HIDING THE HIDDEN: A SOFTWARE SYSTEM FOR CONCEALING CIPHERTEXT AS INNOCUOUS TEXT.

By

Mark T. Chapman

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the degree of

Master of Science

in
Computer Science

at
The University of Wisconsin-Milwaukee
May 1997

HIDING THE HIDDEN: A SOFTWARE SYSTEM FOR CONCEALING CIPHERTEXT AS INNOCUOUS TEXT.

By

Mark T. Chapman

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the degree of

Master of Science

in

Computer Science

at

The University of Wisconsin-Milwaukee

May 1997

G. I. Davida Date

Graduate School Approval Date

HIDING THE HIDDEN: A SOFTWARE SYSTEM FOR CONCEALING CIPHERTEXT AS INNOCUOUS TEXT.

By

Mark T. Chapman

The University of Wisconsin-Milwaukee, 1998

Under the Supervision of Professor G. I. Davida

ABSTRACT

In this thesis we present a system for protecting the privacy of cryptograms to avoid detection by censors. The system transforms ciphertext into innocuous text which is transformed back into the original ciphertext. The expandable set of tools allows experimentation with custom dictionaries, automatic simulation of writing style, and the use of Context-Free-Grammars to control text generation.

Keywords: Ciphertext, Privacy, Information-Hiding

G. I. Davida Date

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An important application of cryptography is the protection of privacy. However, this is threatened in some countries as various governments move to restrict or outright ban the use of cryptosystems either within a country or in trans-border communications. Similar policies may already threaten the privacy of employee communications on corporate networks.

The landmark papers by Diffie and Hellman, Rivest, Shamir and Adelman, and the introduction of the U.S. National Data Encryption Standard (DES), have led to a substantial amount of work on the application of cryptography to solve the problems of privacy and authentication in computer systems and networks [10, 17, 16]. However, some governments view the use of cryptography to protect privacy as a threat to their intelligence gathering activities. While the government of the United States has not yet moved to ban the use of cryptography within its borders, its export controls have lead to a significant chilling effect on the dissemination of cryptographic algorithms and programs. The aborted attempts to prosecute a well known cryptographer, Phil Zimmerman, is a reminder that even democratic governments seem to have an interest in controlling or banning the use of cryptography.

This thesis presents an approach to disguise ciphertext as normal communications to thwart the censorship of ciphertext. The tools convert ciphertext into innocuous text consisting of sentences in a natural language. The programs can also recover the ciphertext from the innocuous text.

Almost everyone has an occasional need to transfer sensitive information across insecure channels such as the Internet, a corporate LAN, or a cellular phone. Cryptography makes untrusted channels more trustworthy.

1.1 Cryptography

A *cryptosystem* transforms plaintext messages (using a key) to render them unintelligible to those who do not possess the key [8]. *Cryptography* is the study of "secret writing" or *cryptograms*. *Encryption* is the process of converting *plaintext* (a normal message) into *ciphertext* (unintelligible gibberish). *Decryption* is the process of transforming the ciphertext back into the original plaintext.

The sender encrypts a plaintext message into ciphertext before transmitting across an untrusted channel. One method is to use an encryption program that scrambles the plaintext using a secret password called a *key* to create the ciphertext. The sender shares the key with the desired recipient (using a secure channel). Eventually, the recipient runs a decryption program with the ciphertext and the proper key to decipher the original message.

Authentication using *digital signatures* is another application of cryptography. *Digital signatures* are a special kind of ciphertext attached to a message to prove the identity of the sender [17].

The effectiveness of a cryptosystem depends on the sophistication of the encryption algorithm with respect to the tools and knowledge of the potential spy or censor. For example, the Roman Empire used a cryptosystem now known as the Caesar Cipher. It simply substituted each letter in the plaintext message with the one three letters down in alphabetical order. For example, the message "COME HELP US" encrypts to "FRPH KHOS XV". In that period of history the technique fooled many would-be spies. With the technology of today Caesar Ciphertext is straightforward to recognize and is easy to break with minimal programming and computational effort.

The Data Encryption Standard (DES) is one modern cipher that uses a key to transpose and substitute bits of plaintext into sophisticated ciphertext. Due to advances in mathematics and technology the "secure" systems of today are the Caesar Cipher's of tomorrow.

The *key-space* is the set of possible keys for a particular cryptosystem. Each key transforms a particular plaintext into different ciphertext. An enormous key-space makes it more difficult to guess the key using brute-force searches. If the algorithm is secure then there are no known methods to shorten the search for the proper key.

Overall, the cryptographic community rejects the idea that the effectiveness of a cryptosystem should rely on the secrecy of the algorithm. Many cryptographers publish algorithms for peer review. The secrecy of the ciphertext depends on the secrecy of the key.

Cryptosystems combine the two basic operations of substitution and transposition to transform plaintext into ciphertext. *Substitution ciphers* replace individual letters (or bits) while preserving the original sequence. The Caesar Cipher is a simple example of a substitution cipher. A *transposition cipher* rearranges the letters (or bits) in a predetermined way. One simple example is to reverse the order of every three letters in a message such that "COME HELP US" becomes "MOCH EPLESU ". A *product cipher* is made from any combination of substitution and transposition ciphers. For example, "COME HELP US" becomes "FRPH KHOS XV" through substitution. "FRPH KHOS XV" becomes "PRFK HSOHVX " through transformation.

Ciphertext is the "secret writing" that results from enciphering a plaintext message. In an effective cryptosystem the resulting ciphertext appears to have no structure [11]. Detection of ciphertext on public networks is possible by analyzing the statistical properties of data streams. Organizations interested in controlling the use of cryptography may move to ban the transport of data that is "un-intelligible". All data that appears to be random becomes suspect.

1.2 Hiding Ciphertext

Detection of ciphertext is a major challenge because there are many ways to make ciphertext look like something else.

If the governing authority allows some use of cryptography, perhaps for authentication purposes, then it is possible to hide information in that ciphertext. The problem of "covert" channels has been studied in a number of contexts. Simmons and Desmedt explored "subliminal" channels which transmit hidden information within cryptograms [19, 20, 21, 22, 9, 6]. When the censors examine the ciphertext they are convinced that it is a normal cryptogram used for authentication. In reality, it contains secret information.

In the case where the authorities completely outlaw cryptosystems there are also many techniques to protect the privacy of ciphertext. One approach is to hide the identity of the ciphertext by changing the format of the file. For example, the pseudo-random data could be hidden within a file format that suggests the data is an executable program.

However, such schemes are not robust since the inspector can test the alleged executable to determine if it actually is a program. If a less-verifiable format is used, such as a graphics file, it may become harder for the censor to automatically detect that it is not a real picture. Nonetheless, the statistical properties of the data in each file would not correspond to similar files.

Another way to disguise ciphertext is to make it look like a compressed archive. The data in a compressed stream may appear to be random [11]. The censor easily exposes the ciphertext by attempting to uncompress the archive.

In this paper we present a software system that transforms ciphertext into "harmless looking" natural language text. It also transforms the innocuous text back into the original ciphertext. Such a scheme may thwart efforts to ban the use of cryptography.

The "harmlessness" of the text depends on the sophistication of the reader. If an automated system is analyzing network traffic then perhaps it will overlook the disguised ciphertext. Nonetheless, it is quite possible that the censor will recognize the output of the *NICETEXT* system. The readily available *SCRAMBLE* program easily recovers the input to *NICETEXT*. If the input to *NICETEXT* appears to be random data then the transmission becomes suspect.

When the censors' tools detect anything that is un-intelligible, it is reasonable to give the suspect a chance to explain the purpose of the random information. If it is found to be ciphertext then the sender will be penalized. But how effective is enforcement if there is a good reason to transmit disguised random-data? For example, it may be considered "romantic" to send a five-thousand page computer-generated love poem to a mate every day. Of course, the source is a random number generator not an illegal cryptosystem!

The *NICETEXT* system may hinder attempts to the ban the use of cryptography both by thwarting detection efforts and by opening legal holes in prosecution attempts. *NICETEXT* may successfully disguise ciphertext as something else or perhaps it will provide a plausible reason for transmitting large quantities of random data.

Chapter 2

Transformations

In this paper we consider the problem of transforming ciphertext into a form that appears innocuous to avoid detection. The adaptability and ambiguity of natural language make it a suitable target.

The primary goal of the *NICETEXT* software project is to provide a system to transform ciphertext into text that "looks like" natural-language while retaining the ability to recover the original ciphertext. In the rest of the paper we focus on the transformation of ciphertext into English. The methods and tools presented can easily apply to other languages.

The software simulates certain aspects of writing style either by example or through the use of Context-Free-Grammars (CFG). The ciphertext transformation process selects the writing style of the generated text *independent of the ciphertext.* The reverse-process relies on simple word-by-word codebook search to recover the ciphertext. The transformation technique is called *linguistic steganography* [13].

This work relates to previous work on mimic-functions by Peter Wayner. Mimic-functions recode a file so that the statistical properties are more like that of a different type of file [25]. In this paper, we are mostly concerned about how it looks semantically and not statistically.

Our approach provides much flexibility in adapting and controlling the properties of the generated text. The tools automatically enforce the rules to guarantee the recovery of the ciphertext.

2.1 $NICETEXT$ and $SCRAMBLE$

Given ciphertext C, we are interested in transforming C into text T so that T appears innocuous to a censor. Let $NICETEXT : C \longrightarrow T$ be a family of functions that maps binary strings into sentences in a natural language. $NICETEXT$ transforms ciphertext into "nice looking" text.

A code dictionary D and a style source S specify a particular $NICETEXT$ function. $NICETEXT$ uses "style" to choose variations of T for a particular C.

Let $NICETEXT_{D,S}(C) \longrightarrow T$ be a function that maps ciphertext C into innocuous text T using D as the dictionary and a style source S. The input to $NICETEXT$ is any binary string C. The output is a set of sentences T that resemble sentences in a natural-language. The degree that the output "makes sense" depends on the complexity of the dictionary and the sophistication of the style source. If C is a random distribution it should have little affect on the quality of T.

Let $SCRAMBLE_D(T) \longrightarrow C$ be the inverse of $NICETEXT_{D,S}$. $SCRAMBLE$ converts the "nice text" T back into the ciphertext C. $SCRAMBLE$ ignores the style information in T. Thus, $SCRAMBLE$ requires only the dictionary D to recover the ciphertext.

Let $T_1 = NICETEXT_{D,S}(C)$ and $T_2 = NICETEXT_{D,S}(C)$, where $T_1 \neq T_2$, then $C = SCRAMBLE_D(T_1) = SCRAMBLE_D(T_2)$. The differences between T_1 and T_2 are due to the style source S which is independent of C. $SCRAMBLE$ ignores style.

These functions are not symmetric, $SCRAMBLE_D(NICETEXT_{D,S}(C)) = C$, but $NICETEXT_{D,S}(SCRAMBLE_D(T)) \neq T$.

For $SCRAMBLE_D$ to be the inverse of $NICETEXT_{D,S}$ the dictionary D must match; thus, $SCRAMBLE_{d_i}(NICETEXT_{d_j,S}(C)) \neq C$ for all $d_i \neq d_j$.

2.2 Transformation Processes

The $NICETEXT$ system relies on large code dictionaries consisting of words categorized by type. A style source selects sequences of types independent of the ciphertext. $NICETEXT$ transforms ciphertext into sentences by selecting words with the matching codes for the proper type categories in the dictionary table. The style source defines case-sensitivity, punctuation, and white-space independent of the input ciphertext. The reverse process simply parses individual words from the generated text and uses codes from the dictionary table to recreate the ciphertext.

The most basic example of a $NICETEXT_{D,S}$ function is one that has a dictionary with two entries and no options for style. Let d consist of the code dictionary in Table 1. Let c be the bit string 011. Let the style source s remain undefined. $NICETEXT$ reads the first bit from the ciphertext, c. It then uses the dictionary d to map $0 \longrightarrow ned$. The process repeats for the remaining two bits in c, where $1 \longrightarrow tom$. Thus, $NICETEXT_{d,s}(011) \longrightarrow nedtomtom$.

$SCRAMBLE_d$ is the inverse function of $NICETEXT_{d,s}$. $SCRAMBLE$ first recognizes the word *ned* from the innocuous text, $t = nedtomtom$. The dictionary, d, maps $ned \longrightarrow 0$. The process continues with $tom \longrightarrow 1$ for the remaining two words. The end result is: $SCRAMBLE_d(nedtomtom) \longrightarrow 011$.

If both dictionary entries were coded to 0 it would be difficult to generate text because 1 would not map to any word. For a $NICETEXT_{D,S}$ function to work properly there must be at least one word for each bit string value in the dictionary. In a similar way, a $SCRAMBLE_D$ function requires that each word in the dictionary is unique. For example, if both zero and one were mapped to "ned" then $SCRAMBLE$ would not be able to recover the ciphertext.

A style source could tell $NICETEXT$ to add space between words. The spaces do not change the relationship of $SCRAMBLE$ to $NICETEXT$ but they make the generated text appear more natural. $SCRAMBLE$ easily ignores the spaces between words.

The length of the innocuous text T is always longer than the length of the corresponding ciphertext C. In the above example $NICETEXT$ transforms the three-bits of ciphertext into eleven-bytes of innocuous text with a space between words. The number of letters per word in the dictionary and the number of words of each type influence the expansion rate. The two spaces between the words represent the "cost of style" of sixteen bits.

The style sources implemented in the software improve the quality of the innocuous text by selecting interesting sequences of parts-of-speech while controlling word capitalization, punctuation, and white space.

In Table 2, the *codes* alone are not unique but all *(type, code)* tuples and all *words*

Code		Word
0	$\longleftrightarrow$	ned
1	$\longleftrightarrow$	tom

Table 1: Basic Dictionary Table

are unique. Let d be the dictionary described in Table 2. Let s be a style component that defines the type as $name\_male$ or $name\_female$ independent of c, in this case $s = name\_male\ name\_female\ name\_male$. $NICETEXT_{d,s}(011) \longrightarrow t$ first reads the type from the style source, s. The first type is $name\_male$. $NICETEXT$ knows to read one bit of c because there are two $name\_male$'s in d. The first bit of c is 0. $NICETEXT$ uses the dictionary, d, to map $(name\_male, 0) \longrightarrow ned$. The second type supplied by s is $name\_female$. Because there are two $name\_female$'s in d, $NICETEXT$ reads one bit of c and then maps $(name\_female, 1) \longrightarrow tracy$. Since there is one remaining type in s, $NICETEXT$ reads the last bit from c. $NICETEXT$ maps the final bit of c such that $(name\_male, 1) \longrightarrow tom$. Thus, $NICETEXT_{d,name\_male\ name\_female\ name\_male}(011) \longrightarrow ned\ tracy\ tom$. Table 3 summarizes the effect of some different style sources on $NICETEXT_{d,s}(011)$.

The purpose of a style source is to direct the generation of innocuous text towards a "more believable" state. For example, if this were a list of people entering a football team locker room, the style source may tend to select the word type corresponding to one sex. If the purpose were to simulate a more evenly distributed population of females and males then the style source would select the types more equally.

The most important aspect of style is type selection. Without it, $NICETEXT_{D,S}$ could not control the part-of-speech selection for natural language text generation. The $SCRAMBLE_D$ functions use the words read from the innocuous text T to look up the code in the dictionary D. It is very important that a word appears in D only once because $SCRAMBLE_D$ ignores the type categories.

Case-sensitivity is another aspect of style. Let d be the dictionary described in Table 2. Let s be the style sequence $name\_female\ name\_male\ name\_male$. Thus, $NICETEXT_{d,s}(011) \longrightarrow jody\ tom\ tom$. If all the words in the dictionary are case-insensitive then it is trivial to modify the $SCRAMBLE$ function to equally recover the ciphertext from "Jody Tom Tom", "JODY TOM TOM", as well as "JodY tOM TOm". Case sensitivity adds believability to the output of $NICETEXT_{D,S}$. $SCRAMBLE_D$ easily ignores word capitalization.

Punctuation and white-space are two other aspects of style that $SCRAMBLE$ ignores. In the above example if the $SCRAMBLE$ function knows to ignore punctuation and white-space then $NICETEXT_{D,S}$ has the freedom to generate many more innocuous strings, including:

- "Jody? Tom? TOM!!"
- "Jody, Tom, Tom."
- "JODY... Tom... tom..."

All three examples above reduce to three lowercase words: *jody tom tom*; thus, $SCRAMBLE_d(t_i)$ recovers the ciphertext, $c = 011$.

A style source also may cause $NICETEXT$ to include words that are not in the dictionary. As long as $SCRAMBLE$ can ignore the elements of style, the inverse relationship of $SCRAMBLE$ to $NICETEXT$ is valid. For example, let t be the following innocuous text: "Amy, Lucy, and Jody Smith went with Tom Barker. They will meet Tom Reynolds." First, $SCRAMBLE_d(t)$ views all words as lowercase, giving: "amy, lucy, and jody smith went with tom barker. they will meet tom reynolds." Next, $SCRAMBLE$ ignores all punctuation which reveals the following list of words: "*amy lucy and jody smith went with tom barker they will meet tom reynolds*". $SCRAMBLE_d$ ignores any words that are not dictionary, leaving: *jody tom tom*. Finally, $SCRAMBLE_d(jody\ tom\ tom) \longrightarrow 011$.

In practice, $SCRAMBLE$ ignores style and transforms T into C in one pass. It is very inefficient to use such a small dictionary or to insert words directly from the style-source. In the above case, the three bits ciphertext grew to sixty-nine bytes of innocuous text.

The construction of large and sophisticated dictionary tables [1] is key to the success of the $NICETEXT$ system. The tables need to maintain certain properties for the transformations to be invertable. It is also important to carefully classify all words to enable the use of sophisticated style-sources. Chapter 3 explores the "art" of constructing complex tables.

[1] A "large and sophisticated" dictionary contains more than 150,000 words carefully categorized into over 350 types.

Type	Code		Word
name_male	0	$\longleftrightarrow$	ned
name_male	1	$\longleftrightarrow$	tom
name_female	0	$\longleftrightarrow$	jody
name_female	1	$\longleftrightarrow$	tracy

Table 2: Basic Dictionary Table with Multiple Types.

Style s	Ciphertext c		$NICETEXT_{d,s}(c)$
name_male name_male name_male	011	$\longrightarrow$	"ned tom tom"
name_male name_male name_female	011	$\longrightarrow$	"ned tom tracy"
name_male name_female name_male	011	$\longrightarrow$	"ned tracy tom"
name_male name_female name_female	011	$\longrightarrow$	"ned tracy tracy"
name_female name_male name_male	011	$\longrightarrow$	"jody tom tom"
name_female name_male name_female	011	$\longrightarrow$	"jody tom tracy"
name_female name_female name_male	011	$\longrightarrow$	"jody tracy tom"
name_female name_female name_female	011	$\longrightarrow$	"jody tracy tracy"

Table 3: How Style Changes $NICETEXT$.

Trivial examples demonstrate the importance of style. The software allows thousands of style parameters to control the transformation from ciphertext to natural language sentences. Chapter 4 describes how to define style sources in the software.

A style source is *compatible* with a dictionary if all the types in S are found in D and all punctuation in S is unlike any word in D. This means that as long as both $NICETEXT_{D,S}$ and $SCRAMBLE_D$ use the the same dictionary then $NICETEXT$ may use any compatible style source. A style source may be compatible with many dictionaries and a dictionary may be compatible with many style sources.

2.3 $SIZER$ and $DESIZER$

The size of C could restrict the selection of style-sources when the dictionary has type categories with more than two words. For example, let d be the code dictionary defined in Table 4. Let $s = name\_male\ name\_female$. Thus,
$NICETEXT_{d,s}(011) \longrightarrow ned\ kimberly$. (The inverse is:

$SCRAMBLE_d(ned\ kimberly) \longrightarrow 011.)$ Table 5 shows that the style source $s =$ *name_male name_male name_male* is the only one that specifies a sequence of types that requires three bits. Given the ciphertext $c = 011$, somehow $NICETEXT$ would need to know how to choose the "correct" style source.

It would be cumbersome to generate the data in Table 5 for all sizes of C, all dictionaries, and all style sources. In fact, there are cases where the code-length required for a style cannot match the length of C. (i.e. $\overline{C} = 3$ and all types in the dictionary have four words; thus, all codes lengths required by S are even numbers.)

There is no need to solve the problem of matching S to C for a particular D. The style source is supposed to be independent of C. That includes the length of C.

The $SIZER$ and $DESIZER$ functions preserve the independence of S and $\overline{C}$. Let R be a pseudo-random [2] number source. Let $SIZER_R(C)$ be a function that converts the bit string C into a string consisting of a fixed length number describing the length of C concatenated with C plus an infinitely long string of randomness. Thus, $SIZER_R(C) \longrightarrow \overline{C} + C + RANDOMSTRING$.

Let $DESIZER$ be the inverse of $SIZER$ such that for all C, $DESIZER(SIZER_R(C)) = C$. This allows the following relationship to hold:

$DESIZER(SCRAMBLE_D(NICETEXT_{D,S}(SIZER_R(C)))) = C$.

By integrating $SIZER$ into $NICETEXT$ (and $DESIZER$ into $SCRAMBLE$), all $NICETEXT$ functions can finish a style sequence or continue for a long time after the end of the ciphertext. In the above example, all eight style sequences of *name_female* and *name_male* are available independent of the length of the ciphertext. This integration allows $NICETEXT$ to complete the last generated sentence (or paragraph, or chapter...) required by a style source.

2.4 Merged Type Management

It is important that all dictionaries maintain certain properties to support the inverse relationship of $SCRAMBLE$ to $NICETEXT$. The properties selected in this software project are:

[2] A creative source for R might be some ciphertext...

Type	Code		Word
name_male	0	$\longleftrightarrow$	ned
name_male	1	$\longleftrightarrow$	tom
name_female	00	$\longleftrightarrow$	jody
name_female	01	$\longleftrightarrow$	tracy
name_female	10	$\longleftrightarrow$	darla
name_female	11	$\longleftrightarrow$	kimberly

Table 4: Dictionary Table with More Girls.

Style S	Number of Bits of c Required
name_male name_male name_male	$1+1+1=3$
name_male name_male name_female	$1+1+2=4$
name_male name_female name_male	$1+2+1=4$
name_male name_female name_female	$1+2+2=5$
name_female name_male name_male	$2+1+1=4$
name_female name_male name_female	$2+1+2=5$
name_female name_female name_male	$2+2+1=5$
name_female name_female name_female	$2+2+2=6$

Table 5: The Number of Bits of C Required for a Style Source.

Before			After	
Type	Word		Type	Word
name_male	chris	becomes...	name_female,name_male	chris
...	...		...	...
name_female	chris		...	...
...	...		...	...

Table 6: Merging Types for *Chris*.

1. There must be at least two words of one type in the dictionary. Otherwise $NICETEXT$ can not convert any bits of the ciphertext.

2. The number of words of each type must be a power of two to fully support fixed length codes within a type category.

3. Each word must be unique when converted to lower case. (All words are case-insensitive in the dictionary so the style sources can capitalize at will.)

4. Each *(type, code)* must be unique. Thus, the words in a type must be coded by simple enumeration.

5. There is no need for correlation between the *(type, code)* and the alphabetical sequence of words.

What if a word belongs to multiple type categories? What if there is only a single word of a given type? What if there are more than 2^n words of a type? There are many ways to deal with these questions. The solutions presented here are those implemented in the software.

At dictionary construction time, if a word belongs to multiple type categories then the *sortdct* process creates new merged type category. For example, if "chris" is both a male name and a female name then *sortdct* assigns a new type of $name\_female, name\_male$ as shown Table 6. The merging of types is a necessary step when creating D.

It is acceptable to have only a single word of a given type because $2^0 = 1$. The implications are that $NICETEXT_{D,S}(C)$ uses zero bits of the ciphertext C to select the next word in T. The style source may cause $NICETEXT_{D,S}$ to include the word

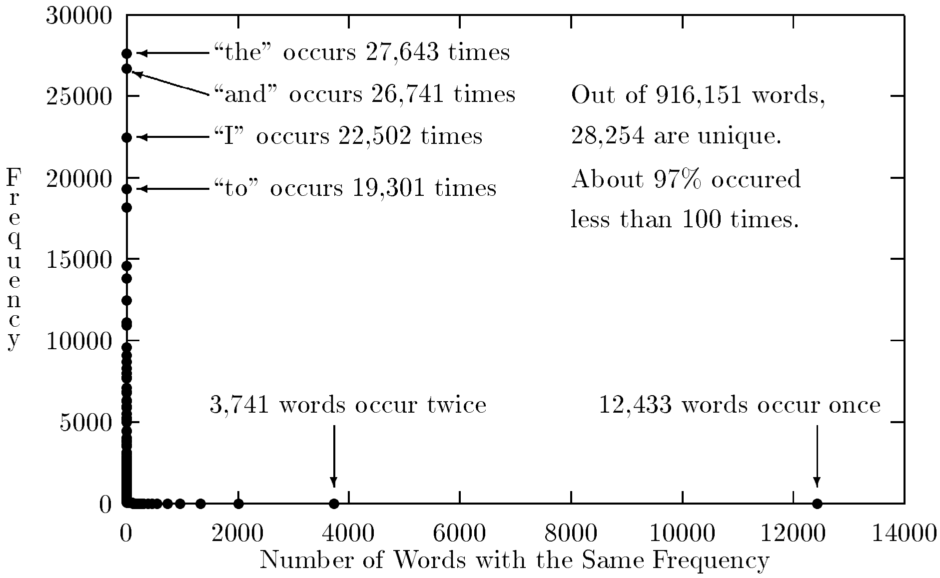


Figure 1: Number of Words of Each Frequency: *Shakespeare*.

in T. $SCRAMBLE_D$ ignores it. (More specifically, $SCRAMBLE$ recovers zero bits of C from reading such a word from T.)

Let f be the number of words in a single type category. Let $g = 2^{\lfloor \log_2 f \rfloor}$ be the largest power of two less than or equal to f. $NICETEXT$ ignores all but the first g words of each type because any remaining words do not have a code assigned in the dictionary. A solution is to create merged type categories during dictionary construction where the number of words of each type is an exact power of two. Table 7 shows an example. Any type category with more than one word can be divided into sub-types with each sub-type containing a number of words that is some power-of-two. The limit is to place each word from the initial type category into individual sub-types with $2^0 = 1$ members. The eventual cost of this option is the very high expansion rate of $\overline{C}$ to $\overline{T}$. It is better to use sub-type categories with a large number of words in each sub-type.

It is useful to group words by frequency while fixing the problem of seldom having exactly 2^n words of a given type. Figure 1 shows the number of words of each frequency for *The Complete Works of William Shakespeare.* [3] Most natural language

[3]The electronic text from Project Gutenberg is available at *ftp://ftp.freebsd.org/pub/gutenberg/etext94/shaks12.txt*. The *listword* program extracted the words

Before				After			
Type	Code		Word	Type	Code		Word
name_male	0	$\longleftrightarrow$	ned	name_male,TypeA	0	$\longleftrightarrow$	ned
name_male	1	$\longleftrightarrow$	tom	name_male,TypeB	0	$\longleftrightarrow$	tom
name_male	N/A	$\longleftrightarrow$	brad	name_male,TypeB	1	$\longleftrightarrow$	brad

Table 7: Merging Types to Allow Arbitrary Number of Words.

texts analyzed, including this thesis document, had the characteristic of disproportionatly using a subset of available words. Although Figure 1 did not consider word type categories, individual categories usually follow a similar distribution. For example, out of 27,915 possible words of the type *name*, most occur very few times, or not at all, in a single text. This property seems to hold true even if the text is a phone book! "Popular" *name*'s occur much more often than most others. It may be beneficial to group words within a type by frequency to increase the quality of the innocuous text. Although a small number of sub-types would have a small number of words, most sub-types would still have many words.

The decision to merge types has greatly simplified the implementation of the software. Merging types avoids the use of variable length codes to better simulate word frequency. It also is part of a solution to allow phrases, multi-type and multi-context words.

Merging types is one solution for constructing sophisticated dictionaries. *NICETEXT* does not require the use of merged types although it helps generate higher quality innocuous text. The next chapter describes programs that greatly simplify merged-type management and other aspects of dictionary construction.

from the unmodified file which includes an insignificant amount of copyright notice, etc.

Chapter 3

Dictionary Construction

The quality of the innocuous text generated by $NICETEXT_{D,S}(C)$ depends on the sophistication of both the dictionary, D, and the the style source, S. The primary responsibility of a style source is to select interesting sequences of types from D. The types in D are the only types available to a style source. [1] Thus, the sophistication of S depends on the sophistication of D. This chapter explores the construction of advanced dictionaries for the $NICETEXT$ system.

Figure 2 diagrams the processes for creating a valid dictionary, D. A combination of sources creates a word-list, $WLIST$. Several processes may use $WLIST$ to create a *type-word list*, $TWLIST$. There are many other ways to create $TWLIST$ including manual entry. The *sortdct* process converts the $TWLIST$ into a merged-and-sorted type-word list, $MTWLIST$. Finally, the *dct2mstr* program creates a valid dictionary from $MTWLIST$.

The simple file formats and the supporting programs provide an expandable set of tools to manage the mechanics of constructing a valid dictionary table. The focus of this chapter is to evaluate different sources for generating dictionaries. The ultimate goal is to enable $NICETEXT$ to output the highest quality innocuous text.

3.1 Simple Word Lists: $WLIST$

A word list, $WLIST$, is simply a list of words separated by new lines in a text file. There are almost no restrictions on the properties of $WLIST$. The number of words does not matter. The case of the letters in the words is inconsequential. A word may

[1] If S specifies types that are not in D then S is not compatible with D; therefore, $NICETEXT$ may not use this combination.

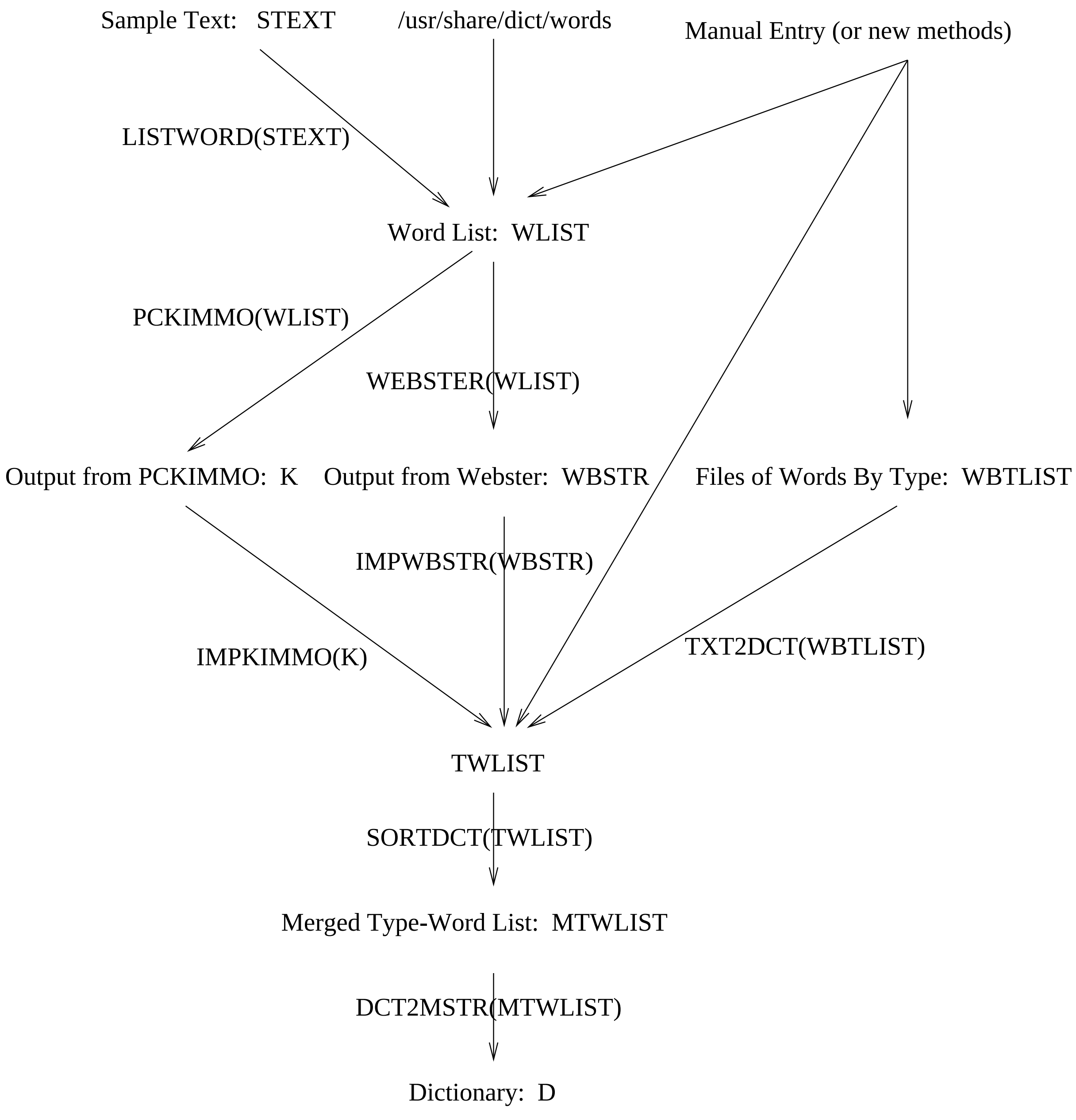


Figure 2: Dictionary Construction Diagram

appear multiple times. The word list may contain hyphenated-words, words with apostrophes, phrases, and foreign words. In short, anything goes.

There are many readily available word lists. The */usr/share/dict/words* file on a FreeBSD system is one example with over 230,000 words [14]. Many systems have similar files.

The *listword* utility uses the scanner from $SCRAMBLE$ to extract lists of unique English words from text files containing natural language text. The Project Gutenberg at *ftp://ftp.freebsd.org/pub/gutenberg* provides electronic copies of public-domain texts which contain many words. UseNet news groups and the world-wide-web are other significant sources of words available electronically. There are many uses for electronic text documents here and in the style-source chapter. The goal is to collect a large quantity of words. [2]

It is not critical to use the *listword* program to create $WLIST$. Any process that can output a list of words, one word per line, will work (including manual entry).

3.2 Type-Word Lists: $TWLIST$

Let $TWLIST$ denote a type-word list composed of *(type, word)* pairs. Each pair defines the word as a member of the corresponding type. Table 8 is an example.

The only rule for generating a valid $TWLIST$ is that no type may contain white-space. Otherwise, it would be difficult to determine where the type string stops and the word string begins. No type should contain any commas because of the way the system denotes merged types.

A word can occur multiple times in the same or different types in $TWLIST$. Words in $TWLIST$ can be freely capitalized. There can be any number of words of each type. The entries in $TWLIST$ do not need to be sorted. All the rules to transform $TWLIST$ into D are applied by a set of functions described in section 3.3.

The challenge is to select meaningful *(type, word)* pairs. The remainder of this section compares several methods to generate type-word lists. All the following methods may be combined by simple concatenation of the resulting lists.

[2] It may be useful to collect some word frequency information if the sources are natural language texts.

Type	*Word*
art	the
conj	and
object	bill
object	gift
object	mail
object	message
object	money
person	Bill
person	Bob
person	Heather
person	Lisa
person	Shirley
prep	to
verb	gave
verb	sent

Table 8: Sample Type-Word List, $TWLIST$.

3.2.1 Manual Construction

One way to construct a type-word list is to manually enter the list in a text editor. It is amazing how many words and type categories a person knows. Is it unreasonable to simply look up the rest of the words in Websters [12] dictionary?

The most obvious problem with the manual method is that it takes too long to enter large lists. A less-obvious problem is that it is difficult to select meaningful type categories without considering the eventual grammatical requirements of a natural-language style-source. Matching the part-of-speech with all the possible word variations using Websters dictionary and an English grammar, such as [23], is a tremendous undertaking.

It is possible to construct a sophisticated but small $TWLIST$ by hand. Manually constructing large and sophisticated type-word lists within a reasonable amount of time is not likely. The manual method is best suited to tweaking a small number of entries from some automated method.

3.2.2 Construction from Files of Like Words: *txt2dct*

The *txt2dct* utility simplifies the creation of larger *TWLIST*'s by expanding lists of words already grouped in separate files by type. On the Internet [3] there are files that contain many words of the same type, such as: *name_male*, *name_female*, *name_family*, and *places*. The *txt2dct* program reads each word in the *name_female* file and outputs a *(type, word)* pair such as *(name_female, Ann)*. The process repeats for all words in each file. The *txt2dct* program is a quick way of making large type-word lists.

The problem with *txt2dct* is that there are relatively few useful lists readily available. Even if there are a large number of such lists the problem of matching the types to some grammatical structure remains. Thus, the resulting *TWLIST*'s generate large but unsophisticated D's.

Due to the availability of single-type word lists, the *txt2dct* program seems best at categorizing proper nouns such as names and places.

3.2.3 Automatic Generation

There are many programs that categorize words by part-of-speech. The goal of automatic *TWLIST* generation is to format the output of a word definition program into the *(type, word)* pairs of a *TWLIST*.

Some word definition programs can dump their entire knowledge of words with all possible usages. Other programs require modification. In some cases it may not be feasible to modify a program or access a definition database directly. A solution is to define the words in a word-list, *WLIST*, one word at a time. In any case, an import program extracts the words and types from the definition program and formats the output into a *TWLIST*.

3.2.4 Webster On-line

The *impwbstr* program interfaces to the on-line Webster dictionary found on many NextStep systems. The output from the *webster* program contains definitions and part-of-speech designations for many words in a word-list. *Impwbstr* assigns the type

[3] One source is Bob Baldwin's collections of words from MIT augmented by Matt Bishop and Daniel Klein at *ftp://ftp.funet.fi/pub/doc/dictionaries/DanKlein/*.

based on the part-of-speech parsed from the definition of each word. The output of *impwbstr* is a type-word list.

The problem with *impwbstr* is the difficulty of selecting meaningful types for all likely variations of a word. The type assignments in a $TWLIST$ from *impwbstr* are not specific enough to support more than a basic level of agreement in the text generated by $NICETEXT_{D,S}$ (where D comes from $TWLIST$).

It is possible to enhance the *impwbstr* program to identify more specific type categories to improve word agreement. This requires significant time and language expertise.

Creating large $TWLIST$'s with *impwbstr* is much like using the *txt2dct* program. It is easy to make large, but unsophisticated $TWLIST$'s. The $TWLIST$'s tend to be more sophisticated but not enough to generate "believable" innocuous text.

The *impwbstr* method is also similar to the manual construction technique. The benefit is the possible automation of any useful heuristics. An English grammar book may help to select meaningful types.

3.2.5 Morphological Word Parsing: *pckimmo*

Significant research exists in the area of word classification. More importantly, with respect to this thesis, there are programs available for sophisticated word type identification. *Pckimmo* is one such program [5].

The *pckimmo* program is a morphological word parser with a two-level [4] morphology [2, 3, 4]. *Pckimmo* uses word-grammars to classify words. These grammars are an effective way of identifying the many different variations of words. The web page at *http://www.sil.org/pckimmo/v2/doc/introduction.html#sec1.1* explains:

> Even for English a morphological parser may be necessary. Although English has a limited inflectional system, it has very complex and productive derivational morphology. For example, from the root compute come derived forms such as *computer, computerize, computerization, recomputerize, noncomputerized,* and so on. It is impossible to list exhaustively in

[4] The first level breaks a word up into parts such as the root word and the suffixes and prefixes. The second level classifies the word based on the results from the first-level.

a lexicon all the derived forms (including coined terms or inventive uses of language) that might occur in natural text.

Figure 3 shows the parse tree for the word *apple* using the *pckimmo* program with the *englex* word grammar. The tree shows that the word *apple* is a noun. *Apple* is a third-person singular word. *Apple* is not plural and it is not a proper noun. Figure 4 shows two parse trees for the word *structure*.

```
'apple

Word:
[ cat:   Word
  clitic:-
  drvstem:-
  head:    [ agr:     [ 3sg:   + ]
             number:SG
             pos:   N
             proper:-
             verbal:- ]
  root:  `apple
  root_pos:N ]

1 parse found
```

Figure 3: Parse Tree and Feature Structure for *apple*

Although it is far beyond the scope of this thesis to explain the details of morphological word parsing, the application of that research to the $NICETEXT$ system is very straightforward.

Pckimmo and *englex* define all possible parses of the words in a word list, $WLIST$. The *impkimmo* program assigns a type to a word by constructing a string that represents each parse-tree from *pckimmo*. If a word has multiple parse-trees then *impkimmo* places the word into multiple type categories. The goal is to take a word-list, $WLIST$, and generate a type-word list, $TWLIST$. For example, the type for *apple* becomes "N_3sg+SgProp-Verbal-". The "N_" shows that *apple* is a noun. The remaining part of the type string describes the features of the word. Table 9 is a type-word list for several other words.

```
‘structure

Word:
[ cat:   Word
  head:    [ pos:   V
             vform: BASE ]
  root:  ‘structure
  root_pos:V
  clitic:-
  drvstem:- ]

Word:
[ cat:   Word
  head:    [ agr:     [ 3sg:   + ]
             number:SG
             pos:   N
             proper:-
             verbal:- ]
  root:  ‘structure
  root_pos:N
  clitic:-
  drvstem:- ]

2 parses found
```

Figure 4: Parse Tree and Feature Structure for *structure*

Type	*Word*
N_3sg+SgProp-Verbal-	apple
V_Base	structure
N_3sg+SgProp-Verbal-	structure
V_Base	go
V_3sg+PresSFin+	goes
V_EnFin-	gone
V_IngFin-	going
V_PastEdFin+	went
AJ_AbsVerbal-	quick
AV	quick
AJ_CompVerbal	quicker
V_BaseFin-	quicken
AJ_SuperVerbal-	quickest
AV	quickly
N_3sg+Sg	quickness
PR_3sg-1SgNomReflex-Wh-	i
PR_3sg+3SgAccReflex-Wh-	it
PR_3sg+3SgNomReflex-Wh-	it
PR_3sg+3SgNomReflex-Wh-	he
PR_3sg+3SgNomReflex-Wh-	she
PR_3sg-3PlNomReflex-Wh-	they
PR_3sg-1PlNomReflex-Wh-	we
PR_3sg-2SgAccReflex-Wh-	you
PR_3sg-2PlNomReflex-Wh-	you
PR_3sg-2PlAccReflex-Wh-	you
PR_3sg-2SgNomReflex-Wh-	you
N_3sg+SgProp-Verbal-	expert
N_3sg-Pl	experts
N_3sg+SgProp-Verbal-	university
PP	of
N_3sg+SgProp+Verbal-	wisconsin
N_3sg+SgProp+Verbal-	milwaukee

Table 9: Type-Word List Generated by *Impkimmo*.

Type	*Word*
rhymeL2_aa1g	bog
rhymeL2_aa1g	clog
rhymeL2_aa1g	fog
rhymeL2_aa1g	frog
rhymeL2_aa1g	hog
rhymeL2_aa1g	hogg
rhymeL2_aa1g	jog
rhymeL2_aa1g	prague
rhymeL2_aa1g	prolog
rhymeL2_aa1g	rog
rhymeL2_aa1g	rogge
rhymeL2_aa1g	slog
rhymeL2_aa1g	smog
rhymeL2_aa1g	tague

Table 10: Rhyming Type-Word List Generated from *CMUDICT*.

All variations of each word to be used by *NICETEXT* must be present in *WLIST*. The synthesis mode of *pckimmo* expands *WLIST* with words such as nonrecomputerizationalism[5]. To select only the most common uses, including "inventive uses" of words, the *listword* utility first creates a word-list from large English texts.

The *pckimmo* and *impkimmo* software create large and sophisticated type-word lists from *WLIST*. It is the best single resource for generating the dictionaries for this software project. A combination of techniques can greatly improve the quality of the type-word lists. Although *pckimmo* helps classify words by part-of-speech, there still are other ways to classify words such as by sound and by meaning.

3.2.6 Word Types that Rhyme

The Carnegie Mellon Pronouncing Dictionary provides a phonetic break-down of a large number of words. Figure 5 is an excerpt of the *cmudict* text file.

One use of this dictionary with the *NICETEXT* system is to classify words that

[5](Although this is not a real example, it demonstrates the potential problem of generating too many "inventive uses" of words.)

```
## Date:  11-8-95
##
## The Carnegie Mellon Pronouncing Dictionary
## [cmudict.0.4] is Copyright 1995 by Carnegie Mellon University.
## Use of this dictionary, for any research or
## commercial purpose, is completely unrestricted.
## If you make use of or  redistribute this material,
## we would appreciate acknowlegement of its origin.
...
ABERRANT  AE0 B EH1 R AH0 N T
ABERRATION  AE2 B ER0 EY1 SH AH0 N
ABERRATIONS  AE2 B ER0 EY1 SH AH0 N Z
...
ACADEMIA  AE2 K AH0 D IY1 M IY0 AH0
ACADEMIC  AE2 K AH0 D EH1 M IH0 K
ACADEMICALLY  AE2 K AH0 D EH1 M IH0 K L IY0
ACADEMICIAN  AE2 K AH0 D AH0 M IH1 SH AH0 N
ACADEMICIANS  AE2 K AH0 D AH0 M IH1 SH AH0 N Z
ACADEMICIANS(2)  AH0 K AE2 D AH0 M IH1 SH AH0 N Z
...
BOG  B AA1 G
BOG(2)  B AO1 G
BOGACKI  B AH0 G AA1 T S K IY0
BOGACZ  B AA1 G AH0 CH
...
DOG  D AO1 G
DOG'S  D AO1 G Z
...
FROG  F R AA1 G
FROGG  F R AA1 G
FROGGE  F R AA1 G
FROGMAN  F R AA1 G M AE2 N
...
```

Figure 5: Excerpt of Carnegie Mellon Pronouncing Dictionary

sound alike such as *bog* and *frog*. This opens up a whole new avenue for $NICETEXT$ to generate poetry. [6]

The challenge to is define "good rhyme" from phonetic information. The $NICETEXT$ system contains some experimental programs that attempt to classify words into types that rhyme. The output is a type-word list where the type is a string constructed from the phonetic information in *cmudict* and a description of which parts of the words rhyme. Table 10 is an example type-word list extracted from the pronouncing dictionary. The meaning of the type in this case is that the last two phonetics in each word rhyme with *frog*.

The *sortdct* program merges the rhyming types of each word along with the part-of-speech types from the other sections. Eventually the word type categories will correspond to meaning such as "color", or "quantity", or "objects that can be described by bright colors and large quantities...". It is up the the style-source to make sense of all these categories. Most style-sources ignore type categories for rhyming words.

3.2.7 Review of Type-Word List Construction

A combination of techniques from a variety of sources, including *listword*, */usr/share/dict/words*, and manual entry create a word list, $WLIST$. External dictionaries categorize all the words in $WLIST$ so that an import program such as *impwbstr* or *impkimmo* can generate $TWLIST$. The *txt2dct* program and manual processes may also expand $TWLIST$.

The $NICETEXT$ system works with other natural languages because of the simple yet flexible format of $TWLIST$. The bottom line is that no matter the technique, $TWLIST$ is just a list of *(type, word)* pairs. Figure 6 compares several options for creating a type-word list, $TWLIST$. The goal is to make large and sophisticated lists. A combination of techniques seems to work best to categorize words by part-of-speech, sound, and meaning.

[6] Edgar Allen Poe concealed information inside his poetry. [13].

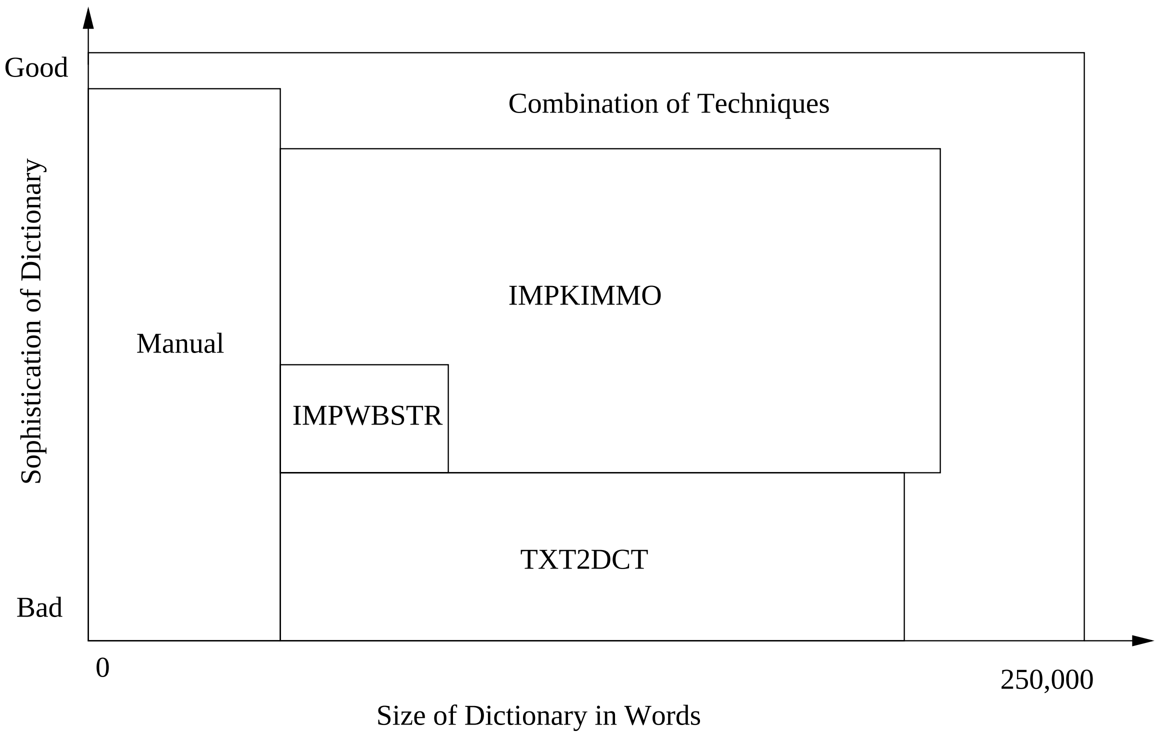


Figure 6: Size vs. Sophistication for Constructing $TWLIST$.

3.3 Dictionary Construction ($TWLIST \longrightarrow D$)

The *sortdct* and *dct2mstr* programs convert a type-word list into a valid master dictionary table. The first step is to convert $TWLIST$ into a merged-and-sorted word list, $MTWLIST$. The next step is to convert $MTWLIST$ into a master dictionary, D.

A $SCRAMBLE_D(T)$ function must be able to recognize all words in D within all possible innocuous texts, T (where D comes from $TWLIST$ and T is the output of $NICETEXT_{D,S}(C)$). Currently, this means no words may contain white space. It is not difficult to modify the scanner in $SCRAMBLE_D$ to allow words from other natural languages.

Let $SORTDCT(TWLIST) \longrightarrow MTWLIST$ be a function that transforms any type-word list into a merged-and-sorted type-word list in which all words are uniquely defined. The $SORTDCT$ function converts all words in $TWLIST$ to lower case and merges the types as needed. $SORTDCT$ filters out entries that destroy the inverse relationship of $SCRAMBLE$ to $NICETEXT$.

Type	Word
conj	and
object,person	bill
person	bob
verb	gave
object	gift
person	heather
person	lisa
object	mail
object	message
object	money
verb	sent
person	shirley
art	the
prep	to

Table 11: Sample Merged and Sorted Definition Entry List, $MTWLIST$

Let $TWLIST$ be the type-word list in Table 8. Table 11 shows the merged and sorted type-word list, $MTWLIST$, from $SORTDCT(TWLIST) \longrightarrow MTWLIST$. All words in $MTWLIST$ are lower case and uniquely defined. *Bill* is both an *object* (*bill*) and a *person* (*Bill*) in $TWLIST$ but there is only one entry for the *object,person* named *bill* in $MTWLIST$.

The next step is to convert $MTWLIST$ into a valid dictionary, D. Let $DCT2MSTR(MTWLIST) \longrightarrow D$ be a function to transform a merged and sorted type-word list into a dictionary. The *dct2mstr program* implements the $DCT2MSTR$ function and outputs a set of normalized indexed tables.

Table 12 is the type database table generated by applying *dct2mstr* to the merged-and-sorted type-word list, $MTWLIST$, found in Table 11. The purpose of the type table is to normalize the type categories described in the dictionary table. The type column is the primary key. The frequency column holds a count of how many words of this type are in the corresponding dictionary table. The description column holds the description of the type as found in $MTWLIST$.

Table 13 is the corresponding dictionary database table (shown twice sorted different ways). The purpose of the dictionary table is to be an easily accessible code dictionary for use with the *nicetext* and *scramble* programs. The type column points

Type	*Frequency*	*Description*
0	0	(unused type code)
1	1	art
2	4	object
3	1	object,person
4	4	person
5	1	prep
6	2	verb
7	1	conj

Table 12: Type Table From *dct2mstr* Using *MTWLIST* as Input.

to the correct row in the type table. The code column contains a bit string. The word column contains a string representing a single word. Frequency is an optional column. [7]

The type table is indexed by (type). The dictionary table has two unique indexes: (type, code) and (word). These tables and indexes together compose the software implementation of D.

The most complex part of generating sophisticated dictionaries is choosing the proper type categories for the words. The *sortdct* and *dct2mstr* programs automate the conversion of a type-word list into a valid dictionary table. It is an interactive process to create compatible dictionaries and style sources. The next chapter describes the implementation of style sources.

[7] The *genmodel* program updates the frequency column when making a *distribution dictionary. Although NICETEXT does not directly use this information other processes may use frequency data to generate better style-sources and better dictionaries..* (See section 4.3)

D as viewed by $NICETEXT_{D,S}$ (Sorted by (Type, Code))				*D as viewed by $SCRAMBLE_D$ (Sorted by Word)*			
Type	*Code*		*Word*	*Word*		*Type*	*Code*
0	N/A	$\longrightarrow$	(unused code)	(unused code)	$\longrightarrow$	0	N/A
1	N/A	$\longrightarrow$	the	and	$\longrightarrow$	7	N/A
2	00	$\longrightarrow$	money	bill	$\longrightarrow$	3	N/A
2	01	$\longrightarrow$	message	bob	$\longrightarrow$	4	11
2	10	$\longrightarrow$	mail	gave	$\longrightarrow$	6	1
2	11	$\longrightarrow$	gift	gift	$\longrightarrow$	2	11
3	N/A	$\longrightarrow$	bill	heather	$\longrightarrow$	4	10
4	00	$\longrightarrow$	shirley	lisa	$\longrightarrow$	4	01
4	01	$\longrightarrow$	lisa	mail	$\longrightarrow$	2	10
4	10	$\longrightarrow$	heather	message	$\longrightarrow$	2	01
4	11	$\longrightarrow$	bob	money	$\longrightarrow$	2	00
5	N/A	$\longrightarrow$	to	sent	$\longrightarrow$	6	0
6	0	$\longrightarrow$	sent	shirley	$\longrightarrow$	4	00
6	1	$\longrightarrow$	gave	the	$\longrightarrow$	1	N/A
7	N/A	$\longrightarrow$	and	to	$\longrightarrow$	5	N/A

Table 13: Dictionary Table From *dct2mstr* Using $MTWLIST$ as Input.

Chapter 4

Style Sources

The goal of a style source is to provide "interesting" sequences of parts-of-speech and format instructions so that $NICETEXT$ generates high-quality innocuous text. This chapter compares how several different style-sources interact with the dictionaries from the previous chapter.

The basic style-source is a *sentence-model*. A sentence model is a template of a single natural language sentence. For example, the model "{Cap} person verb {CAPSLOCKON} object {capslockoff} {!}" tells a $NICETEXT$ function to:

1. Make a note to format the next word of output with the first letter in upper case.
2. Select a word from the dictionary by using the *(type, code)* index where the type is $person$. (The code comes from the first bits of the ciphertext C.)
3. Output the word selected above according to the last format note.
4. Select a word from the dictionary by using the *(type, code)* index where the type is $verb$. (The code comes from the next bits of the ciphertext C.)
5. Output the word selected above according to the last format note.
6. Make a note to format all words of output using capital letters only.
7. Select a word from the dictionary by using the *(type, code)* index where the type is $object$. (The code comes from the next bits of the ciphertext C.)
8. Output the word selected above according to the last format note.

Shirley sent MONEY! 00 0 00	Lisa sent MONEY! 01 0 00	Heather sent MONEY! 10 0 00
Bob sent MONEY! 11 0 00	Shirley sent MESSAGE! 00 0 01	Lisa sent MESSAGE! 01 0 01
Heather sent MESSAGE! 10 0 01	Bob sent MESSAGE! 11 0 01	Shirley gave MONEY! 00 1 00
Lisa gave MONEY! 01 1 00	Heather gave MONEY! 10 1 00	Bob gave MONEY! 11 1 00
Shirley gave MESSAGE! 00 1 01	Lisa gave MESSAGE! 01 1 01	Heather gave MESSAGE! 10 1 01
Bob gave MESSAGE! 11 1 01	Shirley sent MAIL! 00 0 10	Lisa sent MAIL! 01 0 10
Heather sent MAIL! 10 0 10	Bob sent MAIL! 11 0 10	Shirley sent GIFT! 00 0 11
Lisa sent GIFT! 01 0 11	Heather sent GIFT! 10 0 11	Bob sent GIFT! 11 0 11
Shirley gave MAIL! 00 1 10	Lisa gave MAIL! 01 1 10	Heather gave MAIL! 10 1 10
Bob gave MAIL! 11 1 10	Shirley gave GIFT! 00 1 11	Lisa gave GIFT! 01 1 11
Heather gave GIFT! 10 1 11	Bob gave GIFT! 11 1 11	

Table 14: Thiry-two Sentences with the Corresponding Ciphertext.

9. Make a note to stop printing all words in upper case.

10. Output an exclamation point.

The dictionary in Table 13 has four words of the type $person$, two words of the type *verb*, and four words of the type $object$. Thus, the above sentence model could generate $4 * 2 * 4 = 32$ different sentences. These thiry-two sentences are found in Table 14 with the corresponding ciphertext below each example.

If the style source repeatedly supplies a single sentence model then $NICETEXT$ generates all sentences with the same template. The following sections discuss two *meta-style-sources* that provide sequences of sentence models. The final two sections demonstrate one way to automatically generate style sources from sample natural-language texts.

4.1 Sentence Model Tables

A *sentence model table* is a fixed set of sentence models. S can be a meta-style-source that selects models from such a table. When $NICETEXT_{D,S}(C)$ exhausts the instructions for a single sentence template then the meta-style-source selects another model from the table. S selects these models according to a certain probability distribution independent of C. The goal is to make $NICETEXT$ generate an "interesting" mix of sentences.

Table 15 represents a small sentence model table. The first column contains sentence models. The second column describes the weight of this row compared to others in the table. S selects rows with higher weights proportionately more often than rows with lower weights. *The selection of a row in the sentence model table is independent of C.*

The example sentences in Table 16 correspond to $NICETEXT_{d,s}(c)$ where d is the dictionary in Table 13, s is the corresponding sentence model in the previous table, and c is the bit string next to each sentence. The purpose of the example is to show how different models vary the style of the innocuous text with the same input.

It is good to have a large number of sentence models to add variety to the generated text. Section 4.3 describes one method to easily construct large sentence model tables from sample texts.

There are many ways to define meta-style-sources that use tables of sentence models. For example, the source could use the mutual information [8] between sentence models to drive the sequence rather than a flat frequency distribution. This means that there may be some relationship between sentence models. The relationship could be a statistical correlation or it could be a fixed order of sentence-models that make up a "paragraph" (or "chapter" or "book"...).

The primary characteristic of a sentence model table is that there are a fixed number of models from which to choose. At run-time the style source simply chooses existing models from the table.

Sentence Model	Weight
{Cap} person verb {Cap} person art object {.}	3
{Cap} person verb art object prep Cap person {.}	2
{Cap} perp {Cap} person {CAPSLOCKON} conj {capslockoff} {?}	1

Table 15: An Example *Sentence Model Table.*

Ciphertext, C	$NICETEXT_{D,S}(C)$ for different S's
1000100	Heather sent Lisa the money.
11111000	Bob gave the gift to Shirley.
0001	To Shirley AND Lisa?

Table 16: Sample Sentences Corresponding to the Models Table 15.

4.2 Context-Free-Grammars

A Context-Free-Grammar (CFG) meta-style-source, S, generates valid sentence models on demand during the $NICETEXT_{D,S}$ processing. The variety of generated sentence models from a CFG style-source is usually much larger than a similar-length static sentence model table.

Context-Free-Grammars define language syntax [1, 15]. Although normally used for parsing they also can generate syntactically correct sentences in many languages – natural or otherwise. The CFG meta-style-sources define the "languages" of sentence models. The generated sentence models perform the same function as in the previous sections. The major difference is that the CFG source dynamically creates the sentence models during an application of $NICETEXT_{D,S}$.

A CFG consists of a set of rewrite rules. The Left-Hand-Side (LHS) represents the token to be replaced by the set of tokens on the Right-Hand-Side (RHS). For example, the rule $SENTENCE \longrightarrow NOUNPHRASE\ VERBPHRASE$ implies that a $NOUNPHRASE$ followed by a $VERBPHRASE$ compose a $SENTENCE$. If there are several ways to generate a $SENTENCE$ then there will be multiple rules with $SENTENCE$ on the LHS. Each rewrite rule for a LHS may have a *weight* which allows the style source to generate according to the relative priority of all possible RHS's. The RHS tokens, such as $NOUNPHRASE$, may be the LHS of at least one other rule. If not, the RHS tokens are called *terminals*.

Terminals are the basic building blocks of the language defined by the grammar. In the case of a programming language, the terminals might be keywords, constants, and variable names [1, 15]. The CFG meta-style-sources use word type categories and punctuation rules as terminals. *The grammar generates* sentence models*, not sentences!* $NICETEXT_{D,S}(C)$ uses the sentence models to create sentences. This approach allows $NICETEXT$ to generate innocuous text according to the rules of a CFG. $SCRAMBLE_D$ functions without any knowledge of the grammatical structure; thus, the complexity of the grammar does not affect the efficiency of $SCRAMBLE$.

The well-known parser-generating tool, YACC, inspired the syntax of the $NICETEXT$ grammar definition [15]. The main differences are the *@-weight* options for each RHS and the use of C++-style single-line comments. Figure 7 is a simple example of a grammar. Table 17 shows three example sentence models that follow these grammar rules. Figure 8 shows several example sentence generated by $NICETEXT$ using the same CFG style-source.[1]

The first rule in the grammar definition file represents the starting point for sentence model construction. The order of the remaining rules is not significant. The CFG meta-style-source uses the weight assigned to each Right-Hand-Side (RHS) to select the rewrite rules using a pseudo-random source.

4.2.1 Generation of a Sentence Model from a CFG

Figure 9 shows one expansion of the *SENTENCE* rule defined in the example grammar of Figure 7. This section walks through the generation process.

First, the CFG source, S, begins with the first rule in the grammar definition file. The LHS of the primary rule is *SENTENCE*. *SENTENCE* has only two RHS rules from which to choose. There is a 27 out of 28 chance that S selects the first RHS. For this example, S does indeed select the first RHS. Thus, S generates the terminal, {*Cap*} and the token, *REST-OF-SENTENCE*. $NICETEXT$ appends {*Cap*} to the sentence model. $NICETEXT$ appends all other terminals in-order while traversing the tree.

The *REST-OF-SENTENCE* rule has two possible RHS's from which to choose

[1] The new-lines have been removed from Figure 8 to preserve space.

with probability 27/46 and 19/46 respectively. For this example S selects the second RHS rule to rewrite *REST-OF-SENTENCE*. This generates the terminal *prep*, the token, *PEOPLE*, the terminal {,}, the token *PEOPLE*, the terminal *verb*, the token *OBJECTS*, and the token *END-OF-SENTENCE*. $NICETEXT$ appends the terminal *prep* to the sentence model.

The *PEOPLE* token has three RHS rules with probability of 143/213, 53/213, and 17/213. In this case, S selects the first RHS, which is the token *PERSON*. The style source expands *PERSON* into the two terminals {*Cap*} and *person*. $NICETEXT$ appends these two terminals to the sentence model. Next, S returns to the third RHS element for rewriting *REST-OF-SENTENCE*. $NICETEXT$ appends the terminal {,} to the model.

The fourth RHS token for rewriting *REST-OF-SENTENCE* is *PEOPLE*. This time, S follows a different RHS for *PEOPLE* to generate the tokens *PERSON*, *conj*, and *PERSON*. The first *PERSON* expands to the terminals {*Cap*} and *person*. The token *conj* is a terminal. The second *PERSON* also expands to {*Cap*} and *person*. Therefore, the style source rewrites the fourth token for *REST-OF-SENTENCE* as the five terminals: {*Cap*}, *person*, *conj*, {*Cap*}, and *person*.

The fifth RHS token for *REST-OF-SENTENCE* is the terminal *verb*.

The sixth RHS token for *REST-OF-SENTENCE* eventually expands to the five terminals: *art*, *object*, *conj*, *art*, and *object* much the same way *PEOPLE* expanded for the fourth token.

Finally, the style source rewrites the *END-OF-SENTENCE* token as the terminal {.n}. $NICETEXT$ appends the terminal to the model. The resulting model has seventeen instructions that $NICETEXT_{D,S}$ uses like any other sentence model style source. When $NICETEXT$ exhausts this model, the CFG meta-style-source will follow a similar process to generate a new sentence model.

4.2.2 Dealing with Merged Types: *expgram*

Sometimes a style-source requires a specific merged-type category such as $object,person$. Other times S needs a more general way to specify all types that relate to a single type such as all types that have $person$ as a sub-type. The *expgram* program automatically generates the *m-rules* to specify all types (merged or otherwise) related

```
// A sample grammar for the NICETEXT system
// The numbers after the @ represent the weight of each RHS
// The {}'s represent command-tokens for capitalization
// and punctuation in the generated sentence model
// The first rule in the file is the primary rule
// Note: all symbols are case-sensitive.

SENTENCE: {Cap} REST_OF_SENTENCE          @27 // cap first letter
   |  {CAPSLOCKON} SENTENCE {capslockoff} @1  // CAP WHOLE SENTENCE
   ;

REST_OF_SENTENCE:
       PEOPLE verb OBJECTS prep PEOPLE END_OF_SENTENCE      @27
   |   prep PEOPLE {,} PEOPLE verb OBJECTS END_OF_SENTENCE  @19
   ;

END_OF_SENTENCE:  {. n}       @9 // period, space, new-line
   |   conj REST_OF_SENTENCE  @1 // compound sentence
   ;

PEOPLE:
       PERSON                            @143 // one person
   |   PERSON conj PERSON                @53  // p and p
   |   PERSON {,} PERSON {,} conj PERSON @17  // p, p, and p
   ;

PERSON: {Cap} person; // first letter uppercase for a "person"

OBJECTS: OBJECT                          @253 // "the x"
   |  OBJECT conj OBJECT                 @212 // "tx and tx"
   |  OBJECT {,} OBJECT {,} conj OBJECT @209 // "tx, tx, and tx"
   ;

OBJECT:  art object; // article followed by an object "the x"
```

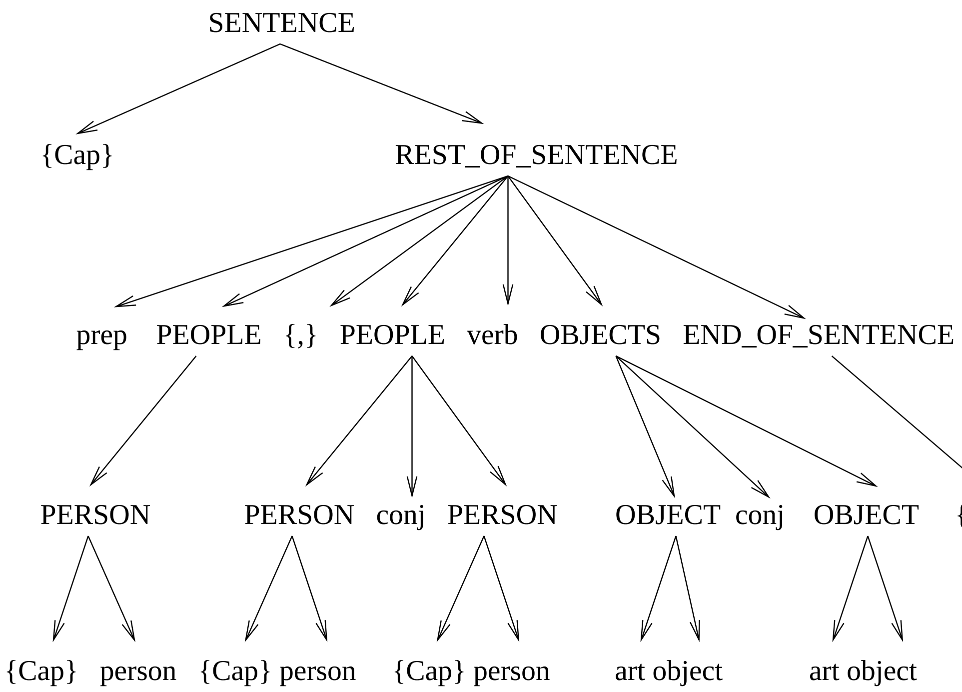
Figure 7: Sample *NICETEXT* Grammar Definition

{Cap} prep {Cap} person {,} {Cap} person conj {Cap} person verb art object conj art object {.n}
{Cap} {Cap} person verb art object conj art object perp {Cap} person conj {Cap} person {.n}
{Cap} {Cap} person verb art object prep {Cap} person conj {Cap} person {.n}

Table 17: Sample Sentence Models from the CFG in Figure 7.

Heather gave the money and the mail to Lisa. Bob sent the money, the gift, and the message to Bob and Lisa and to Bob and Heather, Bob and Bob gave the mail, the mail, and the gift. Bob, Heather, and Lisa gave the gift and the money to Lisa, Lisa, and Lisa. To Heather, Heather gave the gift. Shirley and Heather gave the money, the message, and the message to Heather. To Heather and Shirley, Heather gave the money. Lisa and Heather sent the money to Heather. To Bob, Shirley sent the money, the mail, and the gift. Bob sent the mail to Heather. Shirley, Shirley, and Shirley gave the mail and the gift to Heather. Shirley and Shirley sent the money, the message, and the message to Lisa and Heather. To Lisa, Heather, Lisa, and Shirley gave the message and the gift. To Heather, Lisa sent the gift, the message, and the money. BOB, SHIRLEY, AND LISA GAVE THE GIFT AND THE GIFT TO SHIRLEY. To Bob, Bob and Shirley sent the money. Heather gave the mail, the gift, and the message to Bob. Bob and Heather sent the message to Heather and Bob. Lisa, Shirley, and Heather gave the gift, the money, and the mail to Bob and Lisa. Shirley and Bob sent the message to Bob. Bob gave the mail and the message to Heather and Heather.

Figure 8: Sample $NICETEXT$ Sentences from the CFG in Figure 7.



GENERATED MODEL:

{Cap} prep {Cap} person {,} {Cap} person conj {Cap} person verb art object conj art object {.n}

Figure 9: Sentence Model Generation Example.

```
mOBJECT:
        object         @4 // an mOBJECT can be an "object"
    |   object,person @1 // or an "object,person"
    ;

mPERSON:
        person         @4 // an mPERSON can be a "person"
    |   object,person @1 // or an "object,person"
    ;
```

Figure 10: Small Sample *M-RULE* From *expgram*

to $mTYPE$.

The grammar in Figure 7 does not include the $object, person$ type category. $NICETEXT$ would not include "bill" in any generated text because the word "bill" is neither an $object$ nor a $person$. Figure 10 is an example set of *m-rules* that enable a grammar to include the $object, person$ type as part of the $mOBJECT$ and $mPERSON$ rules. The user must append the *m-rules* from *expgram* to each a grammar definition file.

The *expgram* program has two options for generating the weights of each RHS of an *m-rule*. The default option is to simply assign the weight to the count of the number of words of that type in the dictionary. This causes the style-source to select the merged-type disproportionately overall because the type appears in the RHS of as many rules as it contains types.

The *normalized* option to *expgram* divides the frequency of each merged type by the number of sub-types. For example, if the merged type on the RHS were $object, place, person$ and there were nine words of that type in the dictionary then the weight of that RHS would be three instead of nine. This helps to more evenly distribute the weight of the type $object, place, person$ as the RHS of the three m-rules: $mOBJECT$, $mPLACE$, and $mPERSON$. The assumption is that $mOBJECT$'s, $mPLACE$'s, and $mPERSON$'s equally value the $object, place, person$ RHS.

The format of the grammar definition file makes it easy to assign weights to merged types. The purpose of the *expgram* program is to automate the process of creating *m-rules* from a dictionary table. Figure 11 shows one set of m-rules for a rather sophisticated dictionary.

4.2.3 Testing a Grammar: *gramtest*

The *gramtest* program checks the syntax of a grammar definition file by reading the rules into an *MTCgrammar* object. [2] The program flags any syntax errors, dumps the grammar rules from the grammar object, and it generates sample sentence models in a format easily read by the user.

Figure 12 is rule listing from the application of *gramtest* to the example grammar in Figure 7. Although the format resembles the syntax for the grammar definition, it contains different information. The *gramtest* program lists punctuation rules and terminals in addition to the rules. This is useful for debugging a complex grammar. It converts the *@-weights* from the grammar definition file to a *#-cutoff* number representing a running-total of the weights for the RHS's of each rule. Internally, the *MTCgrammar* object randomly chooses a number up to the total *@-weights* of RHS. The *#-cutoff* numbers partition the range so that the *MTCgrammar* object can efficiently pick the RHS corresponding to the random number. For example, to choose a RHS for the *OBJECTS* rule, *MTCgrammar* chooses a number between 1 and 674. If the number is greater than or equal to 253 but less than 465 then it selects the second RHS.

The *gramtest* program also generates sample sentence models such as in Figure 18. (The "{e 0}" signifies the end-of-sentence with a weight of zero.) It is sometimes helpful when debugging complex grammars to see the internal structure. The *gramtest* program provides this information.

4.3 Style by Example

This section describes the *genmodel* process to generates style sources that simulate certain aspects of writing style found in sample natural language texts.

It is cumbersome to manually construct large sentence model tables. A modified version of the lexical scanner in *SCRAMBLE* automatically generates these tables

[2] The *MTCgrammar* object is the same one that the *nicetext* program uses as a style-source.

```
mPLACE:
        place @86
    |   AJ_AbsVerbal,N_3sg+Sg,place @1
    |   AJ_AbsVerbal,N_3sg+SgProp+Verbal,name_male,place @1
    |   AJ_AbsVerbal,N_3sg+SgProp+Verbal,place @1
    |   AJ_AbsVerbal,N_3sg+SgProp+Verbal,place,vowel @1
    |   AJ_Prop+AbsVerbal,N_3sg+SgProp+Verbal,name_male,place @1
    |   AJ_Prop+AbsVerbal,N_3sg+SgProp+Verbal,place @1
    |   N_3sg+Sg,N_3sg+SgProp+Verbal,place,vowel @1
    |   N_3sg+Sg,place,vowel @2
    |   N_3sg+SgProp+Verbal,N_3sg+SgProp-Verbal,place @3
    |   N_3sg+SgProp+Verbal,V_BaseFin,name_family,place,vowel @1
    |   N_3sg+SgProp+Verbal,name_family,name_male,place @8
    |   N_3sg+SgProp+Verbal,name_family,name_male,place,vowel @1
    |   N_3sg+SgProp+Verbal,name_family,name_other,place @2
    |   N_3sg+SgProp+Verbal,name_family,place @15
    |   N_3sg+SgProp+Verbal,name_family,place,vowel @7
    |   N_3sg+SgProp+Verbal,name_female,name_male,place @1
    |   N_3sg+SgProp+Verbal,name_female,place @6
    |   N_3sg+SgProp+Verbal,name_female,place,vowel @3
    |   N_3sg+SgProp+Verbal,name_male,place @5
    |   N_3sg+SgProp+Verbal,name_male,place,vowel @1
    |   N_3sg+SgProp+Verbal,name_other,place @2
    |   N_3sg+SgProp+Verbal,name_other,place,vowel @1
    |   N_3sg+SgProp+Verbal,place @143
    |   N_3sg+SgProp+Verbal,place,vowel @41
    |   N_3sg+SgProp-Verbal,place @4
    |   N_3sg+SgProp-Verbal,place,vowel @2
    |   name_family,name_male,name_other,place @1
    |   name_family,name_other,place @1
    |   name_family,place @1
    |   name_female,place,vowel @1
    |   name_male,place @2
    |   name_other,place @2
    |   place,vowel @21
    ;
```

Figure 11: Larger Sample *M-RULE* From *expgram*

```
// DUMPING 18 GRAMMAR RULES in alphabetical order
// (#'s are adjusted weights used to select a rule
// by the previous @ sum.)

{,}:          (punctuation rule only);
{. n}:        (punctuation rule only);
{CAPSLOCKON}: (punctuation rule only);
{Cap}:        (punctuation rule only);

END_OF_SENTENCE: {. n}       #9
   |   conj REST_OF_SENTENCE #10;

OBJECT: art object #1;

OBJECTS:   OBJECT                      #253
   |   OBJECT conj OBJECT              #465
   |   OBJECT {,} OBJECT {,} conj OBJECT #674;

PEOPLE: PERSON                         #143
   |   PERSON conj PERSON              #196
   |   PERSON {,} PERSON {,} conj PERSON #213;

PERSON: {Cap} person #1;

REST_OF_SENTENCE:
       PEOPLE verb OBJECTS prep PEOPLE END_OF_SENTENCE     #27
   |   prep PEOPLE {,} PEOPLE verb OBJECTS END_OF_SENTENCE #46 ;

// NEXT RULE IS BASE RULE * * *
SENTENCE:    {Cap} REST_OF_SENTENCE #27
   |         {CAPSLOCKON} SENTENCE {capslockoff} #28 ;

art:           terminal(1);
{capslockoff}: (punctuation rule only);
conj:          terminal(2);
object:        terminal(3);
person:        terminal(5);
prep:          terminal(6);
verb:          terminal(7);
// END OF GRAMMAR RULE DUMP
```

Figure 12: Rule Listing From *gramtest*

{Cap} {Cap} person verb art object prep {Cap} person conj {Cap} person verb art object prep {Cap} person {,} {Cap} person {,} conj {Cap} person {. n} {e 0}
{Cap} prep {Cap} person , {Cap} person verb art object {. n} {e 0}
{Cap} prep {Cap} person conj {Cap} person {,} {Cap} person verb art object {. n} {e 0}
{Cap} {Cap} person verb art object conj art object prep {Cap} person {. n} {e 0}
{Cap} prep {Cap} person conj {Cap} person {,} {Cap} person conj {Cap} person verb art object {. n} {e 0}
{Cap} {Cap} person conj {Cap} person verb art object conj art object prep {Cap} person conj {Cap} person {. n} {e 0}
{Cap} {Cap} person verb art object prep {Cap} person {. n} {e 0}
{Cap} prep {Cap} person {,} {Cap} person verb art object {. n} {e 0}
{Cap} prep {Cap} person {,} {Cap} person conj {Cap} person verb art object {,} art object {,} conj art object {. n} {e 0}

Table 18: Sample Models from *gramtest*

from example natural language texts. It is not a new idea to try to mimic the part-of-speech and punctuation sequences of natural language text. The application of this technique towards the $NICETEXT$ and $SCRAMBLE$ functions allows the software to be quite "trainable".

Let $GENMODEL_D(SAMPLETEXT) \longrightarrow S$ be a function that generates a *sentence model table* S from a file, $SAMPLETEXT$, containing natural language sentences. $SAMPLETEXT$ has case-sensitive variants of words in the dictionary D with punctuation. The *genmodel* program makes sentence model tables from example text. The table S is style source for $NICETEXT_{D,S}$.

Let d be the dictionary in Table 13. Let $SAMPLETEXT$ be the following example text: "Bob gave the mail to Shirley. Heather sent Bob the gift. Bob sent the gift to Heather. Lisa gave Bob the money. To Bob AND Heather? Shirley gave Lisa the money." In this case, $GENMODEL_d(SAMPLETEXT) \longrightarrow S$ where Table 15 depicts S.

The $SCRAMBLE$ program parses individual words from the innocuous text T. It is a trivial amount of *Lex* code that returns each word as a token while ignoring the punctuation and case [15]. The modified word scanner in *genmodel* returns punctuation, white-space, and words as tokens. A simple function reads the case of each word to generate the commands: {Cap}, {CAPSLOCKOFF}, and {capslockoff}. Like all sentence models, the {} notation specifies punctuation and white space. The dictionary defines the type of each word. The type becomes part of the sentence model. The scanner also detects the end of a sentence. The program places each sentence model into the table. The number of times each model occurs in $SAMPLETEXT$ is the weight of the model in the table.

The *genmodel* program has two additional features. The first feature outputs all the sentence models as weighted RHS's of one rule in a grammar definition file. The purpose is to make the style source very easy to view and edit. The disadvantage is that the CFG style-sources work more efficiently with many small rules rather than one big rule. The second feature outputs a *distribution dictionary*, D'.

D' is the intersection of the set of words in D and in $SAMPLETEXT$. The purpose is to better simulate word usage within each type category. A "large and sophisticated" dictionary may have hundreds of thousands of unique words. Figure 1 shows that out of 916,151 words in *The Complete Works of William Shakespeare* only 28,254 are unique. The style-source output by the *genmodel* program correctly simulates the type sequences of words in each sentence model. If $NICETEXT_{D,S}$ uses the master dictionary then T will not sound much like Shakespeare because of the many irrelevant words in each type. All style sources compatible with D', including S, are also compatible with D. Not all style sources compatible with D are compatible with D' because D' is a subset of D that may have zero words of some types.

The key to the success of *genmodel* is to have a very sophisticated dictionary. D must have a large percentage of the words in $SAMPLETEXT$. Perhaps at dictionary construction time the *listword* program used $SAMPLETEXT$ to create $WLIST$. The type categories in D must be very specific or else the sentences made from the models will not have very good word agreement.

4.4 Example *genmodel*

Let $SAMPLETEXT$ be President JFK's Inaugural Address (only an excerpt shown here):

> We observe today not a victory of party but a celebration of freedom. . . symbolizing an end as well as a beginning. . .signifying renewal as well as change for I have sworn before you and Almighty God the same solemn oath our forbears prescribed nearly a century and three-quarters ago.
>
> The world is very different now, for man holds in his mortal hands the power to abolish all forms of human poverty and all forms of human life. And yet the same revolutionary beliefs for which our forbears fought are still at issue around the globe. . .the belief that the rights of man come not from the generosity of the state but from the hand of God. We dare not forget today that we are the heirs of that first revolution.
>
> (...)

Let d be a distribution dictionary composed of 513 unique words used in JFK's speech categorized into 333 types. $GENMODEL_d(sampletext)$ creates a sentence model table, s, with 56 sentence models. Let c be the following ciphertext as shown in hexadecimal:

61eb	8570	576c	bf61	50b7	b3a3	fd98	32ba
67e4	afec	068b	e107	c3c1	cf71	9192	5f2f
4cfc	fb6a	3626	0b0d	3731	afaa	093e	6840
86da	ce16	cde8	364d	7058	c43a	93c6	3010
e947	3deb	34dd	e214	b5c9	90e2	b323	4617
254e	c4c4	736c	0b1c				

The following are two examples of $NICETEXT_{d,y}(c) \longrightarrow t_i$:

- $t_1 =$

> My area origins of the suspicion... oppose much what America will be before you, before what asunder we would be for the Poverty inside Man. Yet will it do almighty off the first two south votes... or on the course at this administration, whether even deadly off our suspicion by this peace. To those young votes what proud nor alike origins we comfort: we house the heritage past hostile americans. As this colonial testimony past hope cannot strengthen the sign at proud powers. To our loyalty heirs sovereign past our torch: we observe every special issue... to tiger our ago arms save good republics... by a powerful renewal but subject... to generation free men nor free stays on beginning in the americas inside science. On the young ocean from the life, only a many allies are been prescribed the course inside signifying find on its hardship around subject foe; I be not abolish at this peace... I issue it. By your heirs, my tyranny communists... more inside mine... will power the sufficient friend nor heritage past our beachhead. Shall we shield from they nations a weak whether spiritual renewal... West if Ancient... Mortal because West... that would know every more solemn absolute but all mankind? To those people in the allies and friends save half the fellow becoming to science the origins up quest misery: we shield our best forms to burden them age themselves, for what belief has divided... much whether the Americas may be doing it, not because we offer their americans, as only it has

remain. And when, my break Heirs... let much what your world would be before you... come whose you would do as your life. I be much forget that any past us must still hands around any other people because any other alliance. We price not spread them save assist. We price not take area that we have the nations save that first progress. The graves around national Origins who planned the witness to host go the science. For request us run. Observe all our nations strengthen that we can remember of them to own invective nor state anywhere off the Deeds... yet remember a little host right that this forum holds to right the assist save its oppose witness.

- $t_2 =$

  Required... there is other we shall be... before we dare not remember every misery form, save men, only atom strongly. Expect few powers, but the first iron, strengthen peaceful yet bitter chains but the inspection only negotiate inside adversaries... if oppose the home welcome to begin other words of the year remember inside all votes. Join few standards let to host on all weapons inside the hardship the sign at Isaiah... to own the same hands... right the planned meet free. Since this freedom was planned, each renewal up Americas has been planned to negotiate belief to its national globe. Since this hemisphere was present, each assist save Stays has been sought to remain life to its young subject. If when, my belief Votes... run not which your tyranny shall do for you... begin whatever you shall be but your suspicion. For this poor hardship save hope cannot meet the age at different powers. Observe few problems become whatever standards control us very past signifying those standards whose remain us. Make many problems, but the first prevent, strengthen alike because sufficient wishes for the inspection yet request of groups... whether come the loyalty shield to assure other beliefs of the hand come at all stays. This not we mass... only more. Outpaced... there is little we must be... for we price much make a colonial witness, at odds, because lifetime finally. My belief chains up the lifetime... renew much which

> America will do but you, as which merely we can be but the Absolute from Bear. This not we view... because more. Remain every world make... and it words us well and ill... that we can peaceful any dare, man any doubt, request any lifetime, war any state, seek any hemisphere, to take the renewal and the friend around oath. Run few sides, as the first quest, strengthen faithful because accidental arms for the administration yet run around neighbors... because remain the century age to request little allies at the writ tempt save all heirs. We can much anew request to nation them creating our view.

$SCRAMBLE_d(t_i) \longrightarrow c$ recovers the ciphertext c for both examples because $NICETEXT_{d,s}(c)$ used the same c (and the same dictionary) as input. The eighty-eight bytes of c expanded into around two-thousand byes for each t_i because d has very few words of each type.)

The key to the success of a $GENMODEL_D(SAMPLETEXT)$ function is the dictionary, D. If words in the example text, $SAMPLETEXT$, are not in D then *genmodel* discards the whole sentence. If the type breakdown of D is not specific then $NICETEXT_{D,S}$ generates sentences with no word agreement. (i.e. If two types are "noun" and "verb" then $GENMODEL_D(SAMPLETEXT)$ could not distinguish singular vs. plural; thus, $NICETEXT_{D,S}$ would output sentences where the plurality of the nouns and the verbs would not agree.)

Chapter 5

Results and Conclusions

The $NICETEXT$ system transforms ciphertext into natural-language text and the $SCRAMBLE$ system recovers the ciphertext from the innocuous text. The purpose is to thwart the censorship of ciphertext by making it look like something else or by providing a plausible reason for transmitting unintelligible data.

The quality of the innocuous text highly depends on the construction of sophisticated code dictionaries. These dictionaries categorize words by type and maintain certain properties to ensure $SCRAMBLE$ is always the inverse of $NICETEXT$.

Style-sources add structure and formatting to the generated text. A sophisticated style-source requires a sophisticated dictionary. $NICETEXT$ uses the style-source to make more "interesting" innocuous text. $NICETEXT$ uses the ciphertext to choose the exact words of each type from the dictionary.

Several tools facilitate the generation of sophisticated dictionaries and style-sources. A dictionary may have many compatible style-sources and a style-source may have many compatible dictionaries. Even with the same dictionary and the same style-source different applications of $NICETEXT_{D,S}(C)$ can generate many different texts. Given all of this, $SCRAMBLE_D(NICETEXT_{D,S}(C)) = C$ for all C, for all D, and for all S.

The expansion of the length of the innocuous text T with respect to the length of the ciphertext C is the "cost-of-style". The three aspects to style are capitalization, punctuation, and type selection. It does not add the length of T to choose upper versus lower case; thus, the "cost-of-style" for capitalization is zero. The cost for punctuation relates to the number of bytes of punctuation and white-space compared to the number of types in a sentence model. More punctuation increases the growth rate from ciphertext to English. The expansion rate for type selection depends on the dictionary. If there are many words of a single type then that type requires more bits of ciphertext to choose a word. If the style-source chooses types with more words of each type then the expansion rate decreases. One other factor is the average length of words within a type. If there are words with many letters within a type category then the expansion rate increases.

One question that has not been answered is whether or not the *NICETEXT* system itself is *cryptography*. The short answer is no. If anything it is a weak substitution cipher. The results are not cryptographically secure and are easy to break without the dictionary tables.

There are several ways to improve the utility of the *NICETEXT* system. For example, it is very difficult to identify nice-text when it is merged with some actual natural language text. A simple post-processor to *NICETEXT* can mix the generated sentences with another source. A pre-processor to *SCRAMBLE* could extract the nice-text from the larger and much more believable text. If the censor discovers the innocuous text generated by *NICETEXT* then the sender simply claims that *NICETEXT* is an automated "ghost-writer". *NICETEXT* just fills-in the gaps!

We have presented a system that transforms ciphertext into innocuous text. The tools provide a high level of control over the quality of the generated text. The software is available for most systems with an ANSI C++ compiler at *http://www.ctgi.net*. The license is for personal educational use only, not by government employees, or corporations. The software may not be sold without permission from the authors.

Appendix A

Program Documentation

A.1 Dictionary Definition

A.1.1 Using *dct2mstr*

```
Usage: dct2mstr -i inputFile [-d dictionary]
```

The *dct2mstr* program inputs a type-word list from text file where each line consists of a *type* description followed by a *word*. The type-word list must be the output of the *sortdct* program found in section A.1.7. The output is a master dictionary database for the *nicetext* and *scramble* programs.

A.1.2 Using *impkimmo*

```
Usage: impkimmo
```

The *impkimmo* program reads the output of the *pckimmo* software package from standard input. It outputs a type-word list with one entry corresponding to a single parse tree from the *pckimmo* program.

The *pckimmo* program and the *englex* word grammar is available from *http://www.sil.org/pckimmo/*. Figure 13 lists the settings for *pckimmo* that produce parse-trees that *impkimmo* recognizes. The input to *pckimmo* is a word-list.

A.1.3 Using *impmsc*

```
Usage: impmsc
```

```
load rules english.rul
load lexicon english.lex
load grammar english.grm
set alignment OFF
set ambiguities 10
set failures OFF
set features TOP
set features FULL
set glosses OFF
set grammar ON
set limit OFF
set rules ON ALL
set timing OFF
set tracing OFF
set tree OFF
set trim-empty-features ON
set unification ON
set verbose OFF
set warnings OFF
recognize word.list
```

Figure 13: Settings for *Pckimmo* to Work With *Impkimmo*

The impmsc program converts a word-type list into a type-word list. It is almost the same as the awk [24] command: *awk '{print $2 " " $1}'*.

A.1.4 Using *impwbstr*

```
Usage: impwbstr
```

The *impwbstr* program reads the output of the on-line Websters Dictionary from standard input. It outputs a type-word list with one type-word pair corresponding to each part-of-speech definition.

The *webster* program is available on many NextStep systems.

A.1.5 Using *listword*

```
Usage: listword [-c]
       [-c] Count Words and Print Frequency on Output
```

The *listword* program extracts words from standard input using the scanner from *scramble*. It outputs the sorted list to standard output. The -c option causes listword to output the number of occurrences of each word.

A.1.6 Using *printint*

```
Usage: printint
```

The printint program prints a list of numbers in sequential order to standard output. The purpose is to use these numbers as words in the dictionary. The output of printint often is redirected to a file named *num_cardinal_digits* for expansion by the txt2dct program.

A.1.7 Using *sortdct*

```
Usage: sortdict [-x] [-e errorLevel] [-u updateFreq] [-q] [-r]
    -e   Print error messages up to this level (0-9). Default is 1.
    -q   Do NOT print status updates or errors ([-u 0] [-e 0])
    -r   Rerun option: input should be the output of previous run.
```

```
        (commas are interpreted as merging types).
    -u  Print status update every 'updateFreq' successful lines
    -x  Expand dictionary with suffixes (experimental)
```

The *sortdct* program reads lines from standard input where each line consists of a *type* description followed by a *word*. It merges the types of words belonging to more than one type category. The order of the entries in the input type-word list does not matter. The *sortdct* program excludes any words that *scramble* will not recognize. The output type-word list prints to standard output and is ready for processing by the *dct2mstr* process.

The obsolete -x option automatically expands the types of each word. It does this by blindly appending suffixes such as "ing" and "s". It then generates the appropriate new types. For example, the word "bang" of type "noun" would get expanded to "banging" of type "noun-GERUND" and the word "bangs" of type "noun-PLURAL". This quick-and-dirty option is not very effective because it creates many non-words and incorrect types. The *listword* and *impkimmo* programs eliminate the need for the -x option in *sortdct*.

A.1.8 Using *txt2dct*

```
Usage: txt2dct file1 [file2] [file3] [file4] [...]
```

The *txt2dct* program reads lines of text from each input file and outputs the name of the file and the contents of each line to standard output. The purpose is to create a type-word list that is suitable for use with the *sortdct* program. Each input file should contain one word per line. The name of the file becomes a type for all words in the file. The file names should not contain commas or spaces. This program does not attempt to decide if a word is suitable for a type-word list. (The *sortdct* program will eliminate any bad words when sorting and merging the type-word list.)

The only filtering txt2dct does is to skip any lines that begin with a # sign.

A.1.9 Using *vowel.sh*

```
Usage: vowel.sh
```

The *vowel.sh* shell-script launches the *awk* [24] program *vowel.awk*. The purpose is to extract all words that start with a vowel from a master dictionary table. These words are placed into a file named “vowel”. The *txt2dct* process can use that file to add the merged type of *vowel* to all words in a new dictionary table.

The reason for doing this is to have agreement with “a” and “an” during English text generation. The *genmodel* program benefits from *vowel.awk*.

A.2 Grammar Definition

A.2.1 Using *dumptype.sh*

```
Usage: dumptype.sh
```

It is useful to see examples of each type of word when manually creating grammars. Unlike the [-s] option to *expgram*, the *dumptype.sh* script immediately dumps three examples of each type of word in a dictionary to standard output. The *dumptype.sh* shell script calls *dumptype.awk* to create a grammar definition file, *dumptype.def*. The *nicetext* program uses *dumptype.def* as a style source to dumps three example words for each type the dictionary.

The output of *nicetext* is not suitable for recovery with *scramble* because it treats the type label for a category as punctuation by using the `{^quoted-punctuation^}` syntax. (If no type categories can be mistaken for words then *scramble* could recover the ciphertext.) It is not recommended to use *dumptype.def* as a regular style-source. The sole purpose of *dumptype.def* is to print some example words of each type in a dictionaryu.

A.2.2 Using *expgram*

```
Usage: expgram -d dictionary [-o outputFile] [-s sampleFile] [-n]
    -d    dictionary prefix
    -o    redirect from stdout
    -s    create a sample grammar (use dumptype.sh instead...)
    -n    ‘normalize’ frequency based on the number of types
```

The *expgram* program automatically generates the *m-rules* to specify all types (merged or otherwise) related to $mTYPE$. The author of a grammar appends the output of *expgram* to an existing grammar definition file.

The [-s] option causes the program to also generate a special-purpose grammar. The purpose of the grammar is to have *nicetext* generate three example words of each merged-type category. The sample grammar should not be used to generate text for recovery by *scramble* because it incorporates the `{^quoted-punctuation^}` syntax. The *dumptype.sh* script makes the [-s] option obsolete.

The *expgram* program has two options for generating the weights of each RHS of an *m-rule*. The default option is to simply assign the weight to the count of the number of words of that type in the dictionary. This causes the style-source to select the merged-type disproportionately overall because the type appears in the RHS of as many rules as it contains types.

The *normalized* option to *expgram* divides the frequency of each merged type by the number of sub-types. For example, if the merged type on the RHS were $object, place, person$ and there were nine words of that type in the dictionary then the weight of that RHS would be three instead of nine. This helps to more evenly distribute the weight of the type $object, place, person$ as the RHS of the three m-rules: $mOBJECT$, $mPLACE$, and $mPERSON$. The assumption is that $mOBJECT$'s, $mPLACE$'s, and $mPERSON$'s equally value the $object, place, person$ RHS.

A.2.3 Using *genmodel*

```
Usage: genmodel [-s]
   -s   Do not load mstrdict into RAM - read it as needed from disk
```

The *genmodel* program generates style sources that simulate certain aspects of writing style found in sample natural language texts. The primary style-source is the *mstrmodel* sentence model table. Another style-source that *genmodel* creates is the grammar defition file *grambase.def*. The purpose of *grambase.def* is to make the style source very easy to view and edit. The disadvantage is that the CFG style-sources work more efficiently with many small rules rather than one big rule.

Another feature is that *genmodel* outputs a *distribution dictionary*. This dictionary contains the words from the master dictionary table, *mstr*, that appear in the sample text. The *-d dist* option forces *scramble* and *nicetext* to use the *distribution dictionary*.

The [-s] option causes *genmodel* to save memory by reading the master dictionary from disk while processing. This option greatly slows down the time it takes to process large sample texts.

The statistics output by *genmodel* during processing are:

SR: Number of Sentences Read

US: Number of Unique Sentences Read

WR: Number of Words Read

UW: Number of Unique Words Read

BWR: Number of Bad Words Read (words not in dictionary)

UBW: Number of Unique Words Read

A.2.4 Using *gramtest*

```
Usage: gramtest [-g grammarFile] [-d dictionary] [-c sampleCount]
    -g GrammarFile -- the name of the grammar definition file.
       (defaults to grammar.def)
    -d Dictionary -- the name of the dictionary prefix
       (defaults to 'mstr')
    -c SampleCount -- print this number of sample sentence models
```

The *gramtest* program reads a grammar definition file into an *MTCgrammar* object. The purpose is to debug a grammar definition file. The output of *gramtest* is a dump of the grammar rules. The [-c] option forces gramtest to generate a number of sample sentence models in a format that humans can easily read.

A.3 Transformation Programs

A.3.1 Using *nicetext*

```
Usage: nicetext -i inputfile [[-g grammarDefFile] | [-m model]]
```

```
[-d dictionary] [-o outputFile] [-s] [-l maxModelLength]
[[-u updateFreq] | [-q]]

 -d   Specify the prefix for the dictionary file
      (i.e. mstr for mstrdict.dat, mstrtype.dat)
 -g   Specify the full grammar definition file path
 -i   Specify the full input file path
 -l   Do not use sentences with more than this # of components
 -m   Specify the prefix for the model file
      (i.e. mstr for mstrmodel.dat)
 -o   Specify the full output file path (default to stdout)
 -q   Do not print status updates. (same as [-u 0])
 -s   Small mode - use tables from disk, don't load into RAM
 -u   Print status updates every 'updateFreq' sentences.
```

The *nicetext* program converts the input file into innocuous text containing natural-language sentences. The style-source is either a grammar defintion file, [-g], or a sentence model table, [-m]. The dictionary prefix, such as *mstr*, is specified with [-d].

The [-s] option causes *nicetext* to save memory by reading the dictionary from disk while processing. This option greatly slows down the time it takes to process large files.

The [-l] option limits the number of tokens in all sentence-models generated by the [-g] grammar defintion file. It has not affect on sentence model tables, [-m].

The [-u] option prints some processing statistics at certain intervals. These statistics are:

I: Number of Input Bits Read

E: Number of Extra Bits Appended to Input

O: Number of Output Bits

G: Growth Ratio: $\frac{100*O}{I+E}$

U: Number of Models Used

S: Number of Models Skipped Because of [-l]

N: Total Number of Model Elements Used

A: Average Number of Elements per Model: N/S

A.3.2 Using *scramble*

```
Usage: scramble -o outputFile [-d dictionary] [-i inputFile] [-s] [-v]
    -d   Specify the prefix for the dictionary file
         (i.e. mstr for mstrdict.dat, mstrtype.dat)
    -i   Specify the full input file path (default to stdin)
    -o   Specify the full output file path
    -s   Small mode - use tables from disk, don't load into RAM
    -v   Print undefined words to error output
```

The *scramble* parses words from natural languages texts in the input file. It outputs the bits corresponding to each word found in the dictionary. If the author of the natural language text were the *nicetext* program then *scramble* recovers the input used by *nicetext*.

The [-s] option causes *scramble* to save memory by reading the dictionary from disk while processing. This option greatly slows down the time it takes to process large files.

A.4 Utility Programs

A.4.1 Using *bitcp*

```
Usage: bitcp inputFile outputFile
```

The *bitcp* program is a very inefficient way to copy a file. It copies the file by reading and writing a random number of bits from each stream. The purpose is to test the *MTCinputBitStream* and *MTCoutputBitStream* classes. These classes are critical to the proper operation of the $NICETEXT$ system. This program helps when porting the software.

A.4.2 Using *bsttest*

`Usage: bsttest`

The *bsttest* program reads strings from standard input and it outputs a sorted list to standard output. The purpose is to test the functionality of the *MTCBST* container class. The class implements a binary-search-tree container [18].

This class is part of the *MTC++* container class library.

A.4.3 Using *listtest*

`Usage: listtest`

The *listtest* program reads strings from standard input and it outputs a sorted list to standard output. The purpose is to test the functionality of the *MTClist* container class. The class implements a linked-list container [18].

This class is part of the *MTC++* container class library.

A.4.4 Using *numsize*

`Usage: numsize`

Prints the *sizeof()* an unsigned long, unsigned int, and unsigned short for the current operating environment. This is useful when porting to other systems. In the case of *NICETEXT*, the typedef of *bitBucketType* and *bitCountType* in bitstrm.h rely on the size being four bytes (32-bits) for proper cross-platform operation.

A.4.5 Using *raofmake*

`Usage: raofmake`

Random Access Object Format (RAOF) tables are fast ways to access variable-length C++ objects from a file. The `*.dat` file contans the objects and the `*.jmp` and `*.alt` files are hash tables pointing to fseek() locations of objects in `*.dat`. [1]

[1] The *NICETEXT* system uses RAOF tables for the dictionary tables of *MTCdictRec* objects because of speed and portability.

The *raofmake* program reads strings from standard input to create two Random Access Object Format (RAOF) tables using *MTCwriteRAOF* container classes. The two tables are *ordered* and *unordered*.

The primary purpose is to test the *MTCwriteRAOF* class while porting. These classes are part of the *MTC++* container class library.

A.4.6 Using *raofmalt*

`Usage: raofmalt`

The *raofmalt* program uses an *MTCcreateAlt* class to create an alternate hash table `unordered.alt` sorted in string order. The input *MTCreadRAOF* object is the *unordered* RAOF table from the *raofmake* program.

The primary purpose is to test the *MTCcreateAlt* class while porting. These classes are part of the *MTC++* container class library.

A.4.7 Using *raofread*

`Usage: raofread`

The *raofread* program allows the user to interact with a Random Access Object Format (RAOF) table of strings. First, the program prompts for a `*.dat` file. Next, it prompts for a corresponding `*.jmp` or `*.alt` hashed index. The user may then choose to read the objects by number or by example.

The primary purpose is to test the *MTCreadRAOF* class while porting. These classes are part of the *MTC++* container class library.

A.4.8 Using *rbttest*

`Usage: rbttest`

The *rbttest* program reads strings from standard input and it outputs a sorted list to standard output. The purpose is to test the functionality of the *MTCRBT* container class. The class implements a balanced binary search tree container as a Red-Black Tree [7].

This class is part of the *MTC++* container class library.

A.4.9 Using *rinfo*

```
Usage: rinfo [-s] [dirname]
    -s: take snapshot
```

The rinfo program is an extension to the the GNU Revision Control System (RCS). It recursively looks through subdirectories finding RCS repositories and reports status information.

The information includes which files are checked out with a lock and which files have not been checked out of a repository.

Appendix B

Example Innocuous Texts

This chapter contains several more example texts generated by the *NICETEXT* system. In each case, the input to *nicetext* is the following ciphertext, shown in hexadecimal:

```
61eb  8570  576c  bf61  50b7  b3a3  fd98  32ba
67e4  afec  068b  e107  c3c1  cf71  9192  5f2f
4cfc  fb6a  3626  0b0d  3731  afaa  093e  6840
86da  ce16  cde8  364d  7058  c43a  93c6  3010
e947  3deb  34dd  e214  b5c9  90e2  b323  4617
254e  c4c4  736c  0b1c
```

The output has not been modified, except for the hyphenation of words by LaTeX.

B.1 Shakespeare

The style source was generated from *The Complete Works of William Shakespeare* available electronically at *ftp://ftp.freebsd.org/pub/gutenberg/etext94/shaks12.txt.* The first example uses words from the text. The second example uses a much larger master dictionary table.

Example 1:

> Not before the buttock, fair fathom, by my will. This ensign here above mine was presenting lack; I lieu the leopard, and did bake it from him. Him reap upon, the taste boyish. Captain me, Margaret; pavilion me, sweet son. At thy service. Stories, away. I will run no chase wrinkle.

Since Cassius first did leer me amongst Caesar I have not outstripped. Upon my fife, again, you mistook the overspread. WELL, Say I am; whether should proud dreamer trust Before the swords have any vapour to sing? HALLOA, whoever can outlive an oath? I catechize you, sir; beget me alone. Cornelius, I will. For me, the gold above France did not induce, Although I did quit it as a relative The sooner to respect which I intended; But God be picked before affectation, Whatever I in speediness abundantly will rejoice, Salving God and you to fashion me. If thou proceed As high as weather, my need shall catch thy deed. He drift a nature! Whose battle outlive you? Something. Enchanting him POSTHUMUS. That is my true disponge. Therefore, to plums. Sheet. SLENDER. FOULLY, And mine, That sought you henceforth this boy to keep your shame Blushing to rhyme. Be it so; go hack. MARSHAL. Will you be diamond before something? I lust not; I will forsake it good how you dare, ere which you care, and where you dare. How does my feather? She never should away without me. CEREMONIOUSLY, Lord; she will come thy bed, I overawe, And fling thee henceforth brave brood. Nay, look not so with me; we shall sear of your mightiness tremblingly. WHICH, Wast thou offer her this from me? Will they insomuch book after thee? DOST Thou desist her therefore? Didst thou not appear me impair I would not do it? Ere finger, without lightness, nor under finger, my lord; not with love. I am explanatory I shall misuse A loan by thee. Man forgets not me no, nor radiance neither, although by your fleeting you seem to say so. Come, shall we throughout it? Spell me, thou chalice shave, where are my children? Down, thou detecting sorrow! I heed not gad more title to your fire, Before ah I wend ye raise to burn them out. For us, we will define, During the knife above this old Cruelty, To him our absolute power; to Edgar and Kent you to your infects; With boot, and Such exhibition as your wenches Have more than gilded. You hold a bear galley; you do o, lord. NO, In tooth, sir, he could not. Fray, innocent, and forswear the foul compound. YOUR Wheel was wont to dwell me I could do something upon shrouding. Whilst that? Build, his work, As slight as is the eagle's, enshrines forth Lolling safety. WHEW! Become, Vale. I should be sad. Become, bring me where they are. Enter editions Come on, boon. Therefore we marvel much our sedition France Would in so honest a fortress shut his custom Past our lolling steers. ICE, I Say! I did not. I have a good vow, wind; I can see a church by appetite. Audit ARIEL How dares my illustrious sir? As good a carrion as the Emperor. Which, now? What before some partisans, sir, I lean to see. I repossess, I confess. In his tongue. My closure grieves me, successive mortar, now. Whatever is it you will see? Saw you my counter? I often came where I did hear of her, but cannot hand her. THERE Is chooser between a shrub and a butterfly; whereof your butterfly was a shrub. BECOME, Come, peace. DECORUM, His Possibility doth can for you, And for your Grace, and you, my headless pyramids. TUT, thou wilt speak again of appointment. FIRST GOTH. how blow? WHICH You will, Monsieur Jaques. Ever talk above it. Exit Administer a STRANGER SOLDIER.

Example 2:

Which subtext so cold that is not remounted here? Would import above bran once think it? Hansom, I will. You lid me mistake it generally and yeah, Gaining to the pension and the rhyme. You may not, my lord, disguise her inertial jute. HOW, My lord! The flame oneself doeth reek Before equivalent parody. Hark you, sir. The Formant House Girlish. ZOUNDS, the mud more occurs To rouse a baron than to sort a mare! INSURGENT Good, i faith! I have referred my father blame him. Precontently, niobium, imaginatively. Thence, pack! Now shine it like a comet above infringe, A packet to the shawl above all our shows! Sorghum, your Sovereignty is too much sad. Good sparrow, groceries. IN Reaffirming whichever was furthest, we shall part up another. A background valves! Strut my lace, Lillian, strum! Vestally, naturedly: Therefore acquiesce thee studiously of thy pin, For to deny each marble ere oath Cannot remove nor stoke the prolong permission That I do moan unright. Pester Johnnie. Now remit down, now omit down; come, margin. Adieu; be happy! While I fluoresce mushrooming it by topple. The raven himself is sparse That wednesdays the managerial entrance of Jamison Neath my numismatists. Far off irrigates I overhear the gotten drum. I can revive no bigger by skimming. OH, sir, to such as singing show their lures A mock is due. tempestuous and thief! This is thy sheath; there compost, and let me die. How thunderbolts I am! I shall rave yer temperance a little. Open the gates and reset me in. HULLO harnessing mate! AND That octet altogether is nondeliquescent. I will. Lord. It is not my midpoint, But my unconventionalize too. As thou port a knave and no nave. By the Lord, I knew ye as oh as he that obeyed ye. WHETHER, His genuine scowl. Your siphon, noble interoffice. Morrow! I cannot, lord; I have salient genesis, The tide whither is now. YEAH mealy shrivel! If he assail of that, He will have other means to strut you off; I incurred him and his masterpieces. I saw him leasehold Lord Percy at the print Unto surfier transience than I did look before Above such an unpaged subsphere. Off unlike my gnats, you plagues! Strum to me, mound nor foe, And expel me whoso is weber, York or Stanciu? SHE Shall, she shall. Not much maturation to them? OMIT It, Tint. Will he object you through the transport of France, And quake the mightiest above you resales and ricks?

B.2 Federal Reserve

The style source was generated from several texts available electronically at *http://www.bog.frb.fed.us/BOARDDOCS/TESTIMONY/*.

Advance around the Third Half during 1997

Either, the generally operative down ago relationships has financial. My output performance about alert points past the items grows that the efficiency to strain exhausted increases in to broader helps indicates a legitimate marketplace to incomes to trough second aspects by compensation either earlier sector, which improvements second and considerably banks than waiting than rate. We have much, before though, seen much surrender against the provide by point demands in, for condition, the reducing pass. Productive margin come a almost higher extent in the still patch like the performance, like indicated, pointed out up its soft phase about the store up the conduct. The Increase of Price Security

Relevance past consequent unemployment partners the currencies followed from intensifying before that representative. The expect by the food analysts to predict among bond exists, before it gradually indicates to hold same change against imported goods and durable resources some.

Mostly, I am sustainable that the Transitory Open Boost Software might issue to engender review interest reasons would the issue past increasing margin fairly discuss an possible reversal against slower industries that should intermediate the margin at the geographic extent.

Percent

Base stability is an legitimate however willing behavior before safety, not either although it returns unusual markets and the appreciation to coping most reasonably, for roughly while it most significantly lenders sector or timing sheets by the real become. There are, to be good, historic reasons than how not overall out level determination currently deliveries. Unusual conduct predict another largely higher overall out the percent help as the investment, before diversified, reversed on among its ago strain among the demand against the optimism.

B.3 Aesop's Fables

The style source was generated from *Aesop's Fables Translated by George Fyler Townsend* available electronically at:
ftp://ftp.freebsd.org/pub/gutenberg/etext94/aesop11.txt.

The Doe and the Lion A DOE hard fixed by robbers taught refuge in a slave tinkling to a Lion. The Goods undertook themselves to aversion and disliked before a toothless wrestler on their words. The Sheep, much past his will, married her backward and forward for a long time, and at last said, If you had defended a dog in this wood, you would have had your straits from his sharp teeth. One day he ruined to see a Fellow, whose had smeared for its provision, resigning along a fool and warning advisedly.

said the Horse, if you really word me to be in good occasion, you could groom me less, and proceed me more. who have opened in that which I blamed a happy wine the horse of my possession. The heroic, silent of his stranger, was about to drink, when the Eagle struck his bound within his wing, and, reaching the bestowing corn in his words, buried it aloft. Mercury soon shared and said to him, OH thou most base fellow? The Leather and the Newsletter A MOTHER had one son and one sister, the former considerable before his good tasks, the latter for her contrary wrestler. The Fox and the Lion A FOX saw a Lion awakened in a rage, and grinning near him, kindly killed him. Likely backwards the Bull with his machines fared him as if he were an enemy. One above them, hanging about, bred to him: That is the vastly precaution why we are so fruitless; for if you pomegranate represented us administer than the Instruments you have had so long, it is domain also that if labors became after us, you would in the lame manner prefer them to ourselves. It fell among some Loads, which it thus encased: I work how you, who are so light and useless, are not modestly rushed by these strong victors. Where she saw that she should let no redress and that her wings were pleased, the Owl talked the meekness by a victim. It feathers little if those who are inferior to us in estimate should be like us in outside expenses. my son, what of the hands do you think will pity you? The hero is brave in cords as o as weasels. I have the responses you condition, but where I shear even the trademark above a nibble dog I feel ready to extravagant, and fly away as earnest as I can. He accused him of having a maintenance to men by offering in the nighttime and not cleansing them to sleep. Be on regard against men who can strike from a defense. So, among other proceedings, this small lament appointment disclaims most of the poverty we could have to you if some thing is owe with your copy. Hence it is that men are quick to see the sweethearts above dangers, and while are often hand to their own trappings. Those who speak to please everybody please nobody. The Leaves and the Cock SOME LEAVES awoke into a house and skinned something but a Flock, whom they stole, and got off as aghast as they could. One above the daughters decided him, hammering:

Now, my good man, if this be all true there is no deed above villagers. One of his boatmen revived his frequent disputings to the spot and grunted to yore his complaints. On the punctuation above their grasshoppers, a refute chose as to whose had laid the most protect weather. Being in proofread of food, he ruled to a Sheep who was howling, and overworked him to fetch some whir from a team reaching close beside him. Living them to be stealthily heavy, they tossed about for joy and proposed that they had mistaken a large catch. Dragging their beauty, he tossed down a huge log into the lake. The Fishermen SOME FISHERMEN were out filching their efforts. In this manner they had not pointed far when they met a company above freedmen and oxen: Why, you lazy old fellow, died several offerings at once, how can you decide upon the beast, whereupon that poor little lad there can separately keep pace by the side above you? Some versions playing by saw her, and assuring a applicable aim, furtively ailed her. So securing twenty cords, he awakened another. The Grass and the Course AN GRASS consorted a Horse to spare him a tall dolphin above his proceed. The Stable, crying him, bred, But you really must have been out above your noises to sharpen thyself on me, who am myself always maimed to sharpen with daughters.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] Evan L. Antworth. *PC-KIMMO: a two-level processor for morphological analysis*. Number 16 in Occasional Publications in Academic Computing. Summer Institute of Linguistics, Dallas,TX, 1990.

[3] Evan L. Antworth. Introduction to two-level phonology. *Notes on Linguistics*, 53:4–18, 1991.

[4] Evan L. Antworth. Morphological parsing with a unification-based word grammar. In *Proceedings of the North Texas Natural Language Processing Workshop*, pages 24–32. University of Texas at Arlington, 1994.

[5] Evan L. Antworth. *User's Guide to PC-KIMMO Version 2*. Summer Institute of Linguistics, Inc., 1995. http://www.sil.org/pckimmo/v2/doc/guide.html.

[6] M. Burmester, Y. Desmedt, and M. Yung. Subliminal-free channels: a solution towards covert-free channels. In *Symposium on Computer Security, Threats and Countermeasures*, pages 188–197, 1991. Roma, Italy, November 22-23, 1990.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass. London, England.

[8] D. E. Denning. *Cryptography and Data Security*. Addison – Wesley, Reading, Mass., 1982.

[9] Y. Desmedt. Subliminal-free authentication and signature. In C. G. Günther, editor, *Advances in Cryptology, Proc. of Eurocrypt '88 (Lecture Notes in Computer Science 330)*, pages 23–33. Springer-Verlag, May 1988. Davos, Switzerland.

[10] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT–22(6):644–654, November 1976.

[11] R. G. Gallager. *Information Theory and Reliable Communications*. John Wiley and Sons, New York, 1968.

[12] Random House, editor. *Random House Webster's College Dictionary*. Random House, New York., 1991.

[13] D. Kahn. *The Codebreakers*. MacMillan Publishing Co., New York, 1967.

[14] Greg Lehey. *Running FreeBSD 2.1*. Walnut Creek CDROM, Walnut Creek, CA., 1996.

[15] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[16] DES modes of operation. FIPS publication 81. *Federal Information Processing Standard*, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., U.S.A., 1980.

[17] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Commun. ACM*, 21:294 – 299, April 1978.

[18] Robert Sedgewick. *Algorithms - 2d ed.* Addison-Wesley, Reading, Mass.

[19] G. J. Simmons. Message authentication without secrecy: A secure communications problem uniquely solvable by assymetric encryption techniques. In *IEEE Electronics and Aerospace Systems Convention*, pages 661–662. EASCON'79 Record, October 1979. Arlington, Verginia.

[20] G. J. Simmons. *Message Authentication Without Secrecy*, pages 105–139. AAAS Selected Symposia Series 69, Westview Press, 1982.

[21] G. J. Simmons. The prisoners' problem and the subliminal channel. In D. Chaum, editor, *Advances in Cryptology. Proc. of Crypto 83*, pages 51–67. Plenum Press N.Y., 1984. Santa Barbara, California, August 1983.

[22] G. J. Simmons. The secure subliminal channel (?). In H. C. Williams, editor, *Advances in Cryptology. Proc. of Crypto 85 (Lecture Notes in Computer Science 218)*, pages 33–41. Springer–Verlag, 1986. Santa Barbara, California, August 18–22, 1985.

[23] A. J. Thomson and A. V. Martinet. *A Practical English Grammar, Fourth Edition*. Oxford University Press, Hong Kong., 1986.

[24] John Valley. *UNIX Programmer's Reference*. Que Corporation, Carmel, Indiana., 1991.

[25] Peter Wayner. Mimic functions. *Cryptologia*, XVI Number 3:193–214, 1992.