposite) (NN side))
(NP (DT the) (JJ other) (JJ) (NN half))
(NP (DT the) (JJ other) (NN hand))
(NP (DT the) (JJ prime) (NN factorization))
(NP (DT the) (JJ principal) (NN ideal))
(NP (DT the) (JJ rationals) (NN Q))
(NP (DT the) (JJ regular) (NN n-gon))
(NP (DT the) (JJ right) (JJ ideal) (NN I))
(NP (DT the) (JJ right) (NN coset))
(NP (DT the) (JJ same))

QuerysimpleNps
tphrase
(NP (DT the) (JJ second) (NN row))
(NP (DT The) (JJ standard) (NN proof))
(NP (DT the) (JJ symmetric) (NN group))
(NP (DT the) (JJS largest) (NN ideal))
(NP (DT the) (JJS least) (NN))
(NP (DT the) (JJS least) (NN common))
(NP (DT the) (JJS smallest) (NN ideal))

APPENDIX G [DOT Language Scripts]

This is the script coded to create the tree node diagrams in the thesis. It is based on the "DOT" language implemented in the "Graphviz tool" by AT&T (http://www.graphviz.org).

Here are the scripts we created for use in HPSG section using the DOT language:

```
digraph HFP
{
 "a" [label="[LOC|CAT[HEAD-4,SUBCAT<>]]=(=S[fin])"]
 "c" [label="1"]
 "b" [label="[LOC|CAT[HEAD-4,SUBCAT<1>]]=(=VP[fin])"]
 "d" [label="[LOC|CAT[HEAD-4,verb[fin]SUBCAT<1-NP[nom](3rd,sing),2-   NP[acc],3-
NP[acc]>]]"]
 "e" [label="2"]
 "f" [label="3"]
  a->c[label="C"];
  c->UMP
  a->b[label="H"];
  b->d[label="H"];
  b->e[label="C1"];
  b->f[label="C2"];
  d->gives;
  e->homomorphism;
  f->g1;
}
digraph schema1
{
  a [label="[HEAD-1, SUBCAT<>]"]
  c [label="2"]
  b [label="[HEAD-1, SUBCAT<2>]"]
  a->c[label="C"]
  a->b[label="H"]

}
```

Here are the scripts we created for use in “Stanford Parser” section under methodology:

```
graph treefactoring
{
	label="Tree Factoring"
	label="Original==>Right-Factored or Left-Factored"
	subgraph cluster01 {
	 label="Original"
	 A--B
	 A--C
	 A--D
	}

	subgraph cluster02 {
	 label="Right-Factored"
	 A1 [label="A"]
	 B1 [label="B"]
	 C1 [label="C"]
	 D1 [label="D"]
	 A1--B1
	 A1--"A|<C-D>"
	 "A|<C-D>"--C1
	 "A|<C-D>"--D1
	}

	subgraph cluster03 {
	 label="Left-Factored"
	 A2 [label="A"]
	 B2 [label="B"]
	 C2 [label="C"]
	 D2 [label="D"]
	 E2 [label="A|<C-D>"]
	 A2--E2
	 A2--D2
	 E2--B2
	 E2--C2
	}
}
```

```
graph treefactoring1
{
label="Tree Factoring"
label="Original==>Right-Factored or Left-Factored"
            subgraph cluster01 {
             label="Original"
             A--B
             A--C
             A--D
            }

            subgraph cluster02 {
             label="Right-Factored"
             A1 [label="A"]
             B1 [label="B"]
             C1 [label="C"]
             D1 [label="D"]
             A1--B1
             A1--"A|<C-D>"
             "A|<C-D>"--C1
             "A|<C-D>"--D1
            }

            subgraph cluster03 {
             label="Left-Factored"
             A2 [label="A"]
             B2 [label="B"]
             C2 [label="C"]
             E2 [label="A|<B-C>"]
             D2 [label="D"]

             A2--E2
             A2--D2
             E2--B2
             E2--C2
            }
}
```

```
graph treefactoring2
{
label="Annotation [? stands for the parent of A]"
          subgraph cluster01 {
           label="Original"
           A--B
           A--C
           A--D
           A--E
           A--F
}

          subgraph cluster02 {
           label="No Smoothing"
           A1 [label="A"]
           B1 [label="B"]
           C1 [label="A|<C-D-E-F>"]
           D1 [label="C"]
           E1 [label="..."]

           A1--B1
           A1--C1
           C1--D1
           C1--E1
          }
          subgraph cluster03 {
           label="Markov order 1"
           A2 [label="A"]
           B2 [label="B"]
           C2 [label="A|<C>"]
           D2 [label="C"]
           E2 [label="..."]

           A2--B2
           A2--C2
           C2--D2
           C2--E2
          }
          subgraph cluster04 {
           label="Markov order 2"
           A3 [label="A"]
           B3 [label="B"]
           C3 [label="A|<C-D>"]
           D3 [label="C"]
           E3 [label="..."]

           A3--B3
           A3--C3
           C3--D3
           C3--E3
          }}
```

APPENDIX H [Software Configuration]

The software installs required to be installed to run the KSM framework for any document are listed in this section. KSM framework as tested can work on the Windows platform, since it is written in Java and uses MSQL database this can be ported to any platform in theory. We need the following software installs for the KSM framework to work:

1) Java runtime is the language platform used for this project it is available at:

http://java.sun.com/javase/downloads/index.jsp.

2) Eclipse is the IDE platform recommended here is the download for this:

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/galileo/SR1/eclipse-java-galileo-SR1-win32.zip

Here is a quick tutorial for working with Eclipse

http://eclipsetutorial.sourceforge.net/Total_Beginner_Companion_Document.pdf

3) Stanford natural language parser (version 1.6) is used for KSM framework here is the installation information:

http://nlp.stanford.edu/software/lex-parser.shtml#Download

4) Python compiler version 2.6 [Optional]

Download from (and install in default directory):

http://www.python.org/download/releases/2.6/

5)Python Natural language tool kit [http://www.nltk.org/download [Optional]]

Download from (and install in the default directory)

[Install the Natural language Site packages]
http://nltk.googlecode.com/files/nltk-2.0b4.win32.msi

[Installs Statistical libraries]
http://sourceforge.net/projects/numpy/files/NumPy/numpy-1.3.0-win32-superpack-python2.6.exe

[Install corporas including the Treebank corpus]

http://www.nltk.org/data

We use the Treebank in NLTK to generate PCFG grammar from the parsed text. Here is a readme file for more details. http://corpora-modeling-file-share.googlegroups.com/web/PCFGGenration-ReadMe.txt

Download the .mrg file which contains the Stanford PCFG parse trees of Abstract Algebra book by Robert Ash's of Chapter 1. http://corpora-modeling-file-share.googlegroups.com/web/wsj_0001.mrg this file is required as specified in the above readme file. Also here is the Python script file as referred in the readme file:

http://corpora-modeling-file-share.googlegroups.com/web/grammar.py

6) MYSQL Database [Open Source]

a)Database Install:
http://dev.mysql.com/get/Downloads/MySQL-5.1/mysql-essential-5.1.39-win32.msi

b)JDBC driver Install:
http://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.10.zip/from/pick#mirrors

7)Microsoft Access 2007 Install [Optional, Not free]

http://www.microsoft.com/downloads/details.aspx?familyid=d9ae78d9-9dc6-4b38-9fa6-2c745a175aed&displaylang=en

8)Microsoft Word 2007 Install [Optional, Not free]

This is used if user wants to use Microsoft Word to markup the Mathematical text. The user is expected to select sentences and the VBA script attached in Appendix I will send the sentences to be parsed and bring the result back as marked up comments. These comments mark the boundary of the lines or sentences selected by the user.

http://office.microsoft.com/en-us/downloads/CD102094631033.aspx

Here is the template file which the user can use to interact with the parser using Microsoft word. http://corpora-modeling-file-share.googlegroups.com/web/ThesisProject.dotm

Refer to the readme file here for details http://corpora-modeling-file-share.googlegroups.com/web/MSWord-ReadMe.txt

We need 2 more files to make the Word interface work. MSWordEx.jar, App.config here are the links to download them:

http://corpora-modeling-file-share.googlegroups.com/web/MSWordEx.jar

http://corpora-modeling-file-share.googlegroups.com/web/app.config

9) Empty Access Database [Optional]

http://corpora-modeling-file-share.googlegroups.com/web/ThesisIntroduction2.zip

10) Empty My SQL Database

http://corpora-modeling-file-share.googlegroups.com/web/Thesisdatabasebackup_MYSQL+20090929+1010.zip

Refer to http://corpora-modeling-file-share.googlegroups.com/web/Database-ReadMe.txt , http://corpora-modeling-file-share.googlegroups.com/web/Database-ReadMe2.txt files for complete instructions.

11)KSM framework Thesisproject.jar configuration

Download from:

http://corpora-modeling-file-share.googlegroups.com/web/thesisproject.jar

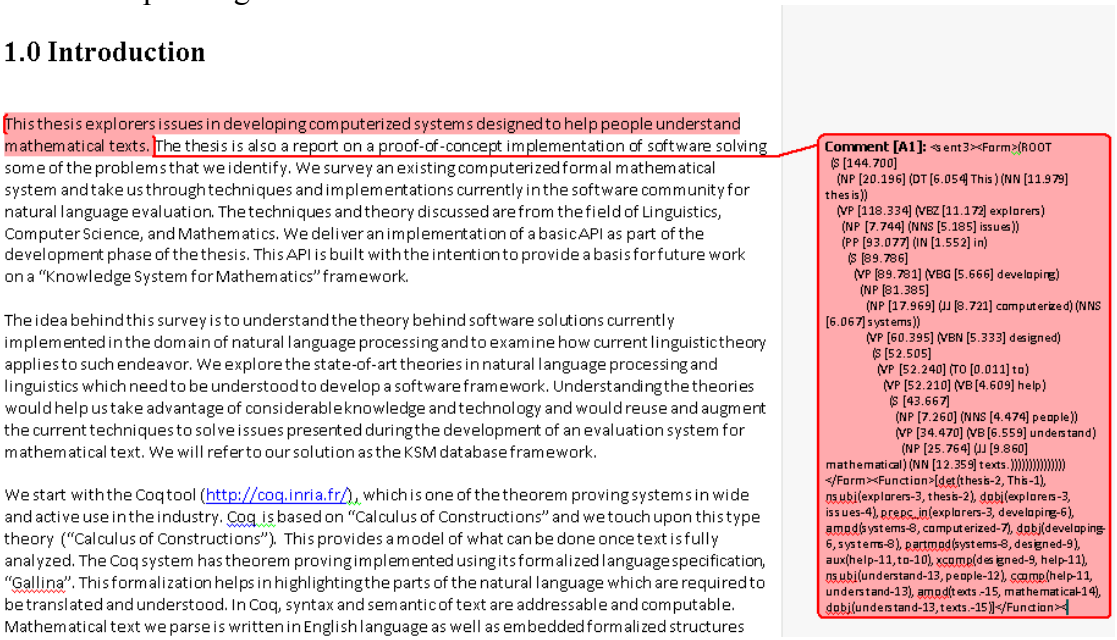
Here are the configuration settings for the jar file (app.config):

```
#This is the product version number for the KSMPackage
app.version=1.0
#This is the array of files to be parsed at one time (maximum of 3 files), the
#values here consists of chapter number and the file location for each chapter
app.inputinfo=1,c:\\thesis\\Introduction1.txt;2,c:\thesis\\chapter2.txt;
#This is the grammar file which has the lexicon, unary and binary grammar rules
#generated from Penn-Treebank by Stanford nlp team
app.treebankfile=C:\\nlpparserstanford\\stanford-parser-2007-08-
19\\englishPCFG.ser.gz
#Here are the parameters to be passed to the parser
app.parserparams=-maxLength,80,-retainTmpSubcategories
#This is the file written for word to display as comments after the request for
#parsing is complete
app.fileforword=c:\\readinword.txt
#Here is the sample parameters for For MySQL Database. These consist of class
name, connection string to MySql database and dbtype
app.Classname=com.mysql.jdbc.Driver
app.Connectionstring=jdbc:mysql://localhost/ThesisIntroduction2?user=root&pas
sword=aaaaaaaa
app.Dbtype=2
#Here is the sample parameters for For Microsoft Access 2007 Database. These
consist of class name, connection string to MySql database and dbtype
#app.Classname=sun.jdbc.odbc.JdbcOdbcDriver
#app.Connectionstring=jdbc:odbc:Driver={Microsoft Access Driver (*.mdb,
*.accdb)};DBQ=c:\thesisIntroduction.accdb
#app.Dbtype=1
#Here is the pattern matching parameters function in Corporaparse.java using
tregex tool. We evaluate the chapter's sentences and match them for a pattern
specified here. The output of this process goes to Book_pattern table.
app.searchpattern=@VP^PP
app.patternrun=0
app.patternchapternumber=1
```

Here is the complete source code for the KSM framework: http://corpora-modeling-file-share.googlegroups.com/web/sourceofthesisproject.zip

APPENDIX I [MS Word Integration]

Here is a listing of the Visual basic for application script which is in the ThesisProject.dotm file. It enables the user to select a sentence or lines in word and hit CTRL+"." Key to send the line for parsing to the thesisproject.jar file. Once parsed (using dependency and part of spec and phrase parsing) the result is available as comments attached to the sentence selected via balloon markers which are normally used for adding comments in words. The shape for these markings can be changed. Attached below the screenshot is the VBA script we use for the integration of the parser with Microsoft Word. This script assumes c:\MSWordEx.jar, c:\app.config and c:\readinword.txt files are file locations for the executable, configuration and the text file. This text file is the output file where the parser writes the Xml formatted output of the PCFG and dependency parse. The location of the text file is also specified in the app.config file. Here is the screen snapshot of the commenting feature after selecting a sentence and pressing CTRL+ ".".



[Figure 14]

```
Public Declare Function ShellExecute _
    Lib "shell32.dll" _
    Alias "ShellExecuteA" ( _
    ByVal hwnd As Long, _
    ByVal lpOperation As String, _
    ByVal lpFile As String, _
    ByVal lpParameters As String, _
    ByVal lpDirectory As String, _
    ByVal nShowCmd As Long) _
    As Long
Declare Sub Sleep Lib "Kernel32" (ByVal dwMilliseconds As Long)
Sub RunParser()
    Dim strFile As String
    Dim strAction As String
```

```
    Dim lngErr As Long
    Dim strselection As String
    ' Edit this:
    strFile = "java"  ' the file you want to open/etc.
    strAction = "OPEN"  ' action might be OPEN, NEW or other, depending on what you need to do
    DeleteFile ("c:\\readinword.txt")

    If Selection.Active = True Then
       ' MsgBox Selection.Information(wdFirstCharacterLineNumber)
        strselection = Selection.Text
        lngErr = ShellExecute(0, strAction, strFile, "-mx500m -jar c:\MSWordEx.jar " & Chr(34) & strselection & Chr(34) &
" " & Selection.Information(wdFirstCharacterLineNumber), "c:\", 0)
        Sleep 1000
    ' optionally, add code to test lngErr
    'read the file c:\readinword and add the comment to the selected sentence
    'get the line number
        MsgBox "Please wait for 5 seconds while the parser processes the sentence"
        tstr = GetText("c:\\readinword.txt", Selection.Information(wdFirstCharacterLineNumber))
        Call Selection.Comments.Add(Selection.Range, tstr)
     End If
End Sub

Function GetText(sFile As String, ilineno As Integer) As String
   Dim nSourceFile As Integer, sText As String

   "Close any open text files
   Close

   "Get the number of the next free text file
   nSourceFile = FreeFile

    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    While fso.FileExists(sFile) = False
       DoEvents
    Wend
    While InStr(1, sText, "<sent" & ilineno & ">") = 0
   "Write the entire file to sText

    Open sFile For Input As #nSourceFile
    sText = Input$(LOF(1), 1)
    Close
   Wend
   tstring = Mid(sText, InStr(1, sText, "<sent" & ilineno & ">"), InStr(1, sText, "</sent" & ilineno & ">"))
   GetText = tstring

End Function
Sub DeleteFile(tfile As String)
Dim fso
Dim file As String
file = tfile ' change to match the file w/Path
Set fso = CreateObject("Scripting.FileSystemObject")
If fso.FileExists(file) Then
    fso.DeleteFile file, True
End If
End Sub
```

APPENDIX J [Treebank Categories]

The Penn Treebank POS tagset.

1.	CC	Coordinating conjunction	25.	TO	*to*
2.	CD	Cardinal number	26.	UH	Interjection
3.	DT	Determiner	27.	VB	Verb, base form
4.	EX	Existential *there*	28.	VBD	Verb, past tense
5.	FW	Foreign word	29.	VBG	Verb, gerund/present participle
6.	IN	Preposition/subordinating conjunction	30.	VBN	Verb, past participle
7.	JJ	Adjective	31.	VBP	Verb, non-3rd ps. sing. present
8.	JJR	Adjective, comparative	32.	VBZ	Verb, 3rd ps. sing. present
9.	JJS	Adjective, superlative	33.	WDT	*wh*-determiner
10.	LS	List item marker	34.	WP	*wh*-pronoun
11.	MD	Modal	35.	WP$	Possessive *wh*-pronoun
12.	NN	Noun, singular or mass	36.	WRB	*wh*-adverb
13.	NNS	Noun, plural	37.	#	Pound sign
14.	NNP	Proper noun, singular	38.	$	Dollar sign
15.	NNPS	Proper noun, plural	39.	.	Sentence-final punctuation
16.	PDT	Predeterminer	40.	,	Comma
17.	POS	Possessive ending	41.	:	Colon, semi-colon
18.	PRP	Personal pronoun	42.	(	Left bracket character
19.	PP$	Possessive pronoun	43.	)	Right bracket character
20.	RB	Adverb	44.	"	Straight double quote
21.	RBR	Adverb, comparative	45.	`	Left open single quote
22.	RBS	Adverb, superlative	46.	``	Left open double quote
23.	RP	Particle	47.	'	Right close single quote
24.	SYM	Symbol (mathematical or scientific)	48.	''	Right close double quote

dep - dependent
    aux - auxiliary
        auxpass - passive auxiliary
        cop - copula
    conj - conjunct
    cc - coordination
    arg - argument
        subj - subject
            nsubj - nominal subject
                nsubjpass - passive nominal subject
            csubj - clausal subject
        comp - complement
            obj - object
                dobj - direct object
                iobj - indirect object
                pobj - object of preposition
            attr - attributive
            ccomp - clausal complement with internal su
            xcomp - clausal complement with external s
            compl - complementizer
            mark - marker (word introducing an advcl)
            rel - relative (word introducing a rcmod)
            acomp - adjectival complement
        agent - agent
    ref - referent
    expl - expletive (expletive *there*)
    mod - modifier
        advcl - adverbial clause modifier
        purpcl - purpose clause modifier
        tmod - temporal modifier
        rcmod - relative clause modifier
        amod - adjectival modifier
        infmod - infinitival modifier
        partmod - participial modifier
        num - numeric modifier
        number - element of compound number
        appos - appositional modifier
        nn - noun compound modifier
        abbrev - abbreviation modifier
        advmod - adverbial modifier
            neg - negation modifier
        poss - possession modifier
        possessive - possessive modifier (*'s*)
        prt - phrasal verb particle
        det - determiner
        prep - prepositional modifier
    sdep - semantic dependent
        xsubj - controlling subject