THE DEVELOPMENT OF PETALS:

PERSONALITY TAGGED LOGICAL STATISTICAL GENERATOR

by

Matthew Paul Huenerfauth

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences.

Spring 2001

Copyright 2001 Matthew Paul Huenerfauth
All Rights Reserved

THE DEVELOPMENT OF PETALS:

PERSONALITY TAGGED LOGICAL/STATISTICAL GENERATOR

by

Matthew Paul Huenerfauth

Approved: ____

Kathleen F. McCoy, PhD.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____

M. Sandra Carberry, PhD.
Chair of the Department of Computer and Information Sciences

Approved: ____

Conrado M. Gempesaw II, Ph.D.
Vice Provost for Academic Programs and Planning

ACKNOWLEDGMENTS

I would like to thank Dr. Kathleen McCoy for her guidance and support during my involvement with the ICICLE project. Her suggestions and encouragement have made this thesis both a successful and an enjoyable learning experience.

Lisa Michaud contributed to several discussions about the generation component and user model requirements, and Dr. Vijay Shanker encouraged my exploration of machine learning techniques in generation. Bowen Hui assisted with the initial data collection project.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT viii

INTRODUCTION: An Overview Of PeTaLS 1

CHAPTER 1: Deep Generation: Traditional Methods and New Ideas 3
Planning-Based Techniques 5
Schemata-Based Techniques 7

CHAPTER 2: Applying Machine learning to Natural language generation 9
Traditional Applications of ML to Understanding and Generation 9
Motivations for Using ML in PeTaLS 10
Reversing the Machine learning Paradigm 11
Drawing from Statistical Surface Generation Research 12
Ordering Bunsetsus: The Naïve Approach 13
Airline Information: A More Efficient Technique 14
Writing Personalities and Data Scarcity 15

CHAPTER 3: Acquiring the Training Data Corpus 17
Data Collection Project Overview 17
Tagging Process 18
Selecting the Tags 20
Realizing the Tags 20
Adding Tags to the ICICLE System 21

CHAPTER 4: Situating PeTaLS within ICICLE 22
Before We Get Started: Defining the Problem 22
Error Identification Phase 22
The User Models and History 23
The Knowledge Base 25
Output of PeTaLS 26

CHAPTER 5: The PeTaLS Architecture: Selection Component 27
What Can't We Learn? 27
Selecting a Formalism 28
Selection Component Operation 29

CHAPTER 6: The PeTaLS Architecture: Ordering Component 31
The Ordering Algorithm 31
Scoring the Sequences..32
Choosing a Markovian Subscore Algorithm..33
Cautious Results Analysis..36
CHAPTER 7: Final Thoughts and Future Directions 38
Future Development Issues..38
Conclusions..40
REFERENCES 42
APPENDIX A: Selection Component Code...46
APPENDIX B: Ordering Component Code..58

LIST OF TABLES

Table 1: Propositional Tag Types for ICICLE. 19

Table 2: User Model and KB Access Functions. 30

LIST OF FIGURES

Figure 1: A Training Set Example 11

Figure 2: Example of the Tagging Process. 18

Figure 3: Performance of Markovian Subscoring Algorithms.. 36

ABSTRACT

The subject of this thesis is the development of the PeTaLS generation architecture designed to work with the ICICLE English-writing tutorial system. The Personality Tagged Logical Statistical Generator uses machine learning and logical rule based techniques to accomplish the deep generation (content selection and propositional ordering) of the instructional text to be delivered to the student. Training on a corpus of text written by English as a Second Language instructors, the system can learn to mimic the informational organization strategy of individual authors. This thesis includes a comparison between PeTaLS and other deep generation architectures, the advantages of machine learning to generation, training data collection and analysis, and a discussion of system development considerations.

Introduction

AN OVERVIEW OF PETALS

The purpose of this thesis is to develop a text generation component for the Interactive Computer Identification and Correction of Language Errors (ICICLE) Project, an English-writing computerized tutorial system designed for deaf students native to American Sign Language [Michaud & McCoy 2000], [Schneider & McCoy 1998], [McCoy & Masterman 1997], and [McCoy, Pennington, and Suri 1996].

With some studies showing that only half of deaf eighteen year-olds are able to read at a fourth grade reading level [Conrad 1977], [DiFrancesca 1972], [Strong 1988], [Whightstone et al. 1963], there is a critical need for more educational resources for the deaf. ICICLE is designed to increase students' practice time with and exposure to language by complementing in-class instruction from a language teacher. Using ICICLE, a student who is learning to write in English could submit an essay she is working on into a special ICICLE word processor / grammar checker. After analyzing the student's text for language errors, the system would engage the user in a tutorial session to help her understand the writing errors she made and to learn how to correct them. Further discussion of the educational motivations behind ICICLE can be found in [Michaud et al. 2000].

While an eventual goal is to create an interactive tutorial dialogue, this thesis focuses on how to produce a single paragraph of explanation of a single error. This high-quality one-shot explanation of the user's grammatical error would begin the tutorial session and form the basis of further explanation. The dialogue could continue

from this point by answering the student's questions or by providing further explanation for topics which the student requests; however, these interactions are beyond the scope of this work.

A text generator is a system that allows a computer to communicate to a user using a human language; output could be in many forms: written text, spoken sounds, or even American Sign Language animations. No matter what the eventual output form, a text generator would need to decide exactly what to communicate to the user and how to arrange this information. The portion of a generation system that makes these initial decisions is often called a "deep generator" or "text planner" [Bateman 1996].

The Personality Tagged Logical Statistical (PeTaLS) Generator is designed to be the deep generation component for the ICICLE system. After PeTaLS has decided what instructional information to communicate to the student and how to organize this information, another module (a "surface realizer") would output the message to the student in a variety of yet-to-be-determined language forms. A fuller description of deep versus surface generation can be found in Chapter 1.

There are many aspects of the PeTaLS research that are discussed in this thesis. This paper includes a comparison between PeTaLS and other deep generation architectures (Chapter 1), the advantages of machine learning-based generation (Chapter 2), training data collection and analysis (Chapter 3), relation of PeTaLS to the rest of ICICLE (Chapter 4), system development considerations (Chapters 5 and 6), and final development issues and future directions (Chapter 7).

Chapter 1

DEEP GENERATION: TRADITIONAL METHODS AND NEW IDEAS

The best organization for the architecture of a text generator is still an open area of inquiry; however, some researchers have identified similarities in the organization of the most successful text generation systems. Broadly, most text generation systems make a distinction between "deep" and "surface" generation [McKeown 1982] [Reiter 1994] [Reiter & Dale 1997].

A deep generation component will typically decide what to say (content selection) and how to arrange that content (propositional ordering) [Bateman 1996]. A surface generator will take as input a formal structure representing the information that the computer needs to express and an organizational arrangement for this content; it is responsible for generating actual English (or another language) sentences that realize the input content [van Noord & Neuman 1996]. As mentioned previously, this surface portion of the ICICLE generation process will be handled by another component.

One question that must be answered when designing a deep generator such as PeTaLS is what the underlying formalism should be. Broadly, previous work in deep generation has ranged from schemata (e.g. [McKeown 1985], [Paris 1988], and [McCoy 1989]), which allow text patterns to be easily captured, to plans (e.g. [Moore & Swartout 1991] and [Cawsey 1993]), which facilitate reasoning about intentions behind each proposition and a user model.

While this previous work was evaluated as candidate architectures for ICICLE, several design requirements had to be considered. Most importantly, the

architecture had to operate on the user model and grammatical information available in the ICICLE domain (further discussed in Chapter 4). Another major design goal was that the workload of the human developer of the system be as small as possible.

The PeTaLS framework is designed to make the implementation of the generator for a particular domain (in this case English tutoring) be possible without the developer possessing extensive linguistic knowledge or expertise in artificial intelligence programming. Many deep generation systems that implement schemata or plan operators can be challenging to implement successfully because adding additional patterns or operators can cause unintended interactions with previous development work. By basing the system on a machine learning paradigm, much of the generator's development work is done by a learning algorithm, not a computational linguist.

Since the English writing educational curriculum of the ICICLE project and the type of interactions the system can have with students is constantly expanding, an easy-to-implement architecture would facilitate the growth of the generator over the system's development lifetime. Changes made to the content of the knowledge base of English writing grammatical information should not require modifications to the generation architecture. If more information is present in the ICICLE knowledge base, the PeTaLS system will simply include this information when relevant to the explanation of a particular writing error.

The ICICLE system has also been designed so that adding new types of propositions or new styles of text organization requires a small amount of additional development time. In a previous paper [Huenerfauth 2000], I argued in favor of a generation component that could switch among different writing styles to best suit the learning style of a particular student. A generation architecture that lessened the

workload to produce each style might encourage the development of these multiple alternate writing styles for the system's repertoire.

Planning-Based Techniques

One of the most common approaches to the deep generation problem is to convert it into an AI planning problem. Cawsey [1990, 1993] illustrates the role of plan operators in manipulating a user model and recording the system's intentions during generation. Moore and Swartout [1991] discuss how plan operators can be used to record the system's assumptions about the user's knowledge; they show how this information aids recovery from ambiguous questions.

By using hierarchical plan operators to make reasoning decisions about the user's knowledge and how to accomplish communicative goals, a scheduler algorithm can select and structure propositions to be generated. Although there are strengths to this approach, a planning-based system can be conceptually complex to design. The interactions that occur among a large set of interrelated planning operators can make them difficult to extend and maintain over time. Since planning systems often attempt to mimic high-level discourse acts that become hierarchically decomposed, creating a planning system can also require some linguistic expertise. (Because the developer would need to model discourse structures as he or she creates plan operators.) Both of these considerations make planning unattractive in light of the "ease of implementation" design goal discussed above.

A major strength of planning-based deep generators is that they can interact extensively with a user model; as planning operators are selected and executed, the user model is tested and modified. While user modeling is important to ICICLE, some specifics of the domain make planning less attractive. A common

theme in tutorial systems that use planning is that the plans decide how to motivate behaviors or beliefs in the user through communicative goals [Cawsey 1990], [1993]. In the ICICLE language-tutoring domain, the issue of motivation is less immediate; by telling a language learner that a portion of text they have submitted is ungrammatical, there is an implied motivation to fix it. The system can typically operate under the assumption that the user would like to learn to write grammatically (or else they would not be using ICICLE).

Planning-based generation systems work best with short-term goals and intentions, those that can be accomplished within a limited text span. Planning systems operate by identifying a sequence of plan operators that will accomplish a goal during each portion of generated text; in ICICLE, it is hard to identify a particular goal that must be accomplished within a specific instructional span. There are "soft" goals of increasing the user's awareness of grammatical concepts, improving her writing performance, and helping her learn to identify and correct specific language errors. It is not obvious how to modify the model of the user with each proposition that is expressed.

Because of the nature of hierarchical planning operators, planning-based tutorial systems excel at producing texts with nested digressions; however, there are indications that such texts are not desirable in ICICLE. By decomposing a series of plan operators in a planning-based generator, it is easy to create a text that will explain a tangential concept and then return to the previous topic of explanation. In the text collected during the data collection project (described in Chapter 3), no tangential explanations were encountered. Therefore, this capability of a planning system would not be required to mimic the human-written texts in the ICICLE domain.

Another advantage of a planning-based tutorial system is that when the system makes guesses about the user (because of an incomplete model) a trace of intentions and assumptions is produced in the execution of the planning operators. [Moore & Swartout 1991] When the system confuses the user or needs to backtrack, it can do so by inspecting its plan execution history. Without using planning, the PeTaLS system has found ways to address the problem of incomplete user models. A certainty-scoring mechanism is used to record the system's confidence in the need for a particular proposition being expressed. (This is discussed in detail in Chapter 5.)

Schemata-Based Techniques

Another popular deep generation architecture that is seemingly more intuitive than planning operators is the use of schemata or pattern strings [McKeown 1985]. In this approach, the developer creates a set of templates for the way propositions are ordered in a particular domain. With this fine level of text control, a developer could conceivably create templates that would mimic the texts of human authors in the target domain. So, the developer could leverage information gained from a data collection project like the one in Chapter 4. Mimicking naturally produced texts would require the developer to manually deduce patterns of typical proposition arrangements in the output genre. This task can become complex if the patterns are non-obvious, recursive, or hierarchical. Scaling up this approach to broader domains with diverse texts would also be a challenge. One way of making this pattern identification task more manageable would be to apply current advances in machine learning to make the computer do the pattern-identification automatically. The statistical natural language generation techniques discussed in the next chapter might be used for this purpose.

Unlike a planning-based system, a schemata-based one would have difficulty making deep generation decisions based on a user model. In the language acquisition domain of the ICICLE system, information about the current level of language proficiency of the user is critical [Michaud 1999], [Michaud & McCoy 2000]. While the ICICLE system may not consult or manipulate the user model as frequently as Cawsey [1990], [1993] or Moore and Swartout [1991], such interaction with the model is still necessary.

Chapter 2

APPLYING MACHINE LEARNING TO NATURAL LANGUAGE GENERATION

Traditional Applications of ML to Understanding and Generation

While the application of machine learning and statistical techniques to Natural language generation is a recent development, ML has been used successfully in Natural Language Understanding for decades. For example, one of the simplest applications has been the use of Markovian bigram and trigram models to produce highly successful part-of-speech taggers [Church 1988] [DeRose 1988] [Weischedel et al. 1993]. Some of the advantages of these statistical techniques are that they reduce the linguistic burden of the developer. Instead of forcing the developer to produce complex broad-coverage rules to handle constructions and decisions that the natural language system must make, the patterns present in a corpus of sample text can guide the system [Charniak 1993].

With the success of statistical techniques in the natural language understanding domain, it is not surprising that machine learning has recently been applied to natural language generation systems. Nearly all of this research has focused on using ML to guide the development of surface generators. Langkilde and Knight [1998], [1998b] have studied the use of statistical techniques to select between word lattices of possible surface realizations for a sentence. Bangalore and Rambow [2000] have explored techniques for incorporating tree-based syntax representations into

stochastic generators. There has also been other work which uses statistical information to allow surface generators to fill in holes in under-specified text plans [Knight & Hatzivassiloglou 1995] and to decide how to realize referring expressions [Poesio et al. 2000]. While most of this work focuses on the surface generation problem, there are elements that have been incorporated into the PeTaLS framework. Specifically, research (discussed below) into using machine learning to determine ordering of linguistic constituents relates to the propositional ordering problem.

Motivations for Using ML in PeTaLS

One of PeTaLS's design goals is to capture realistic orderings of text components without requiring extensive linguistic knowledge or effort from the human developer. Some of the disadvantages of non-statistical techniques highlighted in the previous chapter are that they require substantial manual effort to identify schemata. Finding a way for the generation system to infer patterns in the collected text samples without the guidance of the human implementer would reduce this workload.

A data collection project was conducted [Huenerfauth 2000] to collect writing samples from English-as-a-Second-Language (ESL) instructors who were asked to write one-paragraph critiques of individual sentences written by deaf students learning English. The writing of these teachers characterized the style of output desired for the ICICLE system. While the details of this data collection process will be described in more detail (Chapter 3), what is significant here is that a training set of desired output text existed in the ICICLE domain. The existence of this set made machine learning an attractive generation technique to consider.

Reversing the Machine learning Paradigm

When applying machine learning techniques to natural language generation, the typical machine learning paradigm needs to be redefined. In traditional ML, an algorithm is trained on a set of data objects, which are ordered pairs (I, C); they contain an instance and a classification. The instance is a set of properties that represents some object in the universe of discourse, and the classification is a label for a concept that the algorithm is trying to learn. For example, the instances could be the properties of specific animals, and the classifications, concepts such as "mammal," "bird," or "fish." After training is completed, the ML system will be presented with a (perhaps previously unseen) instance and will be asked to classify it.

When ML is applied to the deep generation problem, the task of the ML system is reversed. Instead of asking the system to classify an instance, we ask the ML system to generate an instance when given a desired classification. An example might help to clarify this issue; consider the set of training patterns below. Assume each ordered string of the form [A B C] represents of way of ordering the propositions {A, B, C} and that the classification of each instance is the author who likes to order her propositions in that manner.

```
Training Set = {
  ( [A B C], "Edna"),
   ([C A B], "Edna"),
     ([D A], "Edna"),
     ([B A], "Sarah"),
     ([D C], "Sarah") }
```

Figure 1: A Training Set Example

A typical ML system would be asked to label a given pattern with its author; for example, given [D A B], who would be most likely to have written her propositions in that order? When ML is applied to the deep generation problem, the situation is reversed. Given an author (say, "Edna"), the ML system would be asked to find the most likely way of ordering the propositions {A, B, D}. Since the generation problem uses machine learning information in this new direction, the application of ML techniques to generation will extend the field of ML at the same time as it elegantly solves the deep generation problem for ICICLE.

In the case of ICICLE, the system will have pieces of information it may wish to tell the student, but it needs to know how to arrange them. Using the samples of text from various teachers of English-as-a-Second-Language, a machine learning enabled system could learn how individual authors typically arrange the pieces of information they communicate. If the ICICLE generator is asked to arrange a set of information in the writing style of a particular author, it could use what it has learned to output an ordering for those propositions. Once again, the system is given a classification (the particular author), and it is asked to create an instance (the pattern of information to be expressed).

Drawing from Statistical Surface Generation Research

Although previous statistical work in Natural language generation has been focused on the surface portion of the generation process, some aspects of this work can be applied to the PeTaLS domain. Specifically, previous work that dealt with the ordering of linguistic items has immediate ties to the problem of propositional ordering. While this previous work was concerned with surface-level constructs such as noun phrase modifiers [Ratnaparkhi 2000] or Japanese bunsetsus (subcategorized

verb modifiers) [Uchimoto et al. 2000], the way in which these researchers approached the "paradigm reversal" discussed above are relevant to PeTaLS. The algorithms they used to construct orderings based on statistical data (create instances from classifications) provide a starting point for the PeTaLS framework.

Ordering Bunsetsus: The Naïve Approach

Uchimoto et al. in [2000] describe a machine learning based technique for determining the best ordering for bunsetsus in a Japanese sentence based on a corpus of naturally arising sentences. Bunsetsus are "modifiers" for verbs; every constituent in a sentence is a bunsetsu except for the verb itself. Subjects, direct objects, indirect objects, complements, and other subcategorized constituents are all bunsetsus. Since bunsetsu order in Japanese is considered "free," there is no grammatically defined ordering for these elements in the sentence. Instead, there are more or less desirable orderings that depend on the particular words in the sentence, the length of the bunsetsus, and other factors. A machine learning training algorithm was run on their Japanese text corpus to learn rules and patterns to guide their ordering algorithm.

After training, the algorithm used by Uchimoto et al. is naïve in its implementation. To determine the best ordering, the algorithm generates all possible orderings, scores each one according to the learned heuristics, and picks the highest scoring ordering for use in generation. The running time of this algorithm increases rapidly as the cardinalities of the input set of unordered constituents increases. Another limitation of their algorithm is that orderings of different lengths cannot be compared; because of the manner in which the probabilities are multiplied, arrangements of constituents with a shorter length would receive a biased score. So,

the entire process of constituent selection must be strictly completed before this ordering component can be run.

Airline Information: A More Efficient Technique

Ratnaparkhi [2000] proposed a set of trainable algorithms for the surface generation of noun phrases in the airline-scheduling domain. His algorithm is given a set of information objects of different types (destination, take-off time), and it is asked to find an ordering of adjectives and prepositional phrases that will express this information in a natural sounding manner. Ratnaparkhi had a corpus of naturally arising texts in this domain in which each of these pieces of information was tagged. The ICICLE generation problem is similar: given a set of information objects of different types, PeTaLS must find a natural sounding ordering of propositions. While Ratnaparkhi is ordering modifier phrases and PeTaLS is ordering propositions, both are using a machine learning algorithm to train on a tagged corpus of texts in the target domain.

NLG2 was the name of Ratnaparkhi's most successful learning/generation algorithm that operated on his semantically tagged corpus. The learning portion of the algorithm produced an n-gram model of the information ordering. Specifically, NLG2 used a trigram model with backoff to a bigram model in case of sparse information. After training, the generation component used a fixed-size beam search through a space of potential information orderings. The use of a Markovian model with a beam search addresses the computational efficiency problems of the Uchimoto et al. generator.

The PeTaLS ordering component implements a Markovian-based beam search similar to Ratnaparkhi's NLG2 algorithm; however, PeTaLS streamlines and

improves this algorithm in several ways. Because Ratnaparkhi was training his machine learning algorithm on more than n-gram ordering information, he needed to use a complex set of conditional predicates he called "features" to make decisions about his texts. His "features" would return truth-values in the presence of particular words, phrases, or other elements in the text. This information was used in addition to n-grams in order to guide the surface generation task. Since PeTaLS is performing deep generation using only n-grams, much of this complexity could be stripped from the NLG2 algorithm.

Like the Uchimoto et al. algorithm, Ratnaparkhi requires all the information objects in his set to be used in the final output ordering. His algorithm is incapable of comparing orderings of different lengths in an unbiased fashion. The PeTaLS system can compare orderings of unequal length, and it therefore does not require strict content selection decisions to be made before propositional ordering. (Full details about the selection and ordering components are in Chapters 5 and 6.)

Writing Personalities and Data Scarcity

An opportunity made available by the machine learning techniques which neither Uchimoto et al. nor Ratnaparkhi took advantage of is the ability to train the generation system to output text according to the writing style of a particular author. In the training sets used by these researchers, the data was not classified according to its author (as in the "Edna" and "Sarah" example). Instead, the inclusion of a piece of text in the training set gave it an implied classification of "valid"; therefore, the training set only contained positive examples of "valid" texts. If the data items were separated into different pools according to author (or simply classified according to author), then the system could be asked to produce an ordering according to the style

of a particular author. In the context of the PeTaLS system and throughout the remainder of this paper the individual style in which an author chooses to arrange the propositions in his or her writing will be called a "writing personality."

Since an ideal tutoring system could adapt its instructional style to match its user's learning style, a system designed to capture various writing personalities would have the ability to switch between them. Human instructors employ diverse and personal tutorial styles; simply blending this collected data without keeping track of individual authors would lose this valuable diversity of explanatory strategies among the teachers. The problem with considering data collected from each teacher separately is that it is harder to get a sample that is large enough to be statistically viable during the learning process. The collection and tagging of data from teachers is not an automatic process; so, the advantage provided by keeping unique writing personalities must be weighed against the amount of work needed to avoid data scarcity. Currently, PeTaLS does not separate the samples from individual teachers, but as more tutorial text is collected and tagged, the system is capable of switching to such an operating model.

Chapter 3

ACQUIRING THE TRAINING DATA CORPUS

In order to apply machine learning techniques to the problem of ordering the propositions in the ICICLE domain, a corpus of text in the desired output genre had to be collected. A data collection project was conducted [Huenerfauth 2000] to collect writing samples from English-as-a-Second-Language (ESL) instructors. The conditions under which the data was collected were meant to mimic the ICICLE domain as closely as possible. In this manner, texts were collected which served as the target output of the PeTaLS system.

Data Collection Project Overview

Graduate students with English-as-a-Second-Language tutoring experience were contacted and asked to participate in the study. The tutors were presented with a paragraph of English sentences written by students who are deaf and native to ASL; they were instructed to write explanatory text that would instruct the student how to correct the language errors they made in each sentence. The texts were solicited over e-mail so that the instructors would refrain from tutoring actions that might not be possible in the ICICLE domain (e.g. circling portions of text, using tone-of-voice, or drawing pictures). Over 500 paragraphs of explanation were collected through this project, resulting in 1357 tagged elements. (See explanation of tagging process below.)

Tagging Process

To make the data from the above project useful for creating a generation system, individual propositions or phrases were tagged. Tags encouraged abstraction from the content of the explanation texts and facilitated analysis of the structure of propositions and the relationships between them. The individual propositions in the collected explanatory texts were divided into classes according to their informational content; a tag was created for each class. From each paragraph of explanatory text, a sequence of tags was produced which represented the ordering of the propositions in that paragraph. Figure 2 demonstrates the tagging process. An explanation of each tag and the reasoning behind its creation is included in table 1.

Student Text:
It is wise to have aids test for students if aids spreads on the campus.

Instructor Text:
The noun phrase "aids test" should be preceded by the word "an."
Generally speaking, singular nouns and noun phrases require what's called
an article in front of them - usually "the", "a" or "an." "The" is
used when referring to a specific noun, whereas "a" and "an" are used
when referring to a general noun. "An" is used in this case rather than
"a" because the noun begins with a vowel.

Propositional Tagging:
corr_elab rule rule application

Figure 2: Example of the Tagging Process.

Table 1: Propositional Tag Types for ICICLE.

Name of Tag	Explanation	Example
error_class	A name of a class of errors.	Agreement Error
example	An example of correct English text.	"I was kicking the ball."
rule	A prescriptive grammar rule of written English.	The subject and the verb should agree in number.
mapped	A prescriptive grammar rule with specific references to parts of the sentence.	"Dogs" and "run" should agree in number.
correction	A specific correction to the student's text.	"I go" should be "I went."
corr_elab	A description of how the student should make a correction.	The comma following "Once" should be dropped.
definition	A definition of a grammatical term.	An acronym is an abbreviation in which each letter stands for a word in the full name.
exception_case	A caveat or exception condition to an English grammar rule.	Some irregular verbs do not form their past tense by using an –ed ending.
application	A trace of the logical steps needed to make a grammar correction.	Because the subject is first person singular and we want to talk about the past, the correct form is "was."
contrast	An explanation of an error by supposing a false condition.	If "happily" were a noun, then it would make sense to say "dog and happily." It does not.
ok	An identification that something is correct about the sentence.	This sentence is fine.
wrong	An identification that something is wrong with the sentence.	There is an error in this sentence.
semantics	An identification of bad semantics.	Dogs can't fly.
suggestion	A suggestion to change text that is not technically ungrammatical.	It might be better to use the word "surely."
<s>	A tag that indicates the beginning and the end of a paragraph of text.	

Selecting the Tags

The tags listed in table 1 were selected to capture the types of propositions present in the text samples written by English as a Second Language instructors in [Huenerfauth 2000]. During the tagging process, as additional propositions were encountered which could not be classified according to their informational content by one of the tags already present in the table, new tags were added. If additional text samples were analyzed and previously unseen proposition types were encountered, new tags could be added to ICICLE at that time. The PeTaLS architecture has been designed to be flexibly extensible in this manner.

Realizing the Tags

Each tag listed above represents a category of propositions that the ICICLE system may wish to generate. Although accomplishing this surface generation process is beyond the scope of this thesis, it is interesting to consider it here. Some of the propositional tags are very amenable to a "canned text" system. For example, "error_class" or "rule" could be easily realized by storing small pieces of text in a knowledge base entry associated with a particular grammar error. "Mapped" would be similar to "rule," except some string replacement could be performed. "Ok" and "wrong" could be realized by selecting a phrase randomly from a list of general positive or negative comments. Other tags, such as generating a correction, describing how to perform correction elaborately, or explaining the logical reasoning steps behind a decision would require more complex realization mechanisms.

During the course of development, if some tags are not yet functional in the system, the PeTaLS architecture will still operate correctly. For example, learning how to generate a proposition of tag type "semantics" may not be possible for decades.

This doesn't mean PeTaLS is inoperable until that time; ICICLE could simply give PeTaLS sets of propositions to arrange that do not include tag types that cannot yet be realized. The PeTaLS algorithms will still function correctly.

Adding Tags to the ICICLE System

If another tag needed to be added to the system to capture data on a tutorial explanation technique not listed above, expanding the system would be a straightforward process. Additional samples of text in which the tag occurs would need to be collected, these samples would need to be run through the machine learning algorithm, and a way for the ICICLE system to realize a proposition of this new tag type would need to be implemented. None of the previous tagging or development work would need to be modified in order to make the system work correctly with this new proposition type.

Chapter 4

SITUATING PETALS WITHIN ICICLE

Before We Get Started: Defining the Problem

Chapters 5 and 6 explore the PeTaLS content selection and propositional ordering components in significant detail; this analysis will be clearer if it is preceded by an overview of the ICICLE system and its components. In particular, the generator's task will be more clearly defined once its input, output, and informational resources have been specified.

Error Identification Phase

The ICICLE tutoring system initially looks like an ordinary word processor in which the student can enter an essay she is working on for English class. After pressing the "Analyze" button on the screen, the parsing and error identification component of the system takes over. A parser and a series of specially designed grammar rules are used to analyze the student's text. Included in this grammar are standard English grammar rules and special "malrules," grammar rules that capture typical errors the student may have made in her writing [Schneider & McCoy 1998].

At the end of the analysis, a parse tree for each sentence the student has written is created. The system then searches the parse tree for malrules used during the parse, since these signal that an error has been detected. The output of this phase (which is the input to PeTaLS) is a parse tree for a sentence and an error-pointer data

structure. The error-pointer contains information about the grammatical rule that has been violated, the sentence in which the error occurred, and a pointer into the parse tree of the sentence that locates the error (if appropriate).

The User Models and History

When producing the paragraph of instructional text, PeTaLS has access to more information than that which is passed to it from the error identification phase. Two types of user models and a history of the system's interaction with the user inform the decisions PeTaLS must make. One of these user models is currently under development, and the other model and the history log are still in the conceptual phase. The implementation of the PeTaLS system has helped specify the requirements of these information models. The current version of PeTaLS algorithms includes links to function calls that will access these models when they are completed; the system currently operates by disabling some of these calls or simulating the behavior of the user models and history log with a naïve implementation of their functionality.

A student using the ICICLE system will need to log into the system so that the model of her current level of language acquisition and a history log of her previous experiences with the system can be loaded. A user model would allow the system to make generation decisions based on the current level of language proficiency of the student. For example, if the system is confident that the user has mastered a particular concept, then it can concentrate on tutoring her on other English writing topics she needs to improve. Another concern is that a student may not be capable of learning a concept until she has mastered more fundamental concepts upon which it is based. By modeling the user's English writing skills, the system can decide if the user is ready to learn about a particular topic.

ICICLE will use two separate models to capture the user's English writing skills and knowledge. Michaud [1999] makes a distinction between information about the student's level of language competency and her awareness of English grammar concepts. The model used to capture this competency/performance information is called SLALOM [Michaud & McCoy 2000]; as the student demonstrates a new language skill in her text, SLALOM records this change in competency. A model of grammatical awareness would record the student's knowledge of grammatical terms and rules (independent of her ability to use them in practice). While properties of this model have been proposed [Michaud 1999], this model has not yet been implemented.

The creation of a history log of a user's previous interactions with the system is also a future development goal. By developing the deep generator prior to the history log, the internal structure of the history log has been partially determined. The history log can use the proposition data structures used by the generator to record the user's interaction with the system. This proposition structure would be more useful to the system than a plain text record of the previous text delivered to the student.

A functional history log would help the PeTaLS system make decisions about both inclusion and reference. Based on a student's recent exposure to a particular piece of information, the generator may decide to remove a proposition from the pool of possible output. Another option is to change the way the system refers to a piece of information delivered to the user; for example, instead of repeating "the subject must agree with the verb in number" in two adjacent paragraphs, the system could say "the agreement rule in the previous example." History-based inclusion considerations could easily incorporated into the PeTaLS architecture by using this information to help assign desirability scores to the propositions in the pool to be

expressed. Propositions that were recently expressed could be assigned lower scores. Michaud suggests that reference considerations may be best addressed in a revision process that would follow the deep generator [1999].

The Knowledge Base

The other major source of information for PeTaLS is the ICICLE knowledge base of information about the grammar of written English. The development of PeTaLS and the SLALOM user model have influenced the organization of this information. Because both the SLALOM model and PeTaLS deal with English grammar in terms of rules that have been violated by the user, the basic unit of the knowledge base will also be a prescriptive English grammar rule. By indexing the user model, generator, and knowledge base on the same set of rules, a potential matching problem is avoided between them.

A "rule" data structure in the knowledge base contains links to the malrules which capture violations of this rule, an English text and ASL video/animation description of the rule, examples of correct English sentences which follow the rule, exceptions to the rule, the general class of errors to which the rule belongs, and the "ideas" contained within the rule. Error classes, exceptions, examples, and ideas are other entities which exist in the knowledge base; each contain links to related entities, text or ASL descriptions, and "ideas" to which they are related. Many of these entities relate directly to proposition tag types described in the previous chapter.

"Ideas" are basic grammatical terms which are defined in the system and which are commonly used in the descriptions of other entities in the knowledge base. The ideas provide a cross-indexing capability to the information in the knowledge base

and a location to store background information about specific concepts that the user may request to see. If a description for an entity uses a concept or term, then these terms are listed in that entity's "ideas" field. During the course of the generation process, PeTaLS can keep track of all the ideas that were mentioned in the output text. The surface realizer may choose to hyperlink these terms to additional explanation text about each, or it may present the user with a list of terms in a "see also" list at the end of the tutorial paragraph.

Output of PeTaLS

PeTaLS will produce a sequence of propositions that represents the organization of the paragraph of tutorial text that should be delivered to the student. The architecture is flexible such that the internal content form of the propositions can vary. For "rule" or "example" propositions, the content may be full strings of text; for other propositions, the content could take on more abstract forms that have yet to be specified. Depending on the surface realizer that is implemented, the propositions could store semantic/syntactic information (such as the input to the FUF/SURGE system) [Elhadad & Robin 1996]. Until the final output language (English and/or ASL) and the architecture of the surface generator have been determined, providing flexible content storage inside propositions is desirable. What have been specified are type definitions for the proposition objects; they contain fields to store their tag, content, identification of the content form, and other information. The output of PeTaLS will be an ordered list of proposition objects to the surface realizer; this ordered list is referred to as a "sequence" or an "ordering" in the remainder of this thesis.

Chapter 5

THE PETALS ARCHITECTURE: SELECTION COMPONENT

What Can't We Learn?

Despite the advantages of machine learning techniques in natural language generation discussed in the previous chapters, there are some parts of the deep generation problem to which ML is not applicable. The foundation of statistical generation methods is the collection of a corpus that mimics the style of text the system should output; for machine learning to take place, the system needs access to the information the human authors had when making writing decisions so that it can learn to make the same decisions they made. Unfortunately, not all of this state information is captured in a corpus.

While we may be able to learn from a corpus how an author chose to order a selected set of information to produce a paragraph, learning how the author chose the information to include in his paragraph is not possible with a corpus alone. This "inclusion" or "content selection" decision is based on factors like the author's knowledge of English grammar, his opinions about the student's skill level, and his previous interaction with the student. In order for an ML system to learn an author's content selection style, it would need to see all of this information that the author was considering when he made his selections. These assumptions about the user and knowledge of English grammar are not recorded in the data collection project described in the previous chapter. Modifying the project to make the ESL instructors

write down all of the assumptions they made (and did not make) when deciding what to say would be extremely difficult.

Selecting a Formalism

If a machine learning system cannot accomplish the entire deep generation process, then we must consider using another technique for content selection before using ML for propositional ordering. One of the more successful formalisms in other tutorial generation systems has been planning operators. In Chapter 1, a planning system was deemed inappropriate and too complicated for ICICLE's deep generation; asking a planning system to do only half the job (just the content selection) would be even less practical. A developer would still need to write a complex planning architecture, but it would no longer be making any propositional ordering decisions. Much of the advantage of planning is that the hierarchical decomposition of the plan operators can make ordering decisions about the output.

A much simpler content selection formalism would be a rule-based system. Logical predicates could be used to create if-then conditions to decide if particular propositions should be expressed. This formalism has the advantage of being very intuitive and easy to extend; the developer will need to decide under what conditions a proposition should be expressed and then write an if-then rule to capture that decision process. While planning operators necessarily interact with one another, a series of logical condition statements can be constructed orthogonally for each proposition tag type. Adding an additional tag type to the system's repertoire would be a simple process of adding new conditional rules to decide when that proposition type should be expressed.

Since deciding absolutely whether a particular proposition should or should not be expressed may still be a challenging task, the system could instead assign probabilities to each proposition on whether it should be expressed. Instead of passing the ordering component an unordered set of propositions, the content selector would annotate each of the propositions with a "certainty" score. This score would indicate how desirable it is that each proposition appear in the final output. Certainty scores make the content selector's task easier: If the selector is only 50% certain that it would like to express a proposition, it can simply add it to the set with a 50% certainty score. The ordering component will make the final decision if the proposition will be expressed based on its ability to produce a high probability ordering. The ordering component can place a high priority on expressing propositions with high certainty scores, and use propositions with low certainty scores if they help it create an ordering which matches the machine learning data.

Selection Component Operation

The content selection algorithm modifies a data structure called a proposition "pool," which is simply an unordered set of proposition objects that have been tagged with certainty scores. At the beginning of the process, the pool contains a "start tag" and an "end tag" (each with 100% certainty scores). These tags will be realized as empty text strings, but they are later used as placeholders by the ordering component (so they are included here). An add_props function is called that contains a series of if-then conditions which reason about the user models, history log, and grammatical knowledge base to decide if a proposition of each tag type should be added to the pool and what its certainty score should be. Appendix A contains a full listing of this code.

The PeTaLS content selector considers the student's learning process and its own capabilities when making decisions. The most desirable propositions to express are those that will provide the student with information she is ready to learn, is not yet aware of, is interested in, and that will help her correct the error she made. Another concern is whether the system is capable of generating particular propositions; for example, the generator may only be able to produce a corrected version of a student's text when the error involves simple syntactic constructions. If the sentence is too complex, the system may not be able to generate a "correction" proposition and should therefore not add one to the pool. Some of the functions that the selector uses when making these decisions are listed in table 2.

Table 2: User Model and KB Access Functions.

Function	**Returns**
Getaware	Score of how aware the user is of a particular concept.
Userready	Score of whether the user is ready to learn a concept based on their observed performance of other language skills.
Canfix	True if PeTaLS can make a corrected form of the user's sentence.
Canfixe	True if PeTaLS can elaborately explain how to correct the error.
Canmap	True if PeTaLS can map words in the user's sentence to elements of the violated grammar rule.

Chapter 6

THE PETALS ARCHITECTURE: ORDERING COMPONENT

The Ordering Algorithm

The PeTaLS ordering algorithm is based loosely on the Ratnaparkhi NLG2 algorithm [2000]. PeTaLS approaches the ordering task as a beam search through the space of possible ordered sequences of the elements in the proposition pool. (The width of the beam is a constant; it is set to 200 in this implementation.) Since the algorithm also considers subsets of the pool, the final ordering that is selected may not include all of the propositions in the pool. Typically, the propositions with the highest certainty scores will be included in the output of the ordering component.

The beam search begins with a list of one sequence [START], which represents a sequence that contains only the start tag; as the search progresses, this list will include the best 200 orderings that have been encountered by the ordering algorithm so far. After iteration L of the search procedure, the system has a list of the 200 best orderings of length less than or equal to L which have been encountered. At the beginning of iteration L+1, for every ordering O in the list of 200, each of the unused propositions in the pool is appended to the end of O to produce multiple orderings of length less than or equal to L+1. After producing this large list, all of these new sequences are scored using the techniques described below, the top 200 are saved, and the search procedure proceeds to the next iteration. (The full code is listed in Appendix B.)

Scoring the Sequences

There are four types of tests that are performed on each of the sequences produced by the PeTaLS ordering component to determine their scores. The first test is one of completeness. Since a valid sequence must begin with a start tag and conclude with an end tag, sequences at the last iteration that do not satisfy this criterion receive a score of 0. The next three tests (Markovian, Value, and Brevity) are used to produce subscores in the range of 0 to 1.0 that are combined to create a composite score for each generated sequence. The PeTaLS algorithm uses weighted constants to determine the importance of each of the subscores; these weights can be modified to tune the output.

The Value and Brevity subscores compete to determine how long the winning ordered sequence will be. The Value subscore favors sequences that include more propositions; in particular, sequences that include propositions with high certainty scores receive the best Value subscores. The Value subscore is computed by calculating the quotient of the sum of the certainty scores in the sequence over the sum of the certainty scores of all the propositions in the pool. To prevent PeTaLS from simply including every proposition in the pool in the final output, a Brevity subscore is also calculated and weighted in the final decision. The Brevity subscore is calculated by taking the inverse of the length of the particular sequence.

The Markovian subscore indicates how well the ordering of a particular sequence agrees with the writing style of the text in the training corpus. A bigram model is applied to the ordered proposition tags in the corpus, and a training algorithm is used to compile probabilities for each possible pair of adjacent tags in the corpus. For example, if the "rule" tag frequently precedes the "example" tag in the training data, sequences that position an "example" tag immediately after a "rule" tag will

receive higher Markovian subscores. So, for every adjacent pair of tags in a sequence, the machine learning data can return a probability score for the likelihood of those two tags appearing next to each other in that order. To return a Markovian subscore, PeTaLS has to combine each of these adjacent pairwise probabilities; various techniques for performing this combination are considered below.

Choosing a Markovian Subscore Algorithm

An experiment was run to determine the best algorithm to combine the pairwise probabilities into a Markovian subscore. In traditional machine learning fashion, a ten-fold cross-validation test on the tagged training data was used to determine which Markovian subscoring technique could best mimic the ESL teacher text. In this way, all 400 paragraphs of collected text could be used in the training/testing during this experiment by training on 90% of the data and testing on the remaining 10% (and repeating this process 10 times).

Given an unordered set of the propositions based on a particular paragraph of the ESL text, the PeTaLS ordering component was run with each of the five candidate subscoring algorithms below to see which one would cause PeTaLS to create a paragraph most similar to the teacher's writing. The best Markovian subscoring algorithm in this experiment was used in the final implementation of PeTaLS.

When deciding upon candidate algorithms to test, there are several design goals to consider. Most basically, a set of high probability scores should combine to produce a high Markovian subscore; so, adding the probabilities or multiplying them seems intuitive. These are called the "sum(p)" and "product(p)" algorithms. However, it would also be desirable for the algorithm not to be biased in favor of longer or

shorter strings. An addition-based approach would favor longer strings, and a multiplication, shorter ones (since the probability scores will be less than 1).

Two additional subscoring techniques were added to the experiment to address this length-bias issue. The "average(p)" algorithm returns an average of the pairwise probability scores for the entire sequence; this algorithm corrects the length-bias in the "sum(p)" discussed above. To correct for length-bias in the "product(p)" algorithm above, the fourth subscoring algorithm was proposed: "product(p/avg)." This algorithm works as follows: For each adjacent pair of propositions [X Y] in a sequence, the probability value for [X Y] was retrieved from the table of results from the machine learning analysis of the corpus. Before these probabilities are multiplied, each one is divided by the average of all the probabilities in the results table. The quotients of this division should tend to be closer to 1.0 than the original probabilities. So, there will be less of a bias toward shorter strings when these quotient scores are multiplied together in the last step of this algorithm.

The fifth and final algorithm that was considered is called "swapper;" in this approach, for each pair of adjacent tags [X Y] in a sequence, the probability of [X Y] was divided by the probability of [Y X] according to the bigram model. The product of these quotients was returned as the Markovian subscore.

In order to test the selection capabilities of the PeTaLS system, two "dummy" propositions with low "value" scores were added to the proposition pool before each trial. So, if an instructor wrote a paragraph with the propositions [E F G H], the system would give PeTaLS the unordered set {E, F, G, H, D1, D2} where D1 and D2 are the dummy propositions with low "value" scores. In the way, the

interaction among the Markovian, Value, and Brevity subscores would be considered as we test each of the Markovian scoring algorithms.

A naïve manner of evaluating the PeTaLS output would be to simply record the percentage of PeTaLS orderings that were exactly identical to the ordering ESL teachers chose for that set of propositions. For example, if both PeTaLS and the ESL teacher arranged the set {E, F, G, H} in the order [E G H F], then they would have a match. However, this type of "hard" scoring is not the best representation of the success of the algorithm since it does not consider how close the output of PeTaLS was to the ESL teacher's writing in the case of a non-match.

To compensate for this imprecision, a softer scoring technique was used. The "soft scoring" graph (figure 3) shows the percentage of partial orderings which were the same between the PeTaLS output and that of the ESL teachers. For example, the pattern [A B C D] has 6 partial orderings {A<B, A<C, A<D, B<C, B<D, C<D}, and [A B C D] has 5 partial orderings in common with the pattern [B A C D], namely {A<C, A<D, B<C, B<D, C<D} but not {B<A}. So, the "soft" similarity score between these two patterns would be 83.33%. If PeTaLS returns a pattern of a different length than the ESL instructor, then the soft scoring algorithm is designed so that score will also be reduced (by using the length of both strings in the calculation of the denominator of the score).

Both "product(p)" and "product(p/avg)" had the best performance scores in this experiment. Since "product(p/avg)" includes length-bias compensation, it was incorporated into the PeTaLS system.

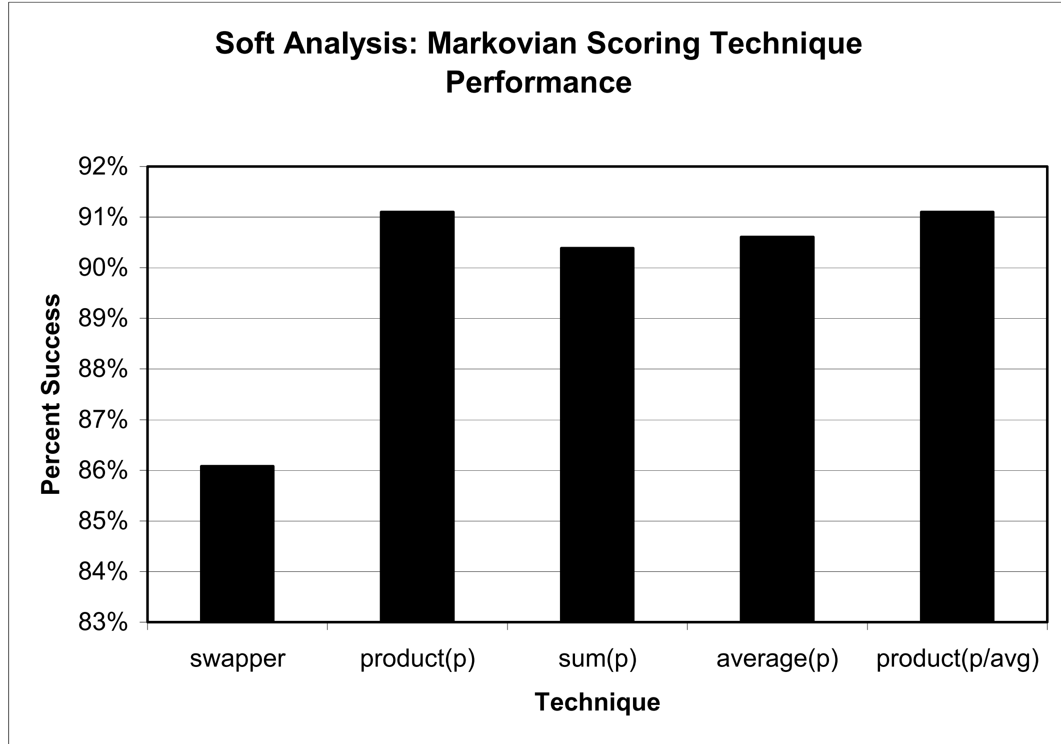

Figure 3: Performance of Markovian Subscoring Algorithms.

Cautious Results Analysis

While this experiment was useful for choosing a scoring algorithm, it is important not to read too much into these data graphs. This evaluation was executed to rank each of the Markovian subscoring algorithms, not to derive an overall performance score for PeTaLS. One reason these graphs should not be interpreted as performance scores is that two dummy propositions were added to each sequence organized by PeTaLS. By including these two propositions, the success score of PeTaLS was artificially lowered.

It would be inaccurate to use a machine learning style cross-validation test to evaluate the PeTaLS system because statistical models do not determine much of

the content selection process. As discussed in Chapter 5, the difficulty in recording all the information and assumptions used by the ESL teachers not only makes using ML training algorithms for content selection impractical, but it also eliminates the use of ML testing techniques (like this experiment) for content selection. In order to run a fair test, we would need to record all the information used by the original author in making his content selection decisions and make this information available to PeTaLS. It would be impossible to accurately simulate the state of the user models and knowledge base so that the operating conditions of PeTaLS matched that of the human author.

Another important consideration when analyzing these results is that maximum performance is not really 100%. Because multiple authors contributed to the data corpus (and because each author may not make ordering decisions consistently), the tagged corpus will not always give the same ordering for the same set of propositions. If there is not 100% consistency in the data, it is impossible for PeTaLS to receive a 100% score in this experiment unless it behaved inconsistently as well (and in the same inconsistent way that the authors happened to behave).

Chapter 7

FINAL THOUGHTS AND FUTURE DIRECTIONS

Future Development Issues

A continuing goal of the ICICLE project is to collect more training data from ESL instructors for the PeTaLS machine learning component to process. As the size of the corpus increases, more complex ML formalisms may be used to capture the writing style patterns; for example, the system could be changed from a bigram to a trigram model. Another possibility with more data is that the writing samples collected from individual authors could be separated to allow the system to learn distinct writing personalities. To supplement the collection of additional data, Shaw and Hatzivassiloglou [1999] have explored ways to extend the information gained from ML analysis of a corpus by using transitive closure and clustering. These techniques may also allow the PeTaLS project to get more information out of the current training corpus.

An issue that the surface realization component will need to resolve is how to express the concepts contained in the "ideas" list created during the generation process. (As propositions are added to the final text plan ordering, the ideas listed in each of their descriptions are appended to an ideas list.) The system may decide to rank these ideas according to how unfamiliar the user is with them and present them as a "see also" list at the end of the generated explanation text. If an HTML-style interface is used to present the text, then the ideas could also be realized as hyperlinks.

The student could click a link to read more information about a particular concept. If the surface realization component uses American Sign Language, then the presentation of these "ideas" becomes more complex. The understandability of a "see also" list or hyperlinking through the medium of ASL would need to be considered; perhaps the student could be presented with both instructional information in ASL and also with English text containing hyperlinked concepts.

The PeTaLS system currently produces a text plan that is a flat sequential ordering of propositions to be expressed. There has been much natural language generation research into the use of tree-shaped text plans [Mann & Thompson 1988] [Marcu et al. 2000]; applying these structures to PeTaLS could also be considered. There are indications that the algorithms underpinning PeTaLS could be adapted to manipulate tree structures instead of flat sequences. Ratnaparkhi also explored techniques that manipulated tree structures and described one such algorithm, NLG3 [2000]. To accommodate a change to trees, the ESL instructional corpus would need to be retagged and manually "parsed" into tree-shaped text plan structures. An open research question is how the machine learning algorithms should be modified to induce patterns from a corpus tagged with tree structures and not flat patterns.

Other research issues will arise during the development of the history log component. The results of the data collection project suggested that there are interesting patterns in how a tutor makes reference to errors a student makes several times in a particular passage. This aspect of the tutorial generation problem requires future attention, and an experiment in which tutors correct repeated-errors passages may be required for further analysis. Learning when authors make these reference

versus inclusions decisions based on the dialogue history might be another problem to which machine learning techniques can be applied.

Conclusions

This thesis has outlined the development of the Personality Tagged Logical Statistical (PeTaLS) Generator, a deep generation system successfully implemented for the ICICLE tutoring system. Using logical rules to determine content selection, certainty values to soften the selection/ordering interface, and machine learning algorithms to arrange the informational predicates, the PeTaLS system accomplishes the deep generation task in a flexible, extensible manner. While the main generation algorithms have been implemented, components that interact with PeTaLS are still under development. In order to support the generation component, a grammar knowledge base and a user model [Michaud 1999] [Michaud & McCoy 2000] will need to be completed. Following the PeTaLS component, a surface realization generator will need to convert the propositional text plan into English or American Sign Language output.

The development of PeTaLS has highlighted the advantages of leveraging machine learning research on the natural language generation problem. This work demonstrates how advances in statistical surface realization can be applied to the deep generation task. The development process has also indicated the limitations of planning and schemata based deep generation systems and the questions a developer must ask when deciding upon a formalism for a particular generation application. In particular, PeTaLS demonstrates that a deep generation system can be implemented without requiring the developer to create complex interacting plan operators or

schemata; by making ease of development a priority, PeTaLS has explored some architectural approaches never before applied to deep generation.

REFERENCES

Bahan, B., Kegl, J., Lee, R. G., MacLaughlin, D., and Neidle, C. "The Licensing of Null Arguments in American Sign Language." *Linguistic Inquiry.* Volume 21. Number 1. pages 1-27. Winter 2000.

Bangalore, S., and Rambow, O. "Exploiting a Probabilistic Hierarchical Model for Generation." In *Proceedings of the 18th International Conference on Computational Linguistics (COLING 2000)*, Saarbrucken, Germany. 2000.

Bateman, J. "Deep Generation." In *Survey of the State of the Art in Human Language Technology*, Editors Cole, R. A., Mariani, J., Uszkoriet, H., Zaenen, A., and Zue, V. Stanford University, Stanford, CA, Cambridge University Press. 1996.

Cawsey, A. "Generating Explanatory Discourse." In *Current Research in Natural Language Generation*, Editors Dale, R., Mellish, C., and Zock, M. London: Academic Press. 1990.

Cawsey, A. *Explanation and Interaction: The Computer Generation of Explanatory Dialogues*. Cambridge, MA: MIT Press. 1993.

Charniak, E. *Statistical Language Learning.* Cambridge, MA: MIT Press. 1993.

Church, K. "A stochastic parts program and noun phrase parser for unrestricted text." In *Proceedings of the Second Conference of Applied Natural Language Processing*, pages 136-143. 1988.

Conrad, R. "The reading ability of deaf school-leavers." *British Journal of Educational Psychology.* Volume 147. pages 138-148. 1977.

DeRose, S. "Grammatical category disambiguation by statistical optimization." *Computational Linguistics.* Volume 14. Number 1. pages 31-39. 1988.

DiFrancesca, S. "Academic achievement test results of a national testing program for hearing impaired students." United States: Spring, 1971. (series d, no.9), Gallaudet College, Office of Demographic Studies, Washington, DC, 1972.

Elhadad, M., and Robin, J. "An overview of SURGE: A reusable comprehensive syntactic realization component." Technical Report 96-03, Dept of Mathematics and Computer Science, Ben Gurion University, Beer Sheva, Israel. 1996.

Hovy, E. H. "Approaches to the Planning of Coherent Text." In *Natural Language in Artificial Intelligence and Computational Linguistics*, Editors Paris, C. L., Swartout, W. R., and Mann W. C. Boston: Kluwer Academic Publishers. 1990.

Huenerfauth, M. "PDPS: Personality-Driven Plan Schemata Architecture for Tutorial Generation." Unpublished. May 2000.

Kibble, R. and Power, R. "An Integrated Framework for text planning and pronominalization." In *Proceedings of International Natural Language Generation Conference (INLG2000)*, Mitzpe Ramon, Israel, June 2000.

Knight, K. and Hatzivassiloglou, V. "Two-level, Many-Paths Generation." In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL'95)*, pages 252-260, Cambridge, Massachusetts. 1995.

Langkilde, I., and Knight, K. "Generation that Exploits Corpus-Based Statistical Knowledge." In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (COLING-ACL 1998)*, pages 704-710, Montreal, Canada. 1998.

Langkilde, I., and Knight, K. "The Practical Value of n-grams in generation." In *Proceedings of the 9th International Natural Language Workshop (INLG '98)*, Niagara-on-the-Lake, Ontario. 1998.

Mann, W. C., and Thompson, S. A. "Rhetorical Structure Theory: Toward a Functional Theory of Text Organization." *Text*. Volume 8. 1988.

Marcu, D., Carlson, L., and Watanabe, M. "The Automatic Translation of Discourse Structures." In *Proceedings of the 6th Applied Natural Language Processing Conference and the 1st Meeting of the North American Chapter of the Association of Computational Linguistics (ANLP-NAACL 2000)*, Seattle, Washington, USA. 2000.

McCoy, K. F. "Generating Context Sensitive Responses to Object-Related Misconceptions." *Artificial Intelligence*. Volume 41. pages 157-195. 1989.

McCoy, K. F., and Masterman, L. N. "A Tutor for Teaching English as a Second Language for Deaf Users of American Sign Language." In *Proceedings of Natural Language Processing for Communication Aids, an ACL/EACL'97 Workshop*, pages 47-54, Madrid, Spain, July 12th, 1997.

McCoy, K. F., Pennington, C., and Suri, L. Z. "English Error Correction: A Syntactic User Model Based on Principled 'Mal-Rule' Scoring." In *Proceedings of the Fifth International Conference on User Modeling*, pages.59-66, Kailua-Kona, Hawaii, January 1996.

McKeown, K. R. "The TEXT system for natural language generation: An overview." In *Proceedings of the 20th Annual Meeting of the Association of Computational Linguistics*, pages 113-120, Toronto, Canada, June 1982.

McKeown, K. R. "Discourse Strategies for Generating Natural-Language Text." AI. North Holland. Volume 27. Number 1. pages 1-41. 1985.

Michaud, L. N. "Toward a Morphosyntactic User Model for Language Analysis and Generation: A Ph.D. Thesis Proposal." Department of Computer and Information Sciences, University of Delaware. 1999.

Michaud, L. N., and McCoy, K. F. "Supporting Intelligent Tutoring in CALL by Modelling the User's Grammar." In *Proceedings of the 13th International FLAIRS Conference, Florida Artificial Intelligence Research Society Annual Conference*, Orlando, Florida, May 2000.

Michaud, L. N., McCoy, K. F., and Pennington, C. "An Intelligent Tutoring System for Deaf Learners of Written English." In *Proceedings of ASSETS'00*, Arlington, VA, November 2000.

Moore, J. D., and Swartout, W. R. "A Reactive Approach to Explanation: Taking the User's Feedback into Account." In *Natural Language in Artificial Intelligence and Computational Linguistics*, Editors Paris, C. L., Swartout, W. R., and Mann W. C. Boston: Kluwer Academic Publishers. 1991.

Paris, C. L. "Tailoring Object Descriptions to a User's Level of Expertise." *Computational Linguistics*. Volume 14. Number 3. pages 64-78. September 1988.

Poesio, M., Henschel, R., and Kibble, R. "Statistical NP Generation: A First Report." In *Proceedings of the ESSLLI Workshop on NP Generation*, Utrecht, August 1999. 2000.

Ratnaparkhi, A. "Trainable Methods for Surface Natural language generation." In *Proceedings of the 6th Applied Natural Language Processing Conference and the 1st Meeting of the North American Chapter of the Association of Computational Linguistics (ANLP-NAACL 2000)*, Seattle, Washington, USA. 2000.

Reiter, E. "Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible?" In *Proceedings of the 7th International Workshop on Natural Language Generation*, pages 163-170. 1994.

Reiter, E., and Dale, R. "Building Applied Natural Language Generation Systems." *Natural Language Engineering*. Volume 3. Part 1. pages 57-87. 1997.

Schneider, D., and McCoy, K. "Recognizing Syntactic Errors in the Writing of Second Language Learners." In *Proceedings of Coling-ACL-98*, pages 1198-1204, Universite de Montreal, Montreal, Quebec, Canada, August 10-14, 1998.

Shaw, J., and Hatzivassiloglou, V. "Ordering Among Premodifiers." In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL-99)*, pages 135-143, College Park, Maryland. 1999.

Strong, M. *Language Learning and Deafness*. New York: Cambridge University Press. 1988.

Uchimoto, K., Murata, M., Ma, Q., Sekine, S., and Isahara, H. "Word Order Acquisition from Corpora." *Journal of Natural Language Processing*. Volume 7. Number 4. pages 163-180. 2000.

van Noord, G., and Neuman, G. "Syntactic Generation." In *Survey of the State of the Art in Human Language Technology*, Standford University, Stanford, CA, Cambridge University Press, Eds. Cole, R. A., Mariani, J., Uszkoriet, H., Zaenen, A., and Zue, V. 1996.

Weischedel, R. M., Meteer, M., Schwartz, R., Ramshaw, L., and Palmucci, J. "Coping with ambiguity and unknown words through probabilistic models." *Computational Linguistics*. Volume 19. Number 2 (special issue on using corpora). 1993.

Wrightstone, J. W., Aronow, M. S., and Moskowitz, S. "Developing reading test norms for deaf children." *American Annals of the Deaf*. Volume 108. pages 311-316. 1963.

Woolf, B., and McDonald, D. "Building a Computer Tutor: Design Issues." *IEEE Computer*. Volume 17. Number 9. pages 60-75. September 1985.

Appendix A

SELECTION COMPONENT CODE

```
;;
;; PeTaLS Selection Component
;; Definition of the KB types.
;; Matt Huenerfauth
;;

;;
;; To Generate Text:
;; (generate-explanation err_ptr)
;; Input: err_ptr = Pointer to the error in the parse
;;                  tree which you would like to explain.
;; Output: A string of text which explains the error.
;;

;; global prop list
(setq *props* '())

;; global idea link list
(setq *ideas* '())

;; random value used in the decision process
(setq *random* 0)

;; Randomizes the *random* value
(defun new-random ()
  (setq *random* (random 10))
)

;; id data structure
;;
;; Anything with a KB id can be looked up using the inkb
;; function or can be made aware to the user
;;
;; Used like pointers in this system.
;;
(defstruct id
  type ; Is this a rule, ec, caveat, idea, etc...
```

```
  name ; what is the textual name for this item
  )

;; rule data structure
(defstruct rule
  id            ; KB id
  malrulecode   ; codename for malrule
  text          ; text of the rule for generator
  example       ; example of correct usage
  exceptions    ; caveats to the rule
  errorclass    ; errorclass of the rule
  ideas         ; ideas contained in this rule
)

;; ec 'errorclass' data structure
(defstruct ec
  id    ; KB id
  text  ; name of the error class for generator
  ideas ; ideasin the error class
)

;; caveat data structure
(defstruct caveat
  id       ; KB id
  casetext ; text for the case of the caveat
  example  ; example of this usage
)

;; idea data structure
(defstruct idea
  id    ; KB id
  name  ; name of this idea
  text  ; text of this idea explanation
  ideas ; related ideas
)

;; prop data structure
(defstruct prop
  id        ; KB id - used for history purposes
  tag       ; Machine learning tag type
  reallevel ; realization: text, functional spec, etc.
  content   ; the text or SURGE input
  value     ; used in the ordering algorithm
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;                                              ;;
;;  Prop adding function.  Contains the plans. ;;
;;                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun addprops (R E)

;; Plan:    ERROR-CLASS

  (cond ((and (inkb (rule-errorclass R))
	     (userready (rule-errorclass R)))

      (add 'error_class
	     (ec-text (getkb (rule-errorclass R)))
	     0.8 'text)
      (moreaware (rule-errorclass R))
      (addideas  (ec-ideas (getkb (rule-errorclass R))))
      ))

;; Plan:    ERROR-CLASS

 (cond ((and (inkb (rule-errorclass R))
	     (not (userready (rule-errorclass R))))

     (add 'error_class
	     (ec-text (getkb (rule-errorclass R)))
	     0.5 'text)
     (moreaware (rule-errorclass R))
     (addideas  (ec-ideas (getkb (rule-errorclass R))))
     ))

;; Plan:    UNKNOWN-CLASS

 (cond ((not (inkb (rule-errorclass R)))

     (add 'error_class
	     "There is an error in this text. " 0.1 'text)
     ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Plan:    RULE

 (cond ((and (inkb (rule-id R))
	     (userready (rile-id R)))
```

```
     (add 'rule (rule-text R) 0.8 'text)
     (moreaware (rule-id R))
     (addideas  (rule-ideas R))
     ))

;; Plan:    RULE

 (cond ((and (inkb (rule-id R))
          (not (userready (rule-id R))))

     (add 'rule (rule-text R) 0.5 'text)
     (moreaware (rule-id R))
     (addideas  (rule-ideas R))
     ))

;; Plan:    UNKNOWN-RULE

 (cond ((not (inkb (rule-id R)))

     (add 'rule "This sentence is ungrammatical. "
          0.1 'text)
     ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Plan:    MAPPED-RULE

 (cond ((and (inkb (rule-id R))
          (userready (rule-id R))
          (aware (rule-id R))
          (canmap R E))

     (add 'mapped (domap R E) 1.0 'text)
     (addideas  (rule-ideas R))
     ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Plan:    CORRECTION

 (cond ((and (canfix R E)
          (not (canfixe R E)))

     (add 'correction (dofix R E) 1.0 'text)
     ))
```

```
;; Plan:    ELAB-CORRECTION

 (cond ((and (canfixe R E))

     (add 'corr_elab (dofixe R E) 1.0 'text)
     ))

;; Plan:    UNKNOWN-CORRECTION

 (cond ((and (not (canfix R E))
          (not (canfixe R E)))

     (add 'correction "This text should be corrected. "
          0.1 'text)
     ))

;; Plan:    EXAMPLE-CORRECT

 (cond ((inkb (rule-example R))

     (add 'example (rule-example R) 0.8 'text)
     (moreaware (rule-id R))
     ))

;;
;; Still to be implemented:
;;  exception_case
;;  application
;;  contrast
;;

) ; end of defun addprops



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; POOL MANIPULATION FUNCTIONS
;;

;;
;; This will add entries to the global props pool
;;
(defun add (newtag newcontent newvalue newreallevel)
```

```
  (setq *props*
     (cons (make-prop :id nil
                 :tag newtag
                 :content newcontent
                 :reallevel newreallevel
                 :value newvalue)
           *props*))
  t)

;;
;; Adds the ideas in this entry to the global ideas pool
;;
;; Should check for uniqueness!!!
;;
(defun addideas (e)
  (if (not (exist-idea e))
      (setq *ideas* (cons e *ideas*))
    )
)

;;
;; Check if idea is already in the list.
;;
(defun exist-idea (e)
  (member e *ideas* :test #'equal)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; REALIZATION CODE
;;

;;
;; Realizes the pool of propositions into a string.
;;
(defun realize (seq)
  (cond ((null seq)
      "")
     (t
      (concat (realize-item (car seq))
           (realize (cdr seq))))))

;;
;; Realization helper function
;;
```

```
(defun realize-item (item)
  (cond ((equal (prop-reallevel item)
             'text)
       (prop-content item))
      ((equal (peop-reallevel item)
             'starttag)
       "BEGINNING OF EXPLANATION: ")
      (t
       "Error: Unable to process this proposition. ")))

;;
;; Handles presenting the ideas
;;
(defun handle-ideas (ilist)
  (pprint ilist)
)

;; Decides if the system can fix this error.
;;
;; For testing, we'll use random number comparison.
;;
(defun canfix (r e)
  (< 8 *random*)
)

;; Decides if the system can elaborately fix the error.
(defun canfixe (r e)
  (< 2 *random*)
)

;; Performs the fix.
(defun dofix (r e)
  "[a corrected version of the sentence] "
)

;; Performs the elaborate fix.
(defun dofixe (r e)
  "[ an elaborate description of how to correct error] "
)

;; Decides if system can map rule items to the sentence.
(defun canmap (r e)
  (< 3 *random*)
)

;;Performs the mapping from words in sentence to rule.
```

```
(defun domap (r e)
  "[ a mapped version of the rule. ]"
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; KNOWLEDGE BASE CODE
;;

;;
;; For now we will implement the KB as an alist
;;
(setq *kb* '())

;;
;; Add an entry into the KB
;;
(defun kb-add (id item)
  (setq *kb* (cons (cons id item)
             *kb*)))


;;
;; Decides if an entry is present in the KB.
;;
(defun inkb (e)
  (cdr (assoc e *kb*))
)

;;
;; Gets an entry from the KB
;;
(defun getkb (e)
  (cdr (assoc e *kb*))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; USER MODEL CODE
;;

;;
;; For now we will implement awareness as alist
;;
```

```
(setq *aware* '())

;;
;; Gets awareness score for an id#
;;
(defun getaware (id)
  (cond ((null (cdr (assoc id *aware*)))
      (setq *aware* (cons (cons id 0) *aware*))
      0)
     (t
      (cdr (assoc id *aware*)))))

;;
;; Makes the user more aware of this entry.
;;
;; For now, increments the awareness count.
;;
(defun moreaware (id)
  (setq *temp-aware* (getaware id))
  (setq (cdr (assoc id *aware*))
     (+ *temp-aware* 1))
)

;;
;; Decides if the user is aware of the entity
;;
;; For now, we define this as having heard something
;; once.
;;
(defun aware (id)
  (> (cdr (assoc id *aware*)) 0)
)

;; Decides if the user is ready to hear this rule.
;;
;; Real version will query the SLALOM model.
;;
(defun userready (e) t)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; MALRULE to KB matching
;;

;; Use an alist
```

```
(setq *malrule2kb* '())

;; Adds lookup entry
(defun add-malrule2kb (malrule-num id-num)
  (cond ((null (cdr (assoc malrule-num *malrule2kb*)))
      (setq *malrule2kb* (cons (cons malrule-num id-num)
                      *malrule2kb*)))
     (t
      (setq (cdr (assoc malrule-num *malrule2kb*))
            id-num))))

;; Gets the rule id# for a particular rule
;;
;; This should return something generic in case
;; of failure to match: 0?
;;
(defun get-id-from-malrule (m)
  (let ((codename (errorptr-codename m)))
    (cond ((null (cdr (assoc codename *malrule2kb*)))
        0) ;; FAILURE TO MATCH
       (t
        (cdr (assoc codename *malrule2kb*)))
    )
  )
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; MAIN SELECTION FUNCTION
;;

(defun select-props (ptr2error)
  (let ((the-rule (getkb (get-id-from-malrule ptr2error)))
     (the-error ptr2error))

    (setq *props* '())
    (setq *ideas* '())
    ;(new-random)

    (addprops the-rule the-error)

  )
)
```

```
;;
;; MAIN GENERATION ALGORITHM
;;
(defun generate-explanation (ptr2error)
  (select-props ptr2error)
  (realize (getorder *props*))
)

;;
;; Testing algorithm
;;
(defun test-generate (ptr2error)
  (format nil (generate-explanation ptr2error))
  (handle-ideas *ideas*)
)



;;
;; Displays the ideas to the screen...
;;
;; For debugging, we'll just print these.
;;
(defun handle-ideas (lst)
  (print "OTHER TOPICS:")
  (print-ideas (sort-ideas (lst)))
)

(defun print-ideas (lst)
  (cond ((null lst)
        (print "--------------"))
       (t
        (print (idea-text (car (lst))))
        (print-ideas (cdr lst)))
))

(defun sort-ideas (lst)
  (sort lst #'a-<-b-ideas))

(defun a-<-b-ideas (a b)
  (< (getaware (idea-id a))
     (getaware (idea-id b))))


;
; descriptive error pointer data structure.
```

```
; this is defined inside the ICICLE system.
;
;(defstruct errorptr
;    origin        ; parent node where error occurs
;    errorfeat     ; *unif-flex* index or 0 for malrules
;    mycat         ; cat of origin
;    myfeat        ; features of origin
;    left          ; left index in text of this span
;    right         ; right index in text of this span
;    errorrule     ; rule used to make origin node
;    childnums     ; nums of children of origin node
;    childcats     ; cats of children
;    childfeats    ; feature lists of children
;    specprob      ; detailed information about error
;    codename      ; error code, malrules start with m
;    description)  ; text description of error
```

Appendix B

ORDERING COMPONENT CODE

```
;;
;; PeTaLS Ordering Component
;; Matt Huenerfauth
;;

;;
;; Function to Call:  (getorder props)
;; Input: props = unordered set of propositions
;; Output: ordered set of propositions
;;

;;
;; For debugging purposes, we'll load these files here.
;; Selection will really be loaded later, debug is just
;; for testing purposes.
;;
(load "selection.lisp")
(load "debug.lisp")

;;
;; The pool of propositions.
;; This is set during execution of getorder.
;;
(setq *POOL* '())

;;
;; The first proposition in the sequence.
;;
(setf *STARTTAG*
      (make-prop :id nil
                 :tag '<s>
                 :reallevel 'starttag
                 :content nil
                 :value 0
))

;;
```

```
;; The last proposition in the sequence.
;;
(setf *ENDTAG*
      (make-prop :id nil
                 :tag '<s>
                 :reallevel 'endtag
                 :content nil
                 :value 0
))

;;
;; Score weight constants.  These should be relative.
;;
(setq *MARK* 2) ;; Markov Model
(setq *VALU* 2) ;; Value Usage
(setq *BREV* 1) ;; Brevity
(setq *LENG* 0) ;; favor longer strings (Debugging.)


;;
;; Makes the best N sequences of length i using
;; values in the list a
;;
(defun makeseq (n i a)
  (cond ((equal i 1)
      (list (list *endtag*)))
     (t
      (top n (next (makeseq n (- i 1) a) a)))))

;;
;; Makes all the possible #(a) * #(seqlist) sequences
;; that could be created from the current seqlist
;; We append new tags to the beginning of previously
;; created sequences (as long as they don't already begin
;; with the starttag.
;;
(defun next (seqlist a)
  (setq L seqlist)
  (loop for s in seqlist do
    (if (not (equal (prop-reallevel (car s)) 'starttag))
        (loop for v in (unused s a) do
       (setq L (append L (list (cons v s)))))
    ) ;endif
   )  ;endloop
 L)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; List functions.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; Removes all the items in list a from list b
;;
(defun unused (a b)
  (cond ((null b) (list nil))
     ((null a) b)
     ((atom a)
      (unused (cdr a) (remove a b)))
     (t
      (unused (cdr a) (remove (car a) b)))))

;;
;; Removes the single item from the List
;;
(defun remove (item List)
  (cond ((null item) List)
     ((null List) nil)
     ((equal item (car List))
      (cdr List))
     (t
      (cons (car List) (remove item (cdr List))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Scoring subfunctions.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; Returns the markovian portion of the score.
;; Note recursion.
;;
;; Product:  Prob of i j  /  Prob of j i
;;
(defun markov-score-0 (seq)
  (cond ((twoleft seq)
      (*
         (/ (aref *probabilities*
               (geti seq) (getiplus1 seq))
            (aref *probabilities*
```

```
                (getiplus1 seq) (geti seq)))
          (markov-score-0 (cdr seq))))
      (t
       1)))



;;
;; Returns the markovian portion of the score.
;; Note recursion.
;;
;; Product:  Prob of i j
;;
(defun markov-score-1 (seq)
  (cond ((twoleft seq)
       (*
          (aref *probabilities* (geti seq) (getiplus1 seq))
          (markov-score-1 (cdr seq))))
      (t
       1)))


;;
;; Returns the markovian portion of the score.
;; Note recursion.
;;
;; Sum:  Prob of i j
;;
(defun markov-score-2 (seq)
  (cond ((twoleft seq)
       (+
          (aref *probabilities* (geti seq) (getiplus1 seq))
          (markov-score-2 (cdr seq))))
      (t
       1)))


;;
;; Returns the markovian portion of the score.
;; Note recursion.
;;
;; AVG:  Prob of i j
;;
(defun markov-score-3 (seq)
  (let ((seq-length (length seq)))
    (cond ((twoleft seq)
```

```
        (/ (+ (aref *probabilities*
                    (geti seq)
                    (getiplus1 seq))
              (* (markov-score-3 (cdr seq))
                 (- seq-length 1)))
             seq-length)
        )
       (t
        1))
    )
  )


;;
;; Returns the markovian portion of the score.
;;
;; Prob of i j / AVG of all Probs
;;
(defun markov-score-4 (seq)
  (cond ((twoleft seq)
      (*
         (/ (aref *probabilities*
                 (geti seq) (getiplus1 seq))
            *avg-probabilities*
            )
         (markov-score-4 (cdr seq))))
     (t
      1)))


;;
;; The average of the probability functions is needed for
;; the calculation of markov-score-4
;;
(defun calc-avg-probabilities ()
  (setq *avg-probabilities*
     (get-average
      (loop for x from 0 to 15 collect
        (loop for y from 0 to 15 collect
          (aref *probabilities* x y))))))


;;
;;flatten a list so that all atoms are at the top level
;;
(defun flatten (lyst)
```

```
  (cond ((null lyst) ())
     ((atom lyst) (list lyst))
     ((and (listp lyst)
           (listp (car lyst)))
      (append (flatten (car lyst)) (flatten (cdr lyst))))
     ((listp lyst)
      (cons (car lyst) (flatten (cdr lyst))))
     (t (format t "Error, shouldn't ever get here")
        (cons (flatten (car lyst)) (flatten (cdr lyst))))
))

;;
;; Returns the average of the items in the list
;;
(defun get-average (lst)
  (/ (sum (flatten lst))
     (length (flatten lst))))

;;
;; Returns the sum of items in list
;;
(defun sum (lst)
  (cond ((null lst) 0)
     (t
      (+ (car lst)
         (sum (cdr lst))))))

;;
;; Flattens, and then...
;; Returns the sum of the items in the list
;;
(defun get-sum (lst)
  (cond ((null lst) 0)
     ((atom (car lst))
      (+ (car lst) (get-sum (cdr lst))))
     (t
      (+ (get-sum (car lst)) (get-sum (cdr lst))))))

;;
;; function for switching between markov score functions
;;
(defun set-markov (num)
  (cond ((equal num 0)
      (setf markov-score #'markov-score-0))
     ((equal num 1)
      (setf markov-score #'markov-score-1))
```

```
     ((equal num 2)
      (setf markov-score #'markov-score-2))
     ((equal num 3)
      (setf markov-score #'markov-score-3))
     ((equal num 4)
      (calc-avg-probabilities)
      (setf markov-score #'markov-score-4))
))

;;
;; Sets the active markov-score function
;;
(set-markov 4)

;;
;; Returns true if at least two items left in the list
;;
(defun twoleft (L)
  (cond ((null L) nil)
     ((atom L) nil)
     ((null (cadr L)) nil)
     (t t)))

;;
;; Returns the value portion of the score
;;
(defun value-score (seq)
     (/ (sumval seq)
     (sumval *POOL*)))

;;
;; Helper function to the value-score function
;;
(defun sumval (seq)
  (reduce '+ (mapcar #'prop-value seq)))

;;
;; Returns the brevity portion of the score
;;
(defun brevity-score (seq)
  (/ 1
     (length seq)))

;;
;; Returns a length score. (Inverse of brevity.)
;;
```

```
(defun length-score (seq)
  (length seq))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Scoring / Ranking Functions
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; Returns the n items in the list with the highest score
;;
(defun top (n seqlist)
  (mapcar #'remscore
       (first0 n
           (sortheads (mapcar #'addscore
                        seqlist)))))

;;
;; Adds a score to the head of the list
;;
(defun addscore (item)
  (cons (score item) item))

;;
;; Removes the score at the head of the list
;;
(defun remscore (item)
  (cdr item))

;;
;; Uses constants to weight the portions of the score
;;
(defun score (seq)
  (+ (* *MARK* (funcall markov-score seq))
     (* *VALU* (value-score seq))
     (* *BREV* (brevity-score seq))
     (* *LENG* (length-score seq))
))

;;
;; We need separate funcitons for scoring/ranking at the
;; toplevel since we need to do a validity check here.
;; Specifically, we give a score of 0 to orderings that
;; don't start with the *starttag* or end with *endtag*
;;
```

```
;;
;; toplevel- Returns n items in list with highest score
;;
(defun toptop (n seqlist)
  (mapcar #'remscore
       (first0 n
           (sortheads (mapcar #'topaddscore
                        seqlist)))))

;;
;; toplevel- Adds a score to the head of the list
;;
(defun topaddscore (item)
  (cons (topscore item) item))

;;
;; toplevel- Score for the top level. Checks completeness.
;;
(defun topscore (seq)
  (cond ((not (equal (prop-reallevel (car seq))
                'starttag)) 0)
     (t
      (+ (* *MARK* (funcall markov-score seq))
         (* *VALU* (value-score seq))
         (* *BREV* (brevity-score seq))
         (* *LENG* (length-score seq))
))))

;;
;; Returns a list of the first n items of the List
;;
(defun first0 (n List)
  (cond ((null List) nil)
        ((equal n 0) nil)
     (t
      (cons (car List) (first0 (- n 1) (cdr List))))))

;;
;; Sorts the list according to the values stored at heads
;;
(defun sortheads (List)
  (sort List #'headmore))

;;
;; Returns true when head of a is greater than head of b
```

```
;;
(defun headmore (a b)
  (> (car a) (car b)))

;;
;; Get the tag of the ith entry in the seq
;;
(defun geti (seq)
  (cdr (assoc (prop-tag (cadr seq)) *w2c*)))

;;
;; Get the tag of the i+1st entry of the seq
;;
(defun getiplus1 (seq)
  (cdr (assoc (prop-tag (car seq)) *w2c*)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Functions to access the probability model.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; Divides the occurrence scores in the probabilities
;; array with the corresponding row sum to create
;; true probabilitiy values.
;;
(defun normalize-probs ()
  (setf *rsum* (make-array (list *psize*)
                :initial-element 0.0001))
  (loop for x from 1 to (- *psize* 1) do
    (loop for y from 1 to (- *psize* 1) do
      (setf (aref *rsum* x)
         (+ (aref *rsum* x)
            (aref *probabilities* x y)))))
  (loop for x from 0 to 15 do
    (loop for y from 0 to 15 do
      (setf (aref *probabilities* x y)
         (/ (aref *probabilities* x y)
            (aref *rsum* x)))
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;; The main function called by the system.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; The main generation method
;;
(defun getorder (List)
  (setq *POOL* (cons *starttag* List))
;  (car (makeseq 200 (+ (length *pool*) 1) *pool*))
  (car (toptop 200
           (next (makeseq 200 (+ (length *pool*) 1)
                    *pool*) *pool*))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; currently unused
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; Makes a list from 1 to N
;;
(defun from1to (n)
  (cond ((equal n 1) 1)
     (t
      (cons (from1to (- n 1)) (list n)))))

;;
;; Counts the number of non-<s> tags in the seq
;;
(defun non-<s> (seq)
  (cond ((null seq) 0)
     ((atom seq)
      (if (equal (prop-tag seq) '<s>)
          0
          1))
      ((equal (prop-tag (car seq)) '<s>)
       (non-<s> (cdr seq)))
      (t
       (+ 1 (non-<s> (cdr seq))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

```
;; Testing helper functions.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; Converts from prop list to list of tags
;;
(defun make-simple (seq)
  (mapcar #'prop-tag seq))

;;
;; Decides if the particular ordering will match the
;; original order
;;
(defun made-right (seq)
   (setq *old-seq* (strip-ss seq))
   (setq *new-seq* (strip-ss
              (make-simple
               (getorder
                (make-props *old-seq*)))))
   (cond ((equal *old-seq*
           *new-seq*)
       1)
      (t
       (* *SOFT-WEIGHT*
          (soft-score (frame-ss *old-seq*)
                (frame-ss *new-seq*)))))
)

;;
;; Wraps a seq with a <s> on either end.
;;
(defun frame-ss (inseq)
  (append '(<s>)
       inseq
       '(<s>))
)

;;
;; Makes a list of 1s and 0s to represent which seqs match
;; to their correct ordering
;;
(defun make-sum-list (seqs)
  (mapcar #'made-right seqs))

;;
```

```
;; Returns count of lists that return the correct ordering
;;
(defun sum-all (lists)
  (sum-help (make-sum-list lists)))

;;
;; Adds all the 1s and 0s in the list
;;
(defun sum-help (value-list)
  (cond ((null value-list) 0)
     (t
      (+ (car value-list)
         (sum-help (cdr value-list))))))

;;
;; Removes <s> from the seq
;;
(defun strip-ss (seq)
  (cond ((null seq) nil)
     ((equal (car seq) '<s>)
      (strip-ss (cdr seq)))
     (t
      (cons (car seq)
            (strip-ss (cdr seq))))))

;;
;; Returns the percent of all the data items which
;; return the correct ordering
;;
(defun run-test (input-data)
  (/ (sum-all input-data)
     (length input-data)))

;;
;; Sets whether dummy propositions will be added to the
;; set for testing purposes.
;;
(defun set-dummy-props (onoff)
  (cond (onoff
      (setq *VALU* 2)
      (setq *BREV* 1)
      (setq *dummy-props* *two-dummies*)
         )
     (t
      (setq *VALU* 100)
      (setq *BREV* 0)
```

```
      (setq *dummy-props* nil)
      )
))

;;
;; Initial Value for *two-dummies*
;;
(setq *two-dummies*
      (list (make-prop :id         nil
                 :tag        'rule
                 :reallevel  'text
                 :content    "Dummy Prop: rule. "
                 :value      0.5)
         (make-prop :id         nil
                 :tag        'mapped
                 :reallevel  'text
                 :content    "Dummy Prop: mapped. "
                 :value      0.5)
      )
)

;;
;; Initial Value for *dummy-props*
;;
(setq *dummy-props* *two-dummies*)

;;
;; Makes a dummy prop out of the tag
;;
(defun make-this-prop (tg)
  (make-prop :id         nil
             :tag        tg
             :reallevel  'text
             :content    tg
          :value      1))

;;
;; Changes the list of tags into the list of props
;;
(defun make-props (plist)
  (append
   (mapcar #'make-this-prop plist)
   *dummy-props*
   )
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Data Manipulation Functions
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;
;; Sets the current testing and training sets
;;
(defun set-training-testing (input-data index)
  (setq *train-test* (make-sets input-data index nil nil))
  (setq *training* (car *train-test*))
  (setq *testing*  (cadr *train-test*))
)


;;
;; Breaks the data into training and testing sets
;;
(defun make-sets (whole index training testing)
  (cond ((null whole)
      ;; end case, return training and testing sets
      (list training testing)
      )
     ((> index 1)
      ;; Add to training set 9 out of 10 times
      (make-sets (cdr whole)
              (- index 1)
              (cons (car whole) training)
              testing)
      )
     (t
      ;; Add to testing set every tenth
      (make-sets (cdr whole)
              10
              training
              (cons (car whole) testing))
      )
))


;;
;; Re-train: Main Function Call
```

```
;;
(defun retrain ()

  ;; Define Vocabulary
  (setf *w2c* '(
     (<s> . 1)
     (application . 2)
     (contrast . 3)
     (corr_elab . 4)
     (correction . 5)
     (definition . 6)
     (error_class . 7)
     (example . 8)
     (exception_case . 9)
     (mapped . 10)
     (ok . 11)
     (rule . 12)
     (semantics . 13)
     (suggestion . 14)
     (wrong . 15) ))

  ;; Probabilities Array...
  (setf *psize* 16)
  (setf *probabilities*
     (make-array '(16 16) :initial-element 0.000001))

  (retrain-outer *training*)
  (normalize-probs)
  (calc-avg-probabilities)
)

;;
;; Retrain: Helper function 1
;;
(defun retrain-outer (seqs)
  (cond ((null seqs) t)
     (t
      (retrain-inner (car seqs))
      (retrain-outer (cdr seqs))
        )
))

;;
;; Retrain: Helper function 2
;;
(defun retrain-inner (seq)
```

```
  (cond ((twoleft seq)
      ;; Change all the 0.000001 to 0
      (if (< (aref *probabilities*
                (cdr (assoc (car seq)  *w2c*))
                (cdr (assoc (cadr seq) *w2c*)))
          1)
             (setf (aref *probabilities*
                (cdr (assoc (car seq)  *w2c*))
                (cdr (assoc (cadr seq) *w2c*)))
             0)
         )
      ;; Increment the value
      (incf (aref *probabilities*
               (cdr (assoc (car seq)   *w2c*))
               (cdr (assoc (cadr seq)  *w2c*))
               )
         )
      ;; look at the next pair
      (retrain-inner (cdr seq))
     )
     ;; No more data to work with
     (t t)
))


;;
;; Used below...
;;
(setf *markov-results* (make-array '(5) :initial-element
0))

;;
;; Print Trials
;; Input: Data to be used
;; Input: Yes/No should we test selection also?
;;
(defun print-trials (input-data onoff)
  (setq *temp-score* 0)

  ;; Will we be testing selection...
  (set-dummy-props onoff)

  (loop for m-val from 0 to 4 do
    (setf (aref *markov-results* m-val) 0))

  (loop for i from 1 to 10 do
```

```
    (progn

      (set-training-testing input-data i)
      (retrain)

      (loop for m-val from 0 to 4 do
     (progn

       (set-markov m-val)

       (setq *temp-score* (run-test *testing*))
       (format t "Decile ~S Markov ~S : ~S~%"
            i m-val *temp-score*)

       (setf (aref *markov-results* m-val)
          (+ *temp-score*
             (aref *markov-results* m-val)))

       (format t
            "SUBTOTAL(~S): Markov ~S : ~S~%------~%"
            i
            m-val
            (/ (aref *markov-results* m-val) 10))

       )
     ) ;; end of m-val loop

     )
   ) ;; end of m-val loop

)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Similarity Scoring Functions...
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;
;; Sets Hard or Soft Scoring...
;;
(setq *SOFT-WEIGHT* 1)
(defun set-soft-scoring ()
  (setq *SOFT-WEIGHT* 1))
(defun set-hard-scoring ()
```

```
  (setq *SOFT-WEIGHT* 0))

;;
;; The soft scoring function...
;;
(defun soft-score (seq-a seq-b)
  (let ((pairs-a (make-partial-order-pairs seq-a))
     (pairs-b (make-partial-order-pairs seq-b)))
    (/ (* 2 (count-common pairs-a pairs-b))
       (+ (length pairs-a) (length pairs-b)))
  )
)

;;
;; Counts the number of items two lists share
;;
(defun count-common (a b)
  (cond ((null a)
      0)
     ((present-in-list (caar a) (cdar a) b)
      (+ 1 (count-common (cdr a) b)))
     (t
      (count-common (cdr a) b))))

;;
;; Tells if the pair (x . y) is in list b
;;
(defun present-in-list (x y b)
  (cond ((null b)
      nil)
     ((equal (cons x y) (car b))
      t)
     (t
      (present-in-list x y (cdr b)))))


;;
;; Used by the soft scoring function.
;;  GIVEN: '(a b c d e)
;; OUTPUT: ((a . b) (a . c) (a . d) (a . e) (b . c)
 ;;         (b . d) (b . e) (c . d) (c . e) (d . e))
;;
(defun make-partial-order-pairs (seq)
  (setq *temp-partial-order-pairs* '())
  (loop for x from 0 to (- (length seq) 1) do
    (loop for y from (+ x 1) to (- (length seq) 1) do
```

```
      (setq *temp-partial-order-pairs*
         (cons (cons (nth x seq)
               (nth y seq))
            *temp-partial-order-pairs*))
    )
  )
  *temp-partial-order-pairs*
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Data for testing
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq *somedata*
 '(
 (<s> error_class correction application <s>)
 (<s> error_class rule example <s>)
 (<s> error_class mapped correction <s>)
 (<s> error_class corr_elab rule <s>)
 (<s> rule <s>)
 (<s> rule <s>)
 (<s> mapped rule contrast <s>)
 (<s> wrong <s>)
 (<s> error_class correction <s>)
 (<s> error_class rule application <s>)
 (<s> error_class mapped application correction <s>)
 (<s> error_class mapped corr_elab <s>)
 (<s> mapped correction application <s>)
 (<s> error_class corr_elab <s>)
 (<s> mapped correction application corr_elab correction
application <s>)
 (<s> error_class correction <s>)
 (<s> error_class corr_elab mapped <s>)
))
```