A CORPUS OF SECOND LANGUAGE ATTRITION DATA

by

D. Rudy Smith

A project report submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Arts

Department of Linguistics and English Language

Brigham Young University

November 2007

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of the project report submitted by

D. Rudy Smith

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Alan K. Melby
Date	C. Ray Graham
Date	Alan D. Manning

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of D. Rudy Smith in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Alan K. Melby, Chair

Accepted for the Department

William G. Eggington
Department Chair

Accepted for the College

Gregory D. Clark
Associate Dean, College of Humanities

ABSTRACT

A CORPUS OF SECOND LANGUAGE ATTRITION DATA

D. Rudy Smith

Department of Linguistics and English Language

Master of Arts

This report addresses the lack of progress in the field of Second Language Attrition (L2A). Review of L2A history and literature show this to be cause by lack of appropriate data. Five criteria for appropriate data are suggested and a corpus of L2A data (57,000 words, spoken Spanish) which meets the criteria is presented. The history of the corpus is explained in detail, including subject selection, instruments and methods of collection, and markup -- XML was used to annotate the corpus with nineteen categories of speech errors, adapted from Nation's (2001) "Learning Vocabulary in Another Language."

An example analysis of how the corpus can be used for L2A research is provided with step-by-step instructions on writing scripts for data extraction and post-processing in the Perl language. Source code is included in the text. Complete beginners tutorials on the XML and Perl languages are included in the appendices. The report also introduces a

website, developed specifically to host the corpus, where researchers may register, download the corpus and share work they have done with the corpus. All files used in the example project, as well as this report, are available for download at the website.

Findings from the example analysis support Plateau Phases, the Regression Hypothesis and suggest the Threshold Hypothesis does not apply to marked forms. This shows the corpus to be of great value to the L2A research community.

ACKNOWLEDGMENTS

I cannot list everyone who has contributed to this project. To all who are not named, I appreciate your guidance and support so much. Thank you.

I want Phyllis to know I think she is the most knowledgeable, efficient and ahead-of-the-game office professional I have ever dealt with; for her great talents and efforts, a personal and heartfelt thank you.

I also want to thank Alan Manning, for his keen insights and prompt reviews of my writing and for teaching me accountability, Alan Melby, for his amazing sense of direction when I had none and for chairing the project, and Ray Graham, for the opportunity to work with him over the last several years, for his praise and thanks as well and great patience with my shortcomings and, for his invaluable data. Thank you.

I would like to recognize Deryle Lonsdale, a great mentor and friend, and thank him for introducing me to syntax, programming and formal Linguistics in general, for his consistent honesty about my weaknesses and for somehow seeing something in me that made him always believe in me. Thank you.

I must also thank Jan for not letting me quit and I am thankful beyond words to Francesca, Emily and Matthew for their patience, love and support and for making this and anything I have accomplished significant. Thank you and I love you.

Last, for the opportunity to study and the ability to comprehend I thank the Lord.

CONTENTS

List of Tables

LIST OF FIGURES

INTRODUCTION

Second Language Attrition (L2A) is where a person achieves a degree of proficiency in a second language and then, after some time outside the learning environment, manifests a reduced level of proficiency in that second language.

Here is an example of actual Spanish language attrition. Both paragraphs were obtained via the same test instrument from the same subject. The first test was administered soon after the subject returned from a 16-month stay in Latin America. Ten to 12 years passed between test A and test B and the subject had little or no practice with Spanish during that time. English gloss is provided in italics.

test A

Juan Perez era medico. Vivía en la avenida América. Él era casado y tenía cinco hijos. Un día él despertó a las seis de la mañana.

Juan Perez was a doctor. He lived on America Avenue. He was married and had five children. One day he woke up at six in the morning.

test B

Juan Pérez era un médico. Él vivió en la calle de América. Había casado y tenía cinco hijos. Una mañana abrió los ojos a las seis de la mañana.

Juan Perez was one doctor. He spent his life on the Street of America. He had given in marriage and had five children. One morning he opened his eyes at six in the morning.

Attrition features in the second test include the following.

Juan Pérez era un médico. The use of the indefinite article 'un' is unnecessary.

Él vivió The conjugation of vivir (to live) here is in perfective aspect (the preterite tense), indicating a completed action. The imperfect, would have been more suitable.

en la calle de América The use of the preposition 'de' is uncustomary in the noun phrase 'la calle de América.'

Había casado The choice of the verb 'haber' (to have done) is inappropriate, especially since it is non-reflexive. The appropriate verb to use would have been estar (to be).

Una mañana abrió los ojos a las seis de la mañana Here the phrase 'abrió los ojos' is a circumlocution for the verb 'despertarse' (to wake up.) The speaker also used the word 'mañana' redundantly.

Many linguists who have an interest in language acquisition are also interested in language attrition. Lambert et. al. (1982) report on a review of the field of Second Language Attrition study as it then stood. Conference attendees then hoped that L2A studies would give insight into designing language pedagogy and language refresher courses so as to reduce or more quickly remedy the effects of attrition.

INSUFFICIENT DATA

Progress in L2A research has been impeded due to a lack of sufficient data from the inception of this linguistic subfield. Lambert (1982) reports on the seminal conference in the field of L2A, held in 1980 at the University of Pennsylvania. In that book Lambert notes the need for better data in order to further the field of L2A, "everyone's got anecdotal evidence, but very little systematic research is done [...] the direct evidence is scanty, not very cumulative and [...] sometimes contradictory." (p7-8.) Fifteen years after these comments, another notable figure in the field, Elite Olshtain, notes that attrition studies still "tend to be limited by the lack of availability of large enough samples of cases similar in nature." And without such, comparison and validation of previous work in the field, hence, progress in attrition studies will not occur

(1989:164.) One problem that L2A is confronted with, then, has been that attrition data is limited.

It has come to a point where the idea of L2A as a phenomenon at all is being questioned. Speaking of fossilization and L2 attrition, Nakuma (2006:21) “rejects their characterization as observable phenomena whose existence can be demonstrated or proven by identifying and measuring their product through longitudinal research.” The article further goes on to say that L2 attrition is a phenomenon at all requires a comprehensive survey of a good sample of L2 users, specifically, “only if their products can be identified and measured will longitudinal research on these ‘phenomena’ be feasible. In that event, the researchers of attrition… would have to be able to anticipate fully and accurately the potential ‘losable linguistic items’ in order to be able to show later that those items have subsequently been lost” and poses the challenge of how to track ‘that which is not there.’

In contrast to these difficulties and opposition, some progress has been made in the field. On the one hand we should not say that nothing has come from the research, that data used to this point have yielded no insights. In the review of the literature we will see several points of agreement as to the description of the phenomenon of L2 attrition.

On the other hand, we will also see that findings of some studies conflict, especially with respect to what I will refer to as the *plateau hypothesis* (that after being removed from an environment which promotes acquisition, L2 learners will not immediately experience attrition, but that attrition sets in after a “plateau phase.”) And, to ignore the fact that the field has made no large gains towards consensus on what L2 attrition actually consists of, as a set of hypotheses that are repeatedly reinforced through subsequent research, would be ignoring reality. These deficits make it clear that the practical application for the research that was initially anticipated (see Lambert 1982) have never been realized.

In analyzing these facts, it is my opinion that the reason for so little progress is not best explained by suggesting that L2A may not be a phenomenon after all, that the data cannot reveal enough for us to take it as a phenomenon, rather, I feel that the field stands where it does primarily because of a lack of the right kind of data being accessible to enough researchers.

TOWARDS A SOLUTION

The aim of the project presented here is to first, remedy the lack of accessibility of such data, via a corpus of L2A data and a website, and second, to show that use of this corpus will lead to a clearer understanding of L2A as a phenomenon and can provide findings which move the field in the direction of consensus over certain hypotheses.

This project makes available via the web a corpus of L2A data, marked up for speaking errors that meets three qualifications for desirable data, described below.

The website allows interested parties to register and download this corpus and tools for analyzing it. It also provides a forum for users to upload and share their work with the data. The website includes what I call an example 'project', an extraction of data as well as an analysis. The findings of the example project show that the corpus can indeed be used to describe attrition as a phenomenon, i.e., provides evidence of attrition after ten years incubation. Finally, the process of extracting the data and its subsequent analysis are described in sufficient detail to be replicated (for linguistic features other than those chosen for the example project.)

18 subjects were Brigham Young University students who acquired Spanish as a second language during a period of 18-24 months. The initial two months were spent in intensive training which consisted of nearly 10 hours a day of formal lessons and task-based practices. The balance of the time was spent immersed in a Spanish-speaking culture as missionaries, interacting in all varieties of situations on a constant basis, entirely in Spanish.

The time of return by subjects from the Spanish culture/countries ranged from between the year 1982 and 1986. Their individual return dates, as well as their continued study and use of Spanish, as self-reported at the time of testing, are included in the data. The tests were first administered in three successive years, 1986, 1987 and 1988 and again in 1997.

The elicitation devices used during these tests consisted of a receptive and productive vocabulary test, picture-guided story narrations and free response to specified discourse situations. Since the vocabulary tests did not render freely spoken language, but numbered lists of words, only responses to the picture-guided narrations situational responses are included in the corpus.

The raw data were obtained by recording and then transcribing the responses to the narrations and situations. Each test yielded between 1200 and 3000 words.

Currently, the data has been annotated with a standardized markup language called Extensible Markup Language (XML) (Bray et. al. 2006) to account for errors corresponding to 19 categories. These categories were adapted form Nation's (2001) work on what it means to know a word. The corpus could be augmented with additional criterion variables, e.g., correct usage of existing categories, new error categories or links to the audio sound bites correlating to the transcribed text.

MOTIVATING XML

On one hand, the larger a corpus is, the more chance it has of being accurately and completely representative of linguistic phenomena. On the other hand, the task of managing a corpus includes recording sources of data, in this case tests administered to subjects, associated predictor variables, e.g., biographical information, test dates, etc., and identifying criterion variables, e.g., instances of particular linguistic features, becomes more difficult as the size of the corpus increases. Managing a corpus can be even more difficult when collaboration, e.g., sharing markup responsibilities or native speaker review, is required.

Though the corpus could hold latent answers to many of the questions one might ask about language attrition, most of these will not be accessible if it is merely a body of text or a collection of documents. For a corpus to be really useful, it must be fitted with an overarching framework for managing the data. XML happens to do that job very well.

Additionally, XML is an internationally recognized and the most popular of standards for storing electronic data. It has come to be used not only for data preservation, but as a vehicle for manipulating and presenting data. Its almost universal popularity and usefulness assures that the standard will be well documented and in use for a very long time.

This corpus uses XML version 1.0. Each text file is encoded as an XML file with the '.xml' file extension. The corpus consists of the data from two tests each for 13 subjects, a total of 26 tests. Each test is saved in an XML file. The name of the file is made up of the subject's id, followed by a dash, the test number, followed by a period, and the schema version for validating the xml file. So the files corresponding to the two tests from the first subject (S1) are:

```
S1-1.0.1.1.xml
S1-4.0.1.1.xml
```

The test number for each subject is either a '1' or a '4', since tests two and three were not used in this corpus. The current XML schema version, used to validate these files, 'v0.1.1'. The following samples provide an idea of what the data looks like in its raw state as compared to its marked-up, XML state, as it appears in the corpus. The details of building the corpus are covered in a later section.

raw data

```
En los Estados muchos tienen carros y las casas quedan a distante de
cada uno. Es lejos al centro.
```

xml data

```
<section type="situation" sectionid="2">
  <transcription>En los Estados muchos tienen carros y las casas quedan
a distante de cada uno Es lejos al centro</transcription>
  <units>
    <unit type="T-unit" unitid="30">
      <text size="6">En los Estados muchos tienen carros</text> </unit>
    <unit type="T-unit" unitid="31">
      <text size="9">y las casas quedan a distante de cada uno</text>
      <error category="210" errorid="1">
        <token>a distante </token>
        <repair>separadas </repair> </error>
      <error category="313" errorid="2">
        <token>uno </token>
        <repair>una </repair> </error>
      <error category="212" errorid="3">
        <token>uno </token>
        <repair>otra</repair> </error>
    </unit>
    <unit type="T-unit" unitid="32">
      <text size="4">Es lejos al centro</text>
      <error category="319" errorid="1">
        <token>lejos </token>
        <repair>distante</repair> </error> </unit> </units> </section>
```

FILE ORGANIZATION

Each test consists of the data from 13 individual oral responses to speech elicitation devices. Each of these responses is organized into a *section* element. The section element has two attributes, *type*, which indicates whether the elicitation device was a picture guided narration or a situational response, and *sectionid*, which is the order which this section appeared in the original transcript.

Each narration has been broken down into T-units and are labeled as *unit* elements. The general definition of T-unit as a main clause was used. This division makes it possible to express the complexity of the speech sample in terms of the T-unit. All of the unit elements have a nested *text* element, the attributes of *type* and *unitid*. The text element has a *size* attribute, indicating the number of words that make up the unit.

If the text contains speech errors from one of the 19 categories, they are stored in *error* elements with *category* and *errorid* attributes. The error element also contains a *token*, the portion of text containing the error, and a *repair*, or correction of the error. A unit may contain several errors.

REVIEW OF THE LITERATURE

THE RIGHT KIND OF DATA

The nature of the data used in past studies, as well as the findings of those studies are of interest. Since the claim is that there is a lack of the right kind of data, it seemed appropriate to clarify what I deem the right kind of data to be. I have chosen five qualifications for ideal data.

First, the nature of the study has a bearing on how many subjects are needed to get a representative sample. My judgment is that in order to investigate the use of complex linguistic structures at least six subjects would be needed.

Second, because adults and children acquire a second language differently (reason says it is possible that attrition different for them as well) and because child L2A data is much more common than adult L2A data, there is a risk of arriving at an incomplete understanding of attrition without knowing it. To avoid this, I feel that data from adult subjects is more valuable than data from child subjects and set the age requirement at 12 years old, which is approximately when the native language is fully acquired.

Third, the subjects must have acquired a high degree of proficiency in the L2. That is to say, they need to have been able to use complex linguistic structures, e.g., compound tenses and aspect, in everyday speech.

Fourth, the time between the first and last collection of the data must span more than five years of incubation (incubation starts when the subject leaves the learning environment.) Limited support for the plateau hypothesis has been found up to five years after return from the environment. Attrition must be detectable after this point, otherwise the phenomenon may not be one of a plateau.

Fifth, the original data, as well as the methods and instruments of collection and the process of its analysis, should be made available to other researchers. One reason for this is that the conclusiveness similar findings from independent studies is questionable as long as there is a question as to the similarity of the data behind the findings, and the most sure way to eliminate this possibility is to have a common data set. The other benefit of making original data available is that there is a greater possibility that it will lead to conclusions in the first place.

This summary list includes abbreviations.

1. That the data derive from a study with more than five subjects (N > 5)
2. That subjects be adult (A > 12)
3. That subjects have reached a high proficiency in the L2 (+P)
4. That incubation (and time course of the study) be longer than five years (I > 5)
5. That the original data be made publicly available or (+O)

COMPETING HYPOTHESES

The field of L2A cannot be said to have developed to the point where there are true competing hypotheses, well defined, continually refined, with clearly defined battle lines, etc. Far from it. What we do see are a few loosely defined notions about what L2A might be. I have chosen the three which seem to have the clearest definitions and most support in the literature.

Previously mentioned, the *Plateau Hypothesis* states that after being removed from an environment which promotes acquisition, L2 learners will not immediately experience attrition, but that attrition sets in after a period of sustained linguistic proficiency in the L2, or a *plateau phase.*

The *Threshold Hypothesis* claims that there is a threshold of acquisition of the L2, a degree of proficiency, after which attrition will not occur. This can be applied to either particular aspects of or to L2 proficiency in general.

The *Regression Hypothesis* states that attrition occurs in the reverse order of acquisition. A variation of this hypothesis, called the *recency hypothesis*, states that the most recently learned linguistic items will be the first to be forgotten. Another variation is where linguistic items that were best learned were last to be lost.

Past L2A research

In the review of the literature below each study is described briefly in terms of the origin and nature of its data. Theoretical contributions are also noted. Summaries of these studies are followed by a table of comparisons, showing how their data measures up to the five suggested criteria and how their findings have contributed to the three above-mentioned hypotheses.

Stern and Stern (1907) noted the acquisition of Malay by a nine-month-old German boy during two and a half years in Sumatra and the complete loss of Malay subsequent to returning to Germany.

Kennedy (1932) studied the retention of Latin by English-speaking secondary school children. Though some scholars doubt their reliability (Oxford 1982), the results of this study suggested that best learned principles were most resistant to attrition, a variation on the threshold hypothesis.

Kenyeres (1938) saw a seven-year-old Hungarian girl, after 10 months in Switzerland, abandon Hungarian entirely and speak only French, and then revert to Hungarian only weeks after returning to Hungary.

Geogehan (1950) tracked high school French, Spanish and Latin students' abilities in syntax, vocabulary and translation. Results were that Latin students showed decline in all three, whereas French and Spanish students actually showed gains in some areas.

Leopold (1954) monitored the process by which a girl, from the age of three to the age of seven, alternated between English and German as her dominant and declining languages. As a result, he was the first to suggest recency as an indicator for attrition, that is that the last features learned will be the first to be forgotten. He also was able to describe some syntactic interference between the two languages.

Scherer (1957) noted a slight decline in German reading, vocabulary and grammar abilities of first-year college German students.

Pratella (1970) tested 350 first and second year Spanish students and found a decline in oral proficiency over the summer.

Burling (1971) gives the account of a young English-speaking child who by the age of three and a half had spent two years in India, acquiring Garo, then quit using it within two months after leaving the country.

Ravem (1973) charted English question patterns of his own Norwegian-speaking children, and his findings supported Leopold's recency hypothesis.

Smythe et. al. (1973) found that 9th, 10th and 11th grade students of French (N 150) actually had some gains in L2 language skills after a summer recess, but that decline had set in by after eight months.

In Cohen (1974) complex Spanish structures of prepositions, gender, ser vs. estar and adjective agreement as used by the kindergarteners participating in the study showed that the proportion of errors to length of utterance increased over the summer recess.

In a follow-up study (Cohen 1975), two of the three participants showed attrition patterns consistent with the regression hypothesis and that a break from L2 study may result in the decrease of some kinds of errors.

Itoh and Hatch (1978) watched a young child take two months to begin to initiate English L2 conversations where Japanese was the native language.

Magiste (1979) saw that age was a factor for robust acquisition when two Dutch speaking girls, ages four and 10, were removed from an English L2 environment. After seven months in the Netherlands, the four-year-old entirely lost English ability, whereas the older girl retained much of hers.

Bahrick (1984b) tested reading comprehension, recall and recognition vocabulary of 733 individuals who studied Spanish while in high school and college. The tests were administered after up to 50 years of incubation (period after acquisition.) Results showed a constant loss of L2 skill, independent of the number of courses taken, that recognition skills are more robust than productive ones and that the degree of acquisition is the best predictor for future attrition. This last finding has been questioned by a recent critic of the field (Nakuma 2006) because its validity hinges on whether the subjects accurately recalled and reported on their L2 abilities from 50 years prior.

Mauerman (1985) studied the attrition of French by 25 native English-speaking adults who had spent two years in French-speaking countries. Testing occurred between four months and five years after the subjects had returned to the United States. The study showed that verb use frequency remained constant while utterance length tended to drop over time.

Olshtain (1986) tracked the English story retelling ability of Hebrew speaking high school students within a year of the end of their study of English. Compared to a control group of native English speakers, attrition of the English used by the Hebrew-speaking students was insignificant enough to add support to the hypothesis of a plateau phase post L2 acquisition.

Weltens et. al. (1989) examined the attrition of 150 native Dutch speakers who studied French for four to six years in high school or college. The subjects were tested for attrition during four years after their study of French ended. It was found that only grammar and lexical items were dramatically affected.

Yoshida (1989) and Yoshida and Arai (1990) found that lack of practice did not attrit listening comprehension, phonology and frequently used pragmatically laden items of Japanese children who had spent time away from then returned to Japan.

Smith (1996) tracked Spanish verb use by 30 English speakers who had spent one and a half to two years in Spanish speaking countries. The results of this study inconclusively support the plateau hypothesis.

Hansen (1999) focused on the negated structures used by English speakers of Japanese 25 to 30 years after returning from having lived up to two years in Japan. Results give some support to the regression hypothesis and the threshold hypothesis.

Reetz-Kurashige (1999) looked for attrition in 18 Japanese children who spent time in the U.S. After a three-year window of incubation she concluded that there is merit to the threshold hypothesis.

Russell (1999) took three collections over a two year period within five years of returning to the U.S. Subjects were adult speakers of English who had lived for one and a half to two years in Japan. Results showed an initial plateau phase with some retrieval failure, decreased vocabulary size but no significant decrease in vocabulary density.

Tomiyama (1999) studied the use of English by an eight-year-old Japanese child over a period of 19 months after returning to Japan. (The study was subsequently extended to cover a two and a half year incubation period.) Results are relatively rich and include descriptions of the differential attrition of sub skills (phonology, lexicon, morphology and syntax), the robustness of receptive over productive skills, the existence of a plateau phase, deteriorations in fluency, utilization of compensatory strategies, in a limited way (by comparing to two other returnee children), individual differences in attrition, affective factors and the early stages of attrition.

Yoshitomi (1999) looked for attrition in four Japanese-speaking girls over a one-year incubation window. The study reports on fluency, lexical retrieval in speaking, accuracy of phonology, morphology and syntax and global speaking skills and found that, for children, retention increased with age and L2 skill loss was more or less equivalent to proficiency.

Comparisons

Table 1 compares these studies to show which of the five qualifications their data meet (indicated with an X) and which of the three major hypotheses (abbreviated P, T and R) their findings support or oppose (+/- to indicate.)

STUDY	N>5	A>12	+P	I>5	+O	P	T	R
Stern & Stern			X					
Kennedy	X	X					+	
Kenyeres								
Geogehan	X	X						
Leopold			X					+
Scherer	X	X						
Pratella	X	X				-		
Burling			X					
Ravem			X	?				+
Smythe et. al.	X	X				+		
Cohen	X					-		
Cohen						+		+
Itoh & Hatch								
Magiste								
Bahrick	X	X		X			+	
Mauerman	X	X	X			+		
Olshtain	X	X				-		
Weltens	X	X	X			+	+	
Yoshida & Arai	X						+	
Smith	X	X	X			+		
Hansen	X	X	X	X			+	+
Reetz-Kurashige	X		X				+	
Russell	X	X	X			+		
Tomiyama			X			+		
Yoshitomi			X					+

Table 1: comparison of studies by data appropriateness and findings.

From the table we see that several of the studies derive from data which meets three of the five qualifications and one even meets four (possibly five; it is possible that Hansen has shared her data with other researchers.) Another positive to note is that most of the studies have findings that relate one way or another to the three hypotheses listed on the table. However, what these studies have not done is to develop together towards the refinement of those hypotheses.

These past studies provide general notions about L2A, that what is best learned will be last lost, what is last learned will be first lost, that loss is more detectable the longer a speaker has been out of the environment, but these ideas have remained in their general state. They have not become more refined by subsequent studies. Research has not moved slowly towards a more clear and detailed picture of what L2A is and there is no indication that it will begin to do so. I believe this stagnation is the direct result of researchers not having a rich and uniform set of data.

I believe that if the field had a good set of data, a common set of data, large enough to be used by many independent studies simultaneously, derived from actual and authentic speech in the L2, data which contains represents in rich linguistic detail the actual proficiency of the speakers, over a period of at least five years, data that can be analyzed in the most empirical fashion and become a template for the development of subsequent data sets, I believe that if the field had this kind of data that studies would begin provide detailed descriptions of the long-standing general notions and that subsequent studies would begin to refer back to these in support, refinement and refutation, producing a body of literature from which a clear picture of the phenomenon of L2A would eventually emerge. And, I believe that the corpus described in this report is that kind of data.

The Corpus

History

Credit must go to Dr. Ray Graham of the Brigham Young University Linguistics Department for gathering the data that this corpus uses. In the mid-1980s Ray selected 80 students at Brigham Young University (BYU) who had recently spent 1 ½ or two years immersed in a Spanish-speaking country or culture, acquired Spanish fully as a Second Language (L2) and subsequently returned from the immersive environment. Most of these returnees abruptly stopped using their L2 almost entirely.

These students became the subjects of a study in L2A and provided data through a series of four tests administered at intervals of roughly one, two, three and ten years after their return. For example, John Doe returned in 1984, was tested first in 1985, again in 1986, again in 1987 and finally in 1997.

The four tests were identical. They consisted of a speech elicitation component, where the subject gave oral narrations and situational responses.

Theoretical basis

Lambert et. al. (1982) report on a conference surveying the then nascent interest in SLA, or the Loss of Language Skills. Since that time, any progress made in the field has been limited mainly by a lack of appropriate data. Most studies draw on data from child language learners and/or students who only partly acquired the L2. (It is much more common for a child than for an adult to reach a high level of proficiency in an L2, then quit using it.) Other problems with the data were a lack of cohesion in the findings, or that the findings did not lead to a single cohesive hypothesis (or even several competing hypotheses.) In other words, the findings were not replicable.

This project seeks to provide lacking data as well as show that the data can produce findings robust enough to eventually result in a definite hypothesis about L2A.

THE DATA

Five questions with instruction made up the instrument to elicit the situational responses. The complete instrument is found in appendix A. The questions were designed to investigate the subjects' L2 proficiency in familiar and unfamiliar tasks. The familiar tasks were defined as those in which the subject had participated on at least semi-weekly basis and were chosen form daily routines and activities of the L2 environment. Questions one, three and four of the elicitation device were designed with these familiar tasks in mind.

These questions designed to investigate the subjects' L2 proficiency in less familiar tasks are represented by questions two and five in appendix A. This type of task was defined as one which the subject had possibly encountered while in the L2 environment, but most likely had no extensive experience with.

A picture guided narration device was used to elicit the data gathered in the narrations. This device and its method of use is fully described in appendix B. This instrument required the subject to listen to an English narration of eight short stories while simultaneously following looking at a set of pictures that illustrated the story of the narration. The subject was then instructed to use the pictures to repeat the narration in Spanish. Each of the eight narrations was designed to require the subject to use a particular linguistic feature such as imperfect or pluperfect verb tenses.

The corpus is formed of the data from 13 of the original 80 subjects. The test instrument which provided these data consisted of two speech elicitation devices which generated eight narrations and five situational responses. The combined number of words spoken by a subject during each test (narrations and situations combined) was between 1200 and 2400 words. These tests were administered four times over a ten to twelve year period. The corpus uses the data from the first and last tests and consists of approximately 57,000 words.

BUILDING THE CORPUS

Originally the data was in audio form (audio tapes.) When I began work on the corpus, tests number one had already been transcribed but were saved in an unknown file format. It was necessary to find the proper conversion process to save them in a currently supported file format. Essentially what I did was to use special settings in WordPerfect™ to open them and then saved them as Microsoft™ Word™ files.

These test one files had been annotated for speech errors previously. The notation used was an ad hoc set of symbols, including the asterisk, 'at' and 'pound' signs and other special symbols, inserted inline before, after or around sections of text which contained speech errors.

I found no legend of the meanings of the symbols but after listing them and comparing examples was able to deduce what they indicated. The set of error categories developed by Ray Graham and myself was considerably larger than this legacy set of error categories. A few of the legacy categories did more or less map to our new set of categories and this correlation saved some of the time it took me to identify and mark speech errors.

The fourth tests had not yet been transcribed. I transcribed them all (sometime between June 2001 and early 2002.) I used a transcribing machine, basically an audio tape player equipped with a foot pedal for starting, stopping and rewinding the play of the tape, while hand typing what I heard. Due to the poor quality of many of the recordings, as well as the tendency of most subjects to pause excessively, use English filler words (e.g. uh, um) interchangeably with Spanish ones (e.g. pues), mumble, code switch, use nonce words, self-talk in English, i.e., say something like, "Oh boy! This is a lot harder than I expected." Transcription was painstaking. Each test required three to five hours to complete; one or two took much longer. I estimate that the total process took between 50 and 60 hours to complete.

Transcription was followed by markup of the transcribed text for errors. This was done by reading through each test, line by line, and assigning an error code (corresponding to the error categories) to each error as it was found. The process of separating the body of text into individual T-units was also done at this time. Each test required between three and six hours to complete, some much more than. The task of marking tests for errors introduced me to the phenomenon of judgment fatigue. This is where, after a few hours of considering errors for categories, an inordinate amount of thought was necessary to make a judgment. I realized that if I didn't take a break at these times, judgment errors were much more frequent.

My original method for marking an error did not use XML, but was done by inserting an error code, preceded by a @ sign, into the line of text just before the error, and to use square brackets where the error included more than one word. The correction of the error, or the repair, was not added until later, around the time I began converting to XML. Here are examples of a single-word and a multi-word error, the carats signaling the words comprising each error.

```
Juan Pérez @315fue un doctor.
                ^
Él vivía en la calle @[las américas]
                       ^      ^
```

Sometime during this process Dr. Graham hired a native Spanish speaker to check the annotations for accuracy. This person read through the annotated texts and made notes on the correct usage for particular errors, marked unmarked errors. His method of review was to print out a hard copy of the tests, make notes on the paper and return the tests to me. I then edited the electronic file with Microsoft Word™.

Once the native readings were complete, the corpus consisted of 26 Word documents and could be searched. This was done using a BYU proprietary software program called Wordcruncher. The program provided a tool to convert the Word document into another file type called an EBT, which was then fed back into the Wordcruncher program.

Error instances in each test or particular sections of tests could be identified via manually typing in a query which specified the item to look for, as well as the number of words surrounding the item which should be shown in the results. The results of the query were displayed in a separate window and organized by section heading. For example, to find the errors of category 315 for a particular test I would open that test's EBT file in Wordcruncher and in the search window type "@315" then click a button to initiate the search. The outputs could be transferred to a text file by highlighting them in the results window, copying and pasting them into a separate application such as Word.

Though cumbersome, this process was useful. However, the current state of the corpus began to show limitations and complications in use, mainly stemming from it's ad hoc nature. For example, if I wanted to look at errors for two particular subjects, it was necessary to create first a Word document of both files and then generate the EBT before searching for errors.

There were also cases where an error was discovered to be an artifact, or a speech error had gone unmarked in one of the tests. This required a correction of the word document and regeneration of the EBT. If research based on previous versions was affected, more often than not the manual searches had to be performed anew as well as cut-and-pasting the results to a separate file.

There was also the question of making alterations to the matrix of error categories. Even the addition of a single category (such as the catch-all category) required a new set of word documents and EBT files. And, it was doubtful that certain improvements to the corpus, such as adding a phonetic transcription of the text, could even be supported by the current system. Incorporating them would require more ad hoc machinery such as a modification to the Wordcruncher software or a separate piece of software developed for the specific incorporation in mind.

Finally, the ad hoc nature of the corpus would require anyone wanting to use it to learn the ad hoc method, which in turn kills the chances of its distribution and wider use.

These were the sorts of issues which I encountered in the development and use of the corpus which got me to consider converting it to a recognized standard. I had had some exposure to XML through a computer science course and after researching it more extensively realized that it would preserve the data in a recognized and documented standard as well as not only remove the limitations imposed by the current form of the corpus, but make many other enhancements possible.

So I moved the corpus to XML, first converting the Word documents to plain text files and then used a programming language called Perl (Active State 2007) scripts and a text editor called Emacs (Harvey 2007) to automatically write the XML. Here is a comparison of the data markup in the original versus the current corpus: original

original

```
Juan Pérez @315fue un doctor
```

current

```
<unit type="T-unit" unitid="1">
  <text size="4">Juan Pérez fue un doctor</text>
  <error category="315" errorid="1">
    <token>fue</token>
    <repair>era</repair>
  </error>
</unit>
```

The corpus is now preserved as 26 XML files.

Error categories

In order to make analysis of the data more productive, the speech errors identified in the transcriptions were organized into 19 categories. These categories were created by Dr. Ray Graham and Rudy Smith of the BYU Linguistics Department and are based on Paul Nation's 2001 book *Learning Vocabulary in another Language*. The categories are shown in Table 2.

CATG	DESCRIPTION	EXAMPLE
110	word is made up	*braktus* > *desayuno*
111	is used	*preschool* > *prekinder*
112	word is mispronounced	*personos* > *personas*
120	morphemes added	*ahoramente* > *ahora*
121	inflectional or derivational morphology error	*trayó* > *trajo*
210	a lexical phrase is used inappropriately	*a distante* > *separados*
211	incorrect word used, self correction	*salgo bajo* > *bajo*
212	incorrect word used	*para* > *por*
310	an article us used inappropriately	*azúcar* > *el azúcar*
311	error in person agreement	*me llama* > *me llamo*
312	error in number agreement	*las escultura* > *la escultura*
313	error in gender agreement	*sitio buena* > *sitio bueno*
314	error in tense features	*tengo* > *tenía*
315	error in aspect features	*recordó* > *recordaba*
316	error in modality features	*habla* > *hable*
317	error in theta role assignment	*leer* > *leer el libro*
318	error in case assignment	*ver las personas* > *ver a las personas*
319	errors need to be categorized	
320	uncustomary collocation	*en cuanto de* > *en cuanto a*

Table 2: error categories.

USING THE CORPUS

EXAMPLE PROJECT

Using the corpus is straightforward. There are three basic steps:

1. Decide what features you want to investigate (define the query.)
2. Extract corresponding data from the corpus.
3. Analyze the data that was extracted.

This section describes the process involved in following these three steps to create an example project, the results of which have been posted to the project website. This project was completed using a personal computer and the Microsoft Windows™ XP™ operating system. Users of other versions of Windows, as well as users of most other operating systems, e.g., Apple/Macintosh™ OSX™, will find the description to be highly representative, if not identical, to what their machines would require.

STEP ONE: DEFINE THE QUERY

Deciding what to look for in the data is driven by your research interests. This example focuses on errors that subjects made of the type category 315, errors in aspect features.

Spanish aspect is similar to its tense in that it is encoded in the inflectional morphology. The different Spanish aspectual morphemes and their English equivalents and examples are given in Table 3 and, where necessary, conjugated for the third person singular.

SPANISH	**EXAMPLE**	**ENGLISH**	**EXAMPLE**
-ó / *ió*	*ayudó*	-ed	helped
-aba / *ía*	*ayudaba*	was -ing / used to [verb]	was helping, or used to help
-ado / *ido*	*ayudado*	-en	been
-ar / *er,ir*	*ayudar*	to [verb]	to help
-ando / *iendo*	*ayudando*	-ing	helping
-ía	*ayudaría*	would [verb]	would help

Table 3: Spanish and English verb tenses.

The next step is to extract instances of the incorrect use of these items from the corpus.

STEP TWO: EXTRACT DATA

There are many ways to get the data we want from the corpus. Because of its simplicity and its having been designed with text searching and manipulation in mind, I chose to use the Perl programming language. I retrieved ActivePerl, a free program available from Active State (www.activestate.com), which installed the Perl language version v5.8.8 on my computer. (This section assumes no familiarity with the field of computer programming; however, only general explanations of using Perl and only basic definitions of key terms are included. Appendix G provides a detailed Perl tutorial, complete with step-by-step instructions for installing Perl, as well as explanations of how Perl was used in the sample project. It also includes a reference of all of the Perl commands used.)

Once Perl was up and running on my machine, I wrote two Perl scripts (a list of Perl commands.) The first script opened an XML file, printed once the subject id and test number corresponding to that file, then searched for instances of category 315 errors in the file. For each instances it printed the error and the prescribed correction for that error. This script (called *perlscript-a.pl* and downloadable from the website) is shown below.

```
use XML::Simple;                # Opens library for XML processing
$file = $ARGV[0];               # The file to process
$error = $ARGV[1];              # The error to look for
open(TEXT, $file);            # Open the file
@lines = <TEXT>;        # Read it into an array
close(TEXT);                  # Close the file
$parser = new XML::Simple;      # Make our own little XML parser
$ref = $parser->XMLin("$file");    # Give it our XML file
print "", $ref->{subject}->{name}, ": "; # Print the subject's name
print "", $ref->{data}->{id}, ": ";      # Print the test id
print "", $error, "\n";                  # Print the error category
## Print out the errors
foreach $section (@{$ref->{data}->{sections}->{section}}) {
    foreach $unit (@{$section->{units}->{unit}}) {
      if (ref( $unit->{error} ) eq "HASH" ) {
          if ( $unit->{error}->{category} == $error ) {
            $count++;
            print $count, "\t";
            print $unit->{error}->{token}, ":";
            print $unit->{error}->{repair}, "\n"; }}
      elsif (ref( $unit->{error} ) eq "ARRAY" ) {
          foreach $error (@{$unit->{error}}) {
            if ( $unit->{error}->{category} == $error ) {
                $count++;
                print $count, "\t";
                print $error->{token}, ":";
                print $error->{repair}, "\n"; }}}
    } # This brace closes the second 'foreach' loop
} # This brace closes the first 'foreach' loop
```

The second script opened all of the XML files corresponding to test one and counted the errors of category 315 which they contained, then did the same for test-four XML files. The two counts were then printed. Here is the script (called *perlscript-b.pl* and downloadable from the website) for that task:

```
use XML::Simple;                # Perl library for XML processing
$error = $ARGV[0];              # The error to look for
@filesT1 = qw( S1-1.0.1.1.xml S2-1.0.1.1.xml S3-1.0.1.1.xml
            S4-1.0.1.1.xml S5-1.0.1.1.xml S6-1.0.1.1.xml
            S7-1.0.1.1.xml S8-1.0.1.1.xml S9-1.0.1.1.xml
            S10-1.0.1.1.xml S11-1.0.1.1.xml S12-1.0.1.1.xml
            S13-1.0.1.1.xml ); # Test one files
@filesT4 = qw( S1-4.0.1.1.xml S2-4.0.1.1.xml S3-4.0.1.1.xml
            S4-4.0.1.1.xml S5-4.0.1.1.xml S6-4.0.1.1.xml
            S7-4.0.1.1.xml S8-4.0.1.1.xml S9-4.0.1.1.xml
            S10-4.0.1.1.xml S11-4.0.1.1.xml S12-4.0.1.1.xml
            S13-4.0.1.1.xml ); # Test four files
## Print out the errors
print "Counts for category ", $error, ":\n";
foreach $file (@filesT1) {
    countErrors(); }
print "    Test1:  ", $count, "\n";
$count = 0;
foreach $file (@filesT4) {
    countErrors(); }
print "    Test4:  ", $count, "\n";
sub countErrors() {
    $parser = new XML::Simple;      # Make our own little XML parser
    $ref = $parser->XMLin("xml/$file");    # Give it our XML file

    foreach $section (@{$ref->{data}->{sections}->{section}}) {
      foreach $unit (@{$section->{units}->{unit}}) {
          if (ref( $unit->{error} ) eq "HASH" ) {
            if ( $unit->{error}->{category} == $error ) {
                $count++; }}
          elsif (ref( $unit->{error} ) eq "ARRAY" ) {
            foreach $error (@{$unit->{error}}) {
                if ( $unit->{error}->{category} == $error ) {
                  $count++; }}}}}}
```

These scripts are run from a command prompt, such as the Windows DOS prompt (see appendix F for a tutorial on the Windows command line), and the output prints to, or appears on the screen. The outputs were subsequently copied and pasted into a text files. The first several lines of this file are shown here and the complete file is found in appendix H and available by download from the website under the name *errorcounts-a.txt*.

```
Counts for category 315:
    Test1:  60
    Test4:  67
S1: Test1
1     hacía :hizo
S1: Test4
1     necesitó :necesitaba
S2: Test1
S2: Test4
1     se levantó :se había levantado
2     fue :había ido
3     había puesto :puso
4     visita :visitar
S3: Test1
1     he vivido :viví
S3: Test4
1     estaba :estuvo
2     habría :habrá
3     decía :dijo
```

```
4     pidió :pedir
```

The first three lines of the printout show the cumulative counts of this type of error for each test, then lists the error and correction for both tests for each subject. Next the errors are shown for each subject separately for each test. Within each test the errors were numbered sequentially to the left in order to make referencing and counting them easier. We can see from this printout that the total number of errors of category 315 for test one was 60 and for test four 67, that S1 made only one error of this kind in each test, etc. Once these data were gathered, I began an analysis.

STEP THREE: ANALYZE THE DATA

I considered first the total error counts, under section one below, then the qualitative report of errors, under section two.

section one data

I thought that the raw error counts might be more revealing if they were shown as a function of another feature of the data and chose three for follow up analyses, 1) individual subjects, 2) total words used and 3) number correct uses structures pertaining to category 315.

In the first follow up I saw that subjects 11, 12 and 13 have far more in their first than fourth tests. This imbalance is likely a function of the amount of exposure these subjects had to Spanish in the years intervening between tests one and four. I did not pursue this option further, though evaluation of the amount of exposure each subject had to Spanish between tests one and four is possible. A survey requesting such information was administered with each test. Such an analysis would likely end up being a case study of a particular subject or set of subjects. This biographical information could be added to the corpus and used as a filter when querying error counts, etc.

For the second follow up I wrote a perl script to count the number of words and errors for tests one and four for each subject, as well as for tests one and four for the corpus. The script also calculated the ratio of errors per words used for each subject by test, as well as for the corpus. It is shown here and available for download from the website by the name *perlscript-f.pl.*

```
use XML::Simple;
@subjects = qw ( S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 );
@testids = qw (1 4);
$vappend = ".0.1.1.xml";
$corpusT1Words=0;
$corpusT2Words=0;
$corpusT1Errors=0;
$corpusT2Errors=0;
main();
print "corpus\n";
print "  t1 $corpusT1Words $corpusT1Errors ".
($corpusT1Errors/$corpusT1Words)."\n";
print "  t4 $corpusT2Words $corpusT2Errors ".
($corpusT2Errors/$corpusT2Words)."\n";
############## subroutines ##################
sub main {
      foreach $subject (@subjects) {
#     print "$subject\n"; # uncomment for more detailed printout
            print "$subject ";
            $t1ratio=0; $t2ratio=0;
            foreach $id (@testids) {
                  $file = "$path$subject-$id$vappend";
                  $parser = new XML::Simple;
                  $test = $parser->XMLin("$file");
                  $testWords=countTestWords($test);
                  $corpusWords+=$testWords;
                  $testErrors=countTestErrors($test);
                  $corpusErrors+=$testErrors;
#           print "  t$id $testWords $testErrors"; # uncomment for detail
                  if($t1ratio==0){
                        $t1ratio=($testErrors/$testWords);
                        $corpusT1Words+=$testWords;
                        $corpusT1Errors+=$testErrors;}
                  else{
                        $t2ratio=($testErrors/$testWords);
                        $corpusT2Words+=$testWords;
                        $corpusT2Errors+=$testErrors;}
#           print " ". ($testErrors/$testWords)."\n"; # uncomment for detail
            }
            print "". ($t2ratio-$t1ratio). "\n";
      }
}
```

```
sub countTestWords{
      $mycount=0;
      my $ref=$_[0];
      foreach $section ( @{$ref->{data}->{sections}->{section}} ) {
            @words = split (/ /,$section->{transcription});
            $mycount += ($#words+1);
      }
      return $mycount;
}
sub countTestErrors{
      $mycount=0;
      my $ref=$_[0];
      foreach $section ( @{$ref->{data}->{sections}->{section}} ) {
            foreach $unit (@{$section->{units}->{unit}}) {
                  if (ref( $unit->{error} ) eq "HASH" ) {
                        $mycount++;
                  }
                  elsif (ref( $unit->{error} ) eq "ARRAY" ) {
                        foreach $cat (@{$unit->{error}}) {
                              $mycount++;
                        }
                  }
            }
      }
      return $mycount;
}
```

The results of this perl script are shown in the following table. The subjects in the first column, errors and ratio of errors per word for each test in columns two through seven, and the final column shows the difference between the ratio of test four and the ratio of test one.

	TEST ONE			TEST FOUR			
subject	**errors**	**words**	**e/w**	**errors**	**words**	**e/w**	**r4-r1**
S1	182	1999	0.091	219	1567	0.14	0.049
S2	267	2188	0.122	278	1868	0.149	0.027
S3	209	1800	0.116	270	1406	0.192	0.076
S4	162	2271	0.071	286	2328	0.123	0.052
S5	194	1861	0.104	241	2054	0.117	0.013
S6	207	2604	0.079	294	2812	0.105	0.025
S7	175	1945	0.09	330	2241	0.147	0.057
S8	165	2119	0.078	177	1938	0.091	0.013
S9	128	2049	0.062	218	1734	0.126	0.063
S10	248	1904	0.13	212	1324	0.16	0.03
S11	227	3049	0.074	322	2698	0.119	0.045
S12	168	3132	0.054	196	2876	0.068	0.015
S13	223	2999	0.074	295	2359	0.125	0.051
corpus	2555	29920	0.085	3338	27205	0.123	0.037

Table 4: word and error counts by test and subject.

We can see that all of the subjects had a higher error/word ratio for the fourth than for the first test, and that the overall ratio of errors to words for all subjects increased from 0.085 to 0.123 (about 43%.) This quantitative account of speech errors made by the thirteen subjects offers good evidence of attrition and refines the definition of the plateau phase to be less than ten years. (Smith, 1999, saw the plateau extended to five years in his test of verb use. Some of the 13 subjects of this study also participated in the Smith study. Smith looked at verb use and this study takes verb use into account as well.)

The third follow up showed an area where the corpus could be improved. In order to derive the ratio of correct to incorrect uses of category 315 structures, markup would first have to be added to the corpus for the *correct* uses of category 315 in the same way the 315 errors were added. Someone would need to read each test and manually write the XML to reflect the correct usage. A native speaker would then need to check the markup for authenticity. This change to the XML would require a slight modification of the schema used to validate the corpus, then the whole corpus would need to be revalidated. After this was done, a Perl script could be written to extract both correct and incorrect usages of category 315 errors in each test and the ratio calculated.

The analysis of the section one data shows both strengths and weaknesses of the corpus. On the one hand, there is good evidence of attrition. On the other hand, more detailed analyses would be possible if the corpus were augmented for correct usage and with subject biographical data.

section two data

With section two I began by subcategorizing individual instances of category 315 errors by the type of aspectual morpheme of that error token and repair. For example the information for S1's errors is below.

```
S1: Test1
    1 hacía: hizo: imp: pret (irr)
S2: Test4
    1 necesitó: necesitaba: pret: imp
```

In test one S1's one used the word hacía, an instance of the imperfect, abbreviated 'imp', in place of hizo, an instance of the preterite, abbreviated 'pret', and which is also an irregular conjugation, as indicated by the (irr). S1's error of test four was the opposite, a use of the perfect in place of the imperfect, this time both the verbs conjugated regularly. I categorized all of the errors made by subjects in this manner. The results are shown in appendix J and can be downloaded from the website by the file name *errorcounts-b.txt.* They include the following abbreviations: 'perf' for perfect, 'pres' for present, 'prog' for progressive and 'cond' for conditional. 'PRESENT' and 'FUTURE' are used to indicated forms of a verb that are not part of the aspectual matrix that makes up category 315, but which occur as either the token or repair.

I decided to do some post processing of the forms used in place of the imperfect aspect. In other words, I looked at the instances in which they should have been used, or were required by the context, but were substituted for by another form. I started out by examining, the -ó/-ió past tense morphemes, often called the preterite tense and indicative of a perfective aspect, they stand in contrast to the imperfective morphemes –aba and –ía. I looked at these substitutions for tests one and four separately. I wrote a Perl script to extract and count the occurrences of each form and analyzed the simple constructions and the complex constructions separately.

simple constructions

There were 13 instances in test one and 17 in test four where the simple perfect (single verb inflected for the perfect) was required and not used. This became more interesting when I considered the associated tokens, or what was used in place of these forms. Ten of the 13 instances of test one and 12 of those in test four show a substitution of the perfect by an imperfect form. Since this phenomenon was not unbalanced in occurrence towards either test one or four, it could imply more about errors in the perfect in general than perfect as an item of attrition, that generality being that errors in perfect are, by substitution of the imperfect, more frequent than by any other form.

Next I began to tease out the instances of a substitution of a perfect irregular form (e.g., the verb *hacer* is not inflected *hació*, rather as *hizo*) by a regular imperfect form. This occurred five times in test one and nine times in test four. Though they are few in number, the second test did have twice as many.

Seeing that imperfect forms were used in place of irregular preterite forms, I decided to look at all of the substitutions of a perfect form by an imperfect form, as well as the inverse, all of the substitutions of an imperfect form by a perfect form. They are listed below.

```
1.    pret -> imp 5/3
2.    pret -> imp(irr) 0/0
3.    pret(irr) -> imp 5/9
4.    pret(irr) -> imp(irr) 0/0
5.    imp -> pret 4/4
6.    imp -> pret(irr) 17/0
7.    imp(irr) -> pret 0/0
8.    imp(irr) -> pret(irr) 9/2
```

`In this list A -> B` means "A was needed and B was used." And where the ratio at the end of each line indicates the number of times the substitution was found in tests one and tests four, respectively.

I noticed from these data that the use of irregular imperfect as a substitution for any perfect (second and fourth items) went unattested in both tests. This stands in contrast to the high number of perfect irregular substitutions for the imperfect (items six and eight). I thought this could be an indication that, in terms of language acquisition , the imperfect is a more marked form than is the perfect. Second I noticed, as shown in item number six, that there were many instances of substituting the perfect irregular for the regular imperfect in the first test and none in the fourth test. This was interesting enough for me to count the instances of individual tokens in test one, shown here:

```
1.    hubo = 12
2.    dijo = 2
3.    hizo, tuvo, tuvimos = 1 (each)
```

I then asked why the form was being used so frequently in this manner in tests one? *Hubo* is less common than *dijo*, *hizo* or *tuvo*/tuvimos, generally. Yet, here it is, being used as an incorrect substitution for the imperfect in test one but not in test four.

These data imply that hypercorrection decays with attrition, that the substitution of an unmarked form by a marked form is less common as one looses proficiency in the L2. Dr Ray Graham (personal communication, November 15, 2007) has noticed that loss of marked items is an attribute of attrition in children. This finding seems to indicate that the threshold hypothesis does not apply to marked forms and is a good example of how the corpus can be used to provide a more detailed description of L2A in adults, compare such to child L2A and to refine existing theories.

complex forms

Here I counted the number of times that the pluperfect, an imperfect form of the verb to have (*haber*) and a perfect form of some other verb (*había comido*, or "had eaten," for example), was substituted for by a simple form, resulting in an error. This happened four times in test one and 22 times in test four, which indicates that substitution of simple forms for complex forms is an aspect of attrition. And, since complex forms are learned after simple forms, it also lends support to the regression hypothesis.

CONCLUSIONS TO THE EXAMPLE PROJECT

In the quantitative analysis we saw find that attrition can be measured as a function of the number of speech errors per words spoken. We also saw that the plateau phase is somewhere between five and ten years, specifically, that between five and ten years incubation a speaker will lose the ability to use verbs correctly.

The qualitative component of the example project showed that attrition is in part characterized by a tendency to use a regular imperfect tense in place of an irregular imperfect tense and to use simple conjugations in place of the pluperfect. We also saw that the threshold hypothesis does not apply to marked forms, as well as support for the regression hypothesis.

These findings show that use of this corpus will help move the field towards a clearer understanding of the phenomenon of attrition in adults, and refine existing hypotheses in the process.

The Website

The corpus was built to solve the problem of the lack of access to L2A data. The website was developed with four purposes in mind.

1. Make the corpus widely available.
2. Document the corpus.
3. Provide instruction on extracting information from the data.
4. Facilitate sharing work done with the data.

User registration

An interested researcher begins by registering their self as a user of the corpus. This requires creating a username and a password. This information is stored in a MySQL™ (MySQL AB 2007) database. The user then navigates to a login page.

Upon login the user sees their homepage. (Screenshots of the website are included in appendix K.) The homepage greets the user and invites them to register a project. If the user has registered any projects on previous visits, these appear in a list. At this point the user has the option of logging out, completing a tutorial on Perl, reading the corpus documentation, registering a new project, editing an existing project or browsing projects from all registered users.

Documentation page

The corpus documentation on the website is very similar to the section in this report. The documentation explains the origin of the data, what is XML and how it is used to organize the data and a description of the Perl tools available for extracting information from the corpus, or the Perl tutorial.

REGISTER PROJECT PAGE

The webpage for registering a new project is similar to the page for registering new users. The user provides a unique id for the project and a title. The new project information is sent to the database and the project is added to the list of projects on the user's homepage. There is no limit to the number of projects a user may register.

EDIT PROJECT PAGE

The user arrives at the edit project page by selecting from their active projects listed on their homepage. They are presented with a field containing the title of the project and a text field containing the abstract. These fields are editable. The user may also attach Perl scripts used in the project and other documents, i.e., images, PDF files, etc. to describe the project.

When the user is done editing, they save their work and the changes are updated in the database.

BROWSE PROJECTS PAGE

When the user navigates to the 'browse projects' page, they are presented with a list of all the projects that users have registered. The user selects one and is able to view the project abstract and download attached files.

Setting up the server

In order for the corpus to be available, the website must be hosted on a server accessible from the world wide web. There are three components which need to be set up for the website to work, the server, the database and the website files.

setting up the server

The server must have MySQL and PHP v4 installed on it and needs to have FTP access. The amount of storage space needed depends on the number of users and the type of files they will attach to their projects. Lastly, a domain name must be associated with the server space.

setting up the database

The database which will store the user information and their projects must be set up for the website to work. This is done via MySQL queries. An explanation of MySQL is beyond the scope of this project, but the necessary tables and properties are shown here:

```
user table:
`name` VARCHAR( 65 ) NOT NULL
`password` VARCHAR( 65 ) NOT NULL
`email` VARCHAR( 65 ) NOT NULL
PRIMARY KEY ( `name` )

projects table:
`id` TINYINT(3) UNSIGNED NOT NULL AUTO INCREMENT
`user` VARCHAR( 65 ) NOT NULL
`title` VARCHAR( 100 ) NOT NULL
`description` TEXT NOT NULL
PRIMARY KEY ( `id` )

Perl scripts table:
`id_files` TINYINT(3) UNSIGNED NOT NULL AUTO INCREMENT
`bin_data` BLOB BINARY NOT NULL
`description` TINYTEXT
`filename` VARCHAR( 50 ) NOT NULL
`filesize` VARCHAR( 50 ) NOT NULL
`filetype` VARCHAR( 50 ) NOT NULL
`proj_id` TINYINT(3) NOT NULL DEFAULT ( 0 )

documents table:
`id_files` TINYINT(3) UNSIGNED NOT NULL AUTO INCREMENT
```

```
`bin_data` MEDIUMBLOB BINARY NOT NULL
`description` TINYTEXT
`filename` VARCHAR( 50 ) NOT NULL
`filesize` VARCHAR( 50 ) NOT NULL
`filetype` VARCHAR( 50 ) NOT NULL
`proj_id` TINYINT(3) NOT NULL DEFAULT ( 0 )
```

setting up the website files

The files that make up the website, included on the companion disk to this report, can be placed in any server directory. The domain name must be pointed to the directory containing the file called ‘index.php.’ Finally, the file named ‘opendb.inc’ must be edited to include server and database connection information. This file is shown below. The seven variables, indicated with *$* signs, must be set.

```
<?php
// Contains connection and database info. Connects to database.
/* Server */
$host = "";
$username = "";
$password = "";
/* Database */
$db = "";
$userTbl = "";
$projectTbl = "";
$perlTbl = "";
$documentTbl = "";
// Connect to host
mysql_connect("$host", "$username", "$password");
// Select database
mysql_select_db("$db")or die("Cannot Select Database, $db");
?>
```

At this point a remote user will be able to type the domain name into the address filed of their web browser and the user login page will be loaded.

CONCLUSIONS AND IMPLICATIONS

The purpose of the project documented by this report was to provide to make available data to the field of L2A and to show that the data can be used to describe the phenomenon and refine its existing hypotheses. I've suggested that appropriate data, data which will move the field forward as a science, should meet the following criteria:

1. That the data derive from a study with more than five subjects (N > 5)
2. That subjects be adult (A > 12)
3. That subjects have reached a high proficiency in the L2 (+P)
4. That incubation (and time course of the study) be longer than five years (I > 5)
5. That the original data be made publicly available or (+O)

From the description of the history of the corpus, we know its data meets these criteria.

From the example project we found that this corpus of L2A data can be used to measure and describe attrition of a second language and that it adds to the understanding of the hypotheses of plateaus, thresholds and regression. These findings indicate that the use of this corpus can be used to advance the field of L2A. I have included the table of comparisons for studies and their findings again below, this time including the corpus described here.

STUDY	N>5	A>12	+P	I>5	+O	P	T	R
Stern & Stern			X					
Kennedy	X	X					+	
Kenyeres								
Geogehan	X	X						
Leopold			X					+
Scherer	X	X						
Pratella	X	X				-		
Burling			X					
Ravem			X	?				+
Smythe et. al.	X	X				+		
Cohen	X					-		
Cohen						+		+
Itoh & Hatch								
Magiste								
Bahrick	X	X		X			+	
Mauerman	X	X	X			+		
Olshtain	X	X				-		
Weltens	X	X	X			+	+	
Yoshida & Arai	X						+	
Smith	X	X	X			+		
Hansen	X	X	X	X			+	+
Reetz-Kurashige	X		X				+	
Russell	X	X	X			+		
Tomiyama			X			+		
Yoshitomi			X					+
Current Corpus	X	X	X	X	X	+	-	+

Table 5: comparison of studies, corpus included.

FUTURE WORK

The benefits of this project can be extended through additional work in these three areas: improving the corpus itself, improving tools for its use, improvements to the hosting website.

improve the corpus

It became apparent in the example project that a big advancement in the power of the corpus would be realized if it were to be marked up for correct use of the categories, in addition to its already showing and being searchable by incorrect use. This addition would require two things, an updated version of the XSD to validate the XML files which would result from this augmentation, and a lot of labor in reading through the tests themselves and hand coding of the data for correct use. Once complete, this would allow for a much broader range of queries to the corpus, more thorough analyses of the data and would result in a more complete understanding of the attrition phenomenon.

In addition to this supplement, there are other types of augmentation that could be made to the corpus which would also lead to progress in study of this phenomenon. Not intending this to be an exhaustive list, the following are some examples.

1. Adding the phonetic/phonemic transcription of the original audio files.
2. Correct usage
3. Biographical Data
4. Linking the text to the original audio files so that bytes may be played back.
5. Incorporating syntactic notations specific to one or more of the popular theories of syntax, e.g., GB/PP, Minimalism, LFG, etc.
6. Broaden the data

improve the tools

The tools for extracting the data and making and sharing analyses could be improved. Here is a list of some of the improvements of this nature.

1. The ability to create a copy of the corpus with the option to select what, if any of the error categories to include.
2. A user interface which allows for adding new categories to the set already specified.
3. A graphical interface for editing the markup, i.e., applying the new categories to the corpus without having to know XML syntax.
4. Tools for collaborative research efforts, e.g., the option to create a multiple-user account which allows for checking in and out of files.
5. Tools for analysis generation which do not require a knowledge of Perl.
6. Tools to merge to versions of the corpus, which were developed independently and have their own schemas, derived from a common schema

improve the website

Improvements to the website could include:

1. Development of an online validator for augmented XML files.
2. Making the tools listed above available via the internet.

REFERENCES

Active State, *Active Perl 5.8.8,* from http://www.activestate.com/ edn, 2007.

Bahrick, Harry P., 'Semantic Memory Content in Permastore: Fifty Years of Memory for Spanish Learned in School', *Journal of Experimental Psychology: General,* 113 (1984), 1-29.

Bray, T. and others, Extensible Markup Language (XML) 1.0 (Fourth Edition): W3C Recommendation 16 August 2006, http://www.w3.org/TR/2006/REC-xml-20060816/ edn, (W3C, 2006).

Burling, Robbins, 'Language Development of a Garo and English Speaking Child', in , ed. by Aron Bar-Adon and Werner F. Leopold(Englewood Cliffs, N.J.: Prentice-hall, 1971).

Cohen, Andrew D., 'Culver City Spanish Immersion Program: How does Summer Recess Affect Spanish Speaking Ability', *Language Learning,* 24 (1974), 55-68.

--------, 'Forgetting a Second Language', *Language Learning,* 25 (1975), 127-139.

Free Software Foundation, *GNU Emacs,* from http://www.gnu.org/software/emacs/ edn, 2007.

Geogehan, B., 'The Retention of Certain Secondary School Subject Matter Over the Period of Summer Vacation' (unpublished Doctorate, Fordham University, 1950).

Hansen, Lynn, 'Not a Total Loss: The Attrition of Japanese Negation Over Three Decades', in *Second Language Attrition in Japanese Contexts*, ed. by Lynn Hansen(New York: Oxford University Press, 1999), pp. 142-153.

Itoh, H. and E. Hatch, 'Second Language Acquisition: A Case Study', in *Second Language Acquisition*, ed. by E. Hatch(Rowley, MA: Newbury House, 1978), pp. 22-58.

Kennedy, L. R., 'The Retention of Certain Latin Syntactical Principles by First and Second Year Latin Students After various Time Intervals', *Journal of Educational Psychology,* 23 (1932), 132-146.

Kenyeres, A., 'Comment Une Petite Hongroise Apprend Le Francais', *Archives De Psychologie,* 26 (1938), 321-361.

Lambert, Richard D., 'Setting the Agenda', in *The Loss of Language Skills*, ed. by Richard D. Lambert and Barbara F. Freed(Rowley, MA: Newbury House Publishers Inc., 1982), pp. 6-10.

Leopold, Werner, *A Child's Learning of Two Languages,* Georgetown University Roundtable on Language and Linguistics, Washington DC edn, (Georgetown University Press, 1954).

Magiste, E., 'The Competing Language Systems of the Multilingual: A Development Study of Decoding and Encoding', *Journal of Verbal Learning and Verbal Behavior,* 18 (1979), 79-89.

Maurman, Peggy, 'Language Attrition in French-Speaking Missionaries' (unpublished Masters, Brigham Young University, 1985).

MySQL AB, *MySQL: The World's most Popular Open Source Database,* http://www.mysql.com/ edn, 2007 vols (2007).

Nakuma, Constancio K., 'Researching Fossilization and Second Language (L2) Attrition: Easy Questions, Difficult Answers', in *Studies of Fossilization in Second Language Acquisition*, ed. by ZhaoHong Han and Terence Odlin, First edn (Tonawanda, NY: Multilingual Matters Ltd., 2006), pp. 21-34.

Nation, I. S. P., *Learning Vocabulary in another Language*, Cambridge Applied Linguistics Series (Cambridge: Cambridge University Press, 2001).

Olshtain, Elite, 'The Attrition of English as a Second Language with Speakers of Hebrew', in *Language Attrition in Progress*, ed. by Bert Weltens, Kees de Bot and Theo van Els(Dordrecht, Holland: Floris Publications, 1986), pp. 187-204.

--------, 'Is Second Language Attrition the Reversal of Second Language Acquisition?', *Studies in Second Language Acquisition,* (1989), 151-165.

Oxford, Rebecca, 'Research on Language Loss: A Review with Implications for Foreign Language Teaching', *The Modern Language Journal,* 66 (1982), 160-169.

Pratella, W. C., The Retention of First and Second Year Spanish Over the Period of the Summer Vacation, 31 vols (1970).

Ravem, R., 'Language Acquisition in a Second Language Environment', in *Focus on the Learner*, ed. by J. Oller and J. Richards(Rowley, MA: Newbury House, 1973), pp. 136-144.

Reetz-Kurashige, Anita, 'Japanese Returnees' Retention of English-Speaking Skills: Changes in Verb Usage Over Time', in *Second Language Attrition in Japanese Contexts*, ed. by Lynn Hansen(New York: Oxford University Press, 1999), pp. 21-58.

Russell, Robert A., 'Lexical Maintenance and Attrition in Japanese as a Second Language', in *Second Language Attrition in Japanese Contexts*, ed. by Lynn Hansen(New York: Oxford University Press, 1999), pp. 114-141.

Scherer, George, 'The Forgetting Rate in Learning German', *German Quarterly,* 30 (1957), 275-277.

Smith, Michael T., 'Tracking Verb Usage in Adult Speakers of Spanish as a Second Language: An Attrition Study' (Masters, Brigham Young University, 1996).

Stern, C. and W. Stern, Die Kindersprache: Eine Psychologische and Sprachteoretishe Untersuchung (Leipzig: Barth, 1907).

Tomiyama, Machico, 'The First Stage of Second Language Attrition: A Case Study of a Japanese Returnee', in *Second Language Attrition in Japanese Contexts*, ed. by Lynn Hansen(New York: Oxford University Press, 1999), pp. 59-79.

Weltens, Bert, T. Van Els and E. Schils, 'The Long-Term Retention of French by Dutch Students', *Studies in Second Language Acquisition,* 11 (1989), 205-216.

Yoshida, K., *A Consideration on the Retention of Foreign Language by Returnees,* A Survey on the Foreign Language Retention of Returnees, 1 vols (Tokyo: Japan Overseas Student Education Foundation, 1989).

Yoshida, K. and K. Arai, *On the Retention of Foreign Listening Skills of Returnees,* A Survey on the Foreign Language Retention of Returnees, 2 vols (Tokyo: Japan Overseas Student Education Foundation, 1990).

Yoshitomi, Asako, 'On the Loss of English as a Second Language by Japanese Returnee Children', in *Second Language Attrition in Japanese Contexts*, ed. by Lynn Hansen(New York: Oxford University Press, 1999), pp. 80-111.

Appendix A: Situational Response Instrument

The instructions for this test were read aloud by a tester, in person or on an audio recording. The transcript follows:

```
Instructions:
On this section of the test you will be given various situations in
English and you must respond orally in Spanish. I will read each
situation to you, after which you will be given four minutes to record
your response. You should try to respond as naturally as possible to
each situation even though the microphone is your only audience. Now
listen and read along silently as I read situation number one.

Situation Number One:
As a missionary you have met a young couple who have expressed some
interest in learning about the LDS church. At the moment they only have
a few minutes to talk with you so you decide to tell them about Joseph
Smith and the first vision. Tell us in as much relevant detail as
possible the Joseph Smith story beginning with his interest in the
religions of his time and his attempts to find the truth.

Now I will activate the tape recorder and you may begin your response
to situation number one. Make sure your recording light is on and give
the number of the situation at the beginning of your response.

(4 minutes)

Please stop.

Now listen and read along silently as I read situation number two.

Situation Number Two:
You are talking with a close friend and he/she asks you what your plans
are over the next few years. Explain in detail what you expect or wish
to be doing over the next five to ten years. You may wish to include a
discussion of your goals and plans in some of the following areas:
education, career, marriage and family, where you wish to live, special
interests, etc.

Now I will activate the tape recorder and you may begin your response
to situation number one. Make sure your recording light is on and give
the number of the situation at the beginning of your response.

(4 minutes)

Please stop.

Now listen and read along as I read situation number three.

Situation Number Three:
As a missionary you have invited a young man to be baptized and he
responds that he does not understand the purpose of baptism. Explain to
```

him the teachings of the church regarding baptism and challenge him to accept your invitation.
Now check to make sure your record light is on and give the number of the situation.

(4 minutes)

Now listen again and read along silently as I read situation four.

Situation Number Four:
You are an exchange student in a Latin American university and have been invited to a gathering of fellow (mostly native Spanish-speaking) students. Everyone is taking turns introducing themselves. It is now your turn. Please tell everyone your name and where you are from. Tell them about your family, about your school and work experiences, your hobbies or interests and anything else about yourself or your family that you would care to.
Now check to make sure your record light is on and give the number of the situation.

(4 minutes)

Please stop.

Now listen again and read along silently as I read situation five.

Situation Number Five:
You are sitting in an airplane on your way home from your mission area to the United States. Sitting next to you is a young man from there who is being sent to America as a representative of his company. you strike up a conversation with him and he begins asking you about America. Based on your knowledge of the country and your experience living there, tell him all you can about the differences between America and that society and culture. You might talk, for example, about differences in work, food, housing, holidays, personal characteristics, the economy, sports, or anything else that occurs to you as being relevant.

(4 minutes)

please stop.

This is the end of the test.

APPENDIX B: PICTURE GUIDED NARRATION INSTRUMENT

This appendix contains 33 images with explanations. These images were taken from an actual test booklet, one image per page, as the subject saw them. The subject was also provided with two audio tape recording devices. One recorder held a tape with instructions, which the subject listened to. The other recorder held a blank tape, which the subject turned on to record their narration, then turned off again, according to instructions provided in the test booklet or on the instruction tape.

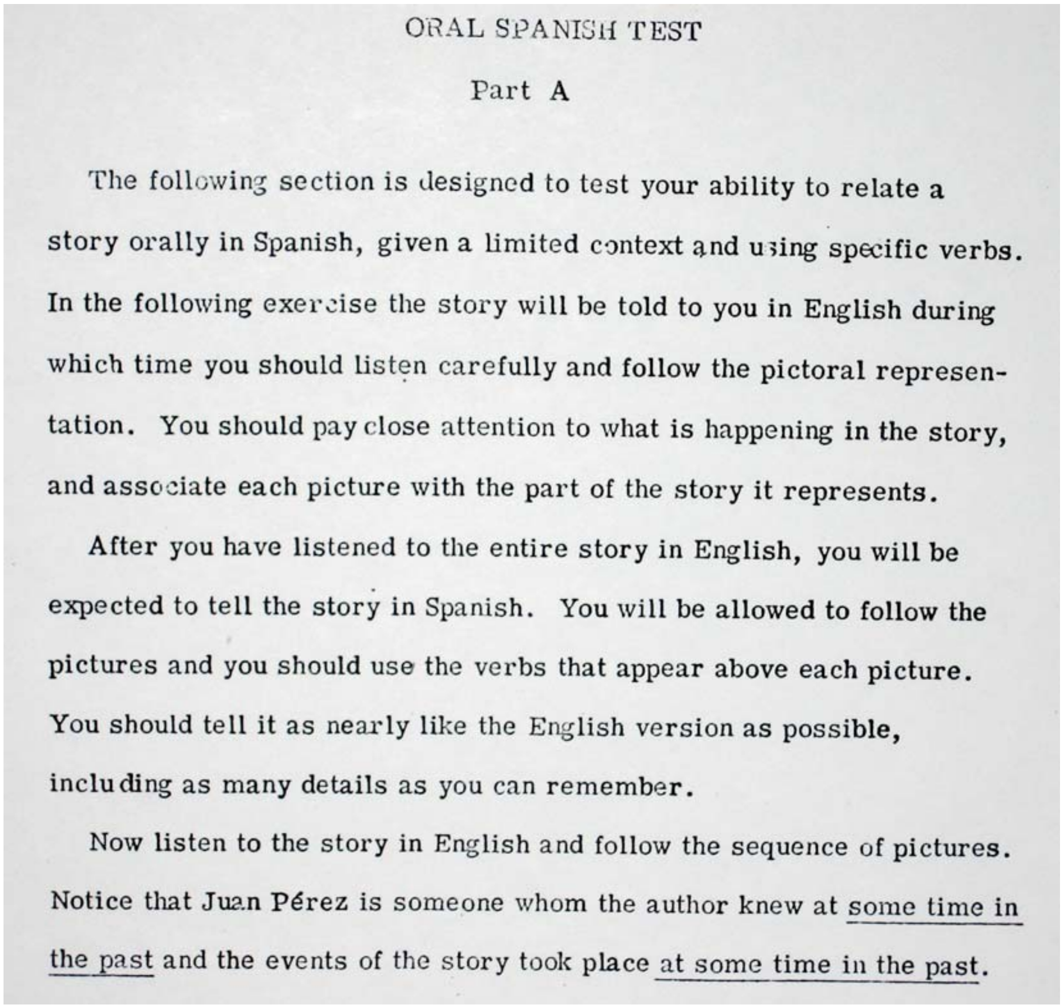
The booklet was comprised of eight narrations, called Part A, Part B, … Part H. Each part was designed to test a specific aspect of Spanish speaking ability (see Table 6.)

PART	ASPECT TESTED
A	Perfect and pluperfect aspects
B	Past tense verb conjugations and subjunctive mood
C	Present tense verb conjugations
D	Imperative mood, formal setting
E	Imperative mood, informal setting
F	Present and future tenses, various subjunctive constructions
G	Perfect aspect conjugations
H	Perfect and imperfect conjugations

Table 6: descriptions of guided narrations.

Part A is shown in Images 1-5 and described in detail. Images 6-34 cover Part B through Part H of the test booklet and are described in less detail, as the procedure followed by the subject during each part of the test was the same.

PART A



ORAL SPANISH TEST

Part A

The following section is designed to test your ability to relate a story orally in Spanish, given a limited context and using specific verbs. In the following exercise the story will be told to you in English during which time you should listen carefully and follow the pictoral representation. You should pay close attention to what is happening in the story, and associate each picture with the part of the story it represents.

After you have listened to the entire story in English, you will be expected to tell the story in Spanish. You will be allowed to follow the pictures and you should use the verbs that appear above each picture. You should tell it as nearly like the English version as possible, including as many details as you can remember.

Now listen to the story in English and follow the sequence of pictures. Notice that Juan Pérez is someone whom the author knew at some time in the past and the events of the story took place at some time in the past.

Image 1: part A, beginning instructions.

Image 1 shows the first page of the test booklet, which was the introduction to the first narration task, called Part A. Subsequent to the printing of this test booklet it was decided that verbs would not be listed above the pictures. In some images that follow a trace of whiteout, used to cover the verbs, is still visible.

The approximation of the story as the subject heard it told in English, before recording their own version in Spanish, is given here:

Juan Perez was a doctor. He lived on American Ave. He was married and had five children. One day he woke at six in the morning, but he was still tired. So, he went back to sleep until seven. He got up. He went into the bathroom, but there was no hot water and he did not like cold water. He went to the kitchen and put some water on to heat on the stove. Then he went back to the bathroom and brushed his teeth. When the water was hot he carried it to the bathroom to bathe with. His wife was preparing breakfast. She had got up early. She had already brought in the newspaper, swept the floor, made the bed and set the table and was almost finished making breakfast.

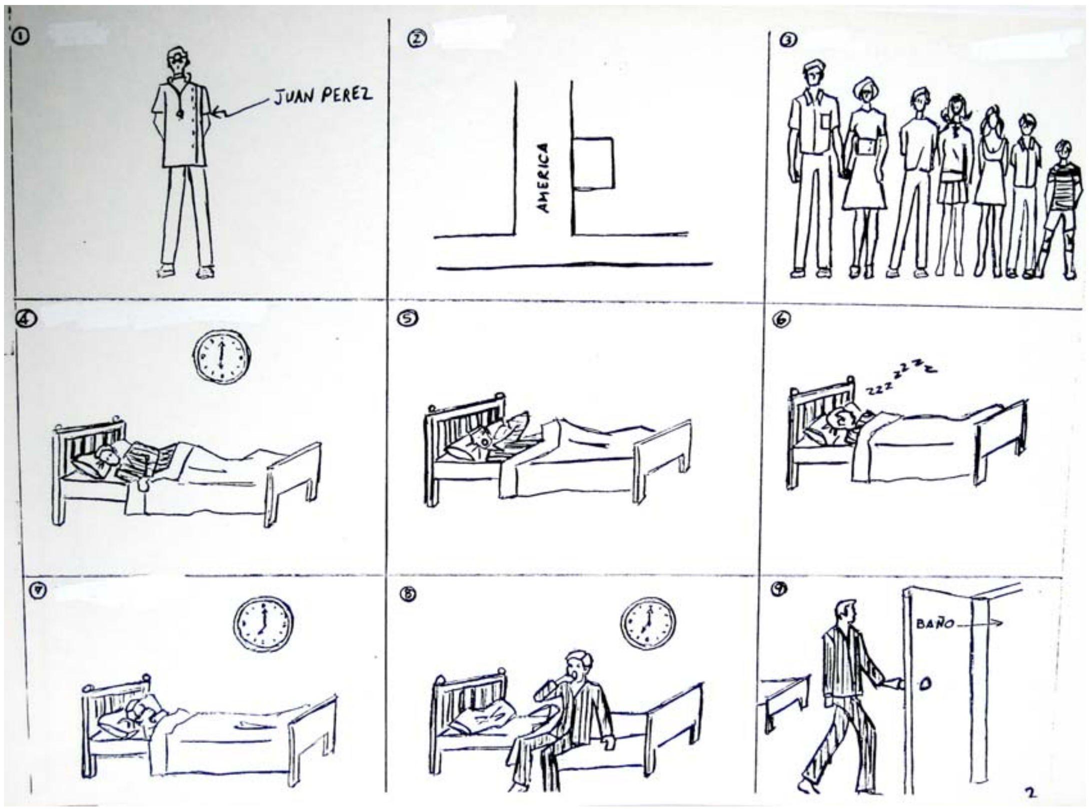


Image 2: part A, first page of pictures.

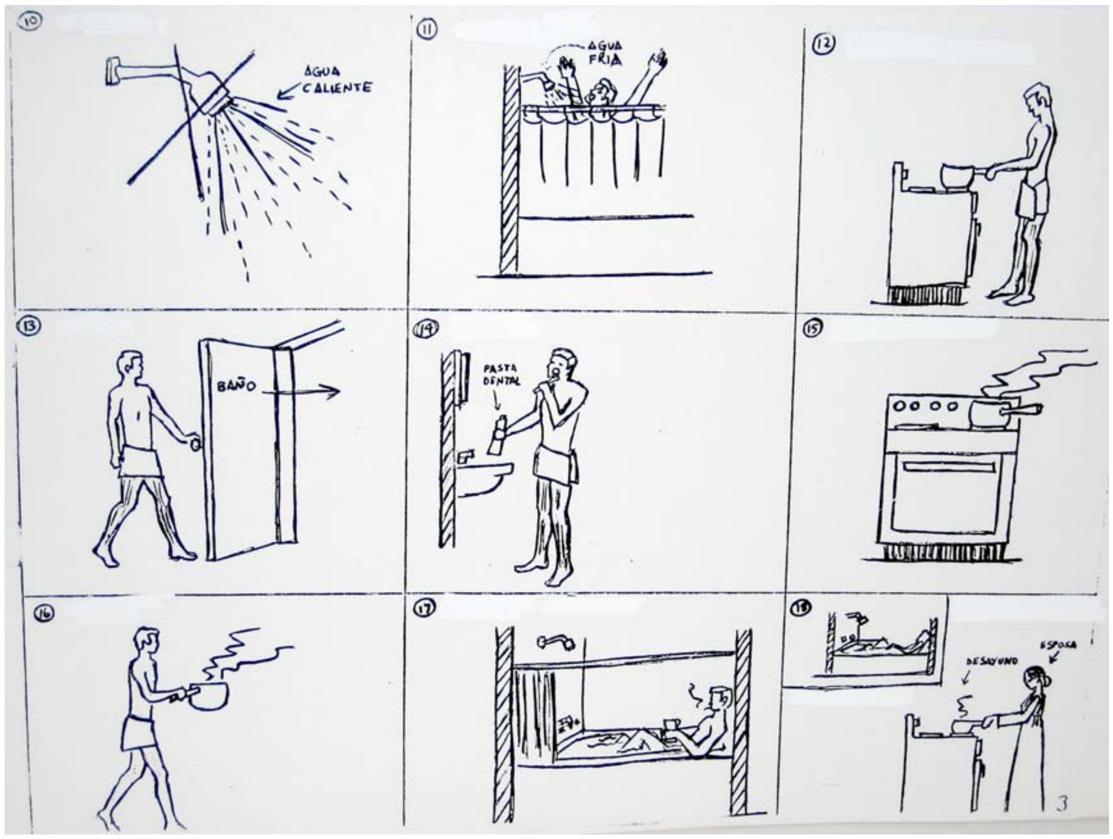


Image 3: part A, second page of pictures.

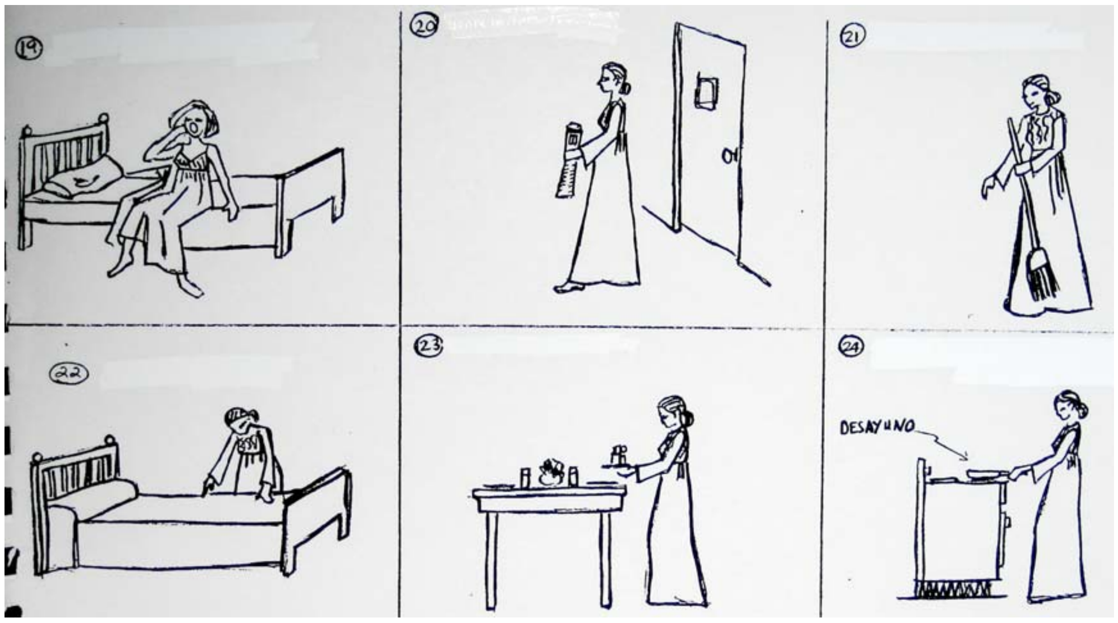
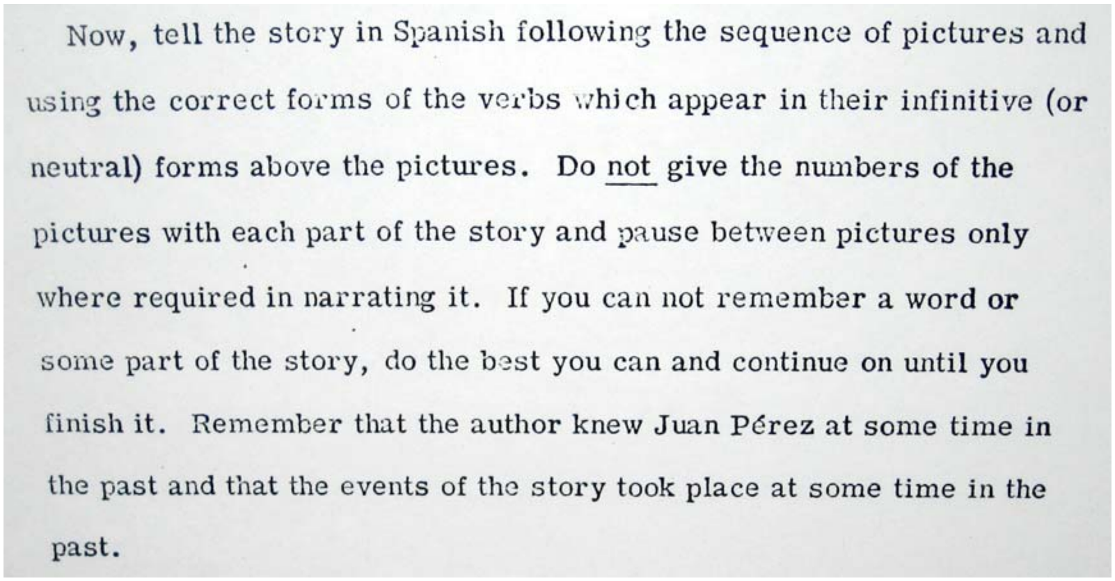


Image 4: part A, third page of pictures.



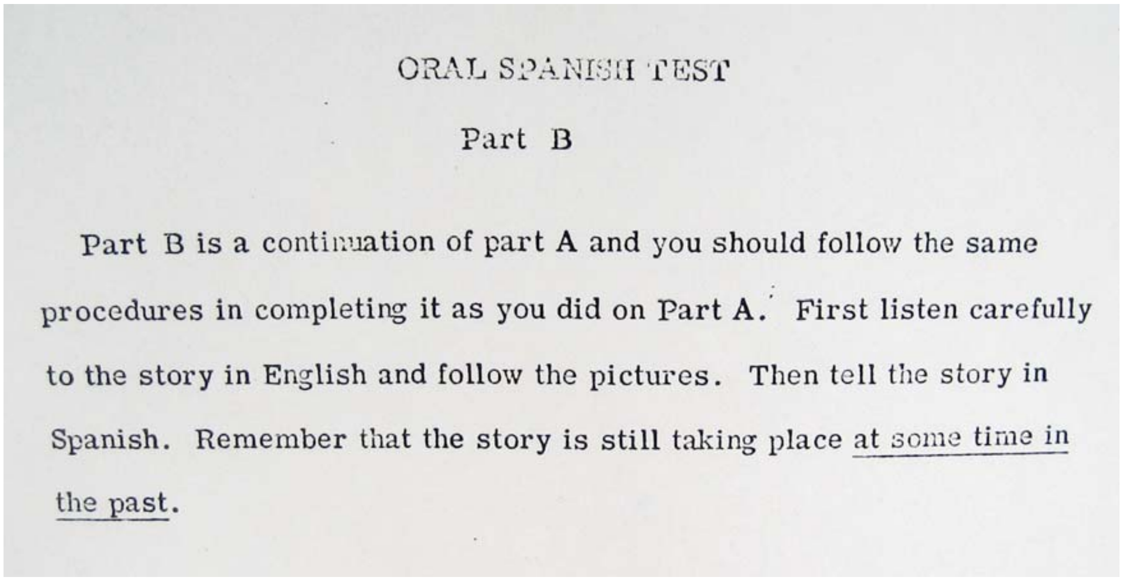
Now, tell the story in Spanish following the sequence of pictures and using the correct forms of the verbs which appear in their infinitive (or neutral) forms above the pictures. Do not give the numbers of the pictures with each part of the story and pause between pictures only where required in narrating it. If you can not remember a word or some part of the story, do the best you can and continue on until you finish it. Remember that the author knew Juan Pérez at some time in the past and that the events of the story took place at some time in the past.

Image 5: part A, concluding instructions.

Images 2-4 are the pages in the test booklet containing the 24 pictures that corresponded to the narration. The subject began by listened to the narration on the first tape recorder while following the pictures as the narration progressed. The subject was then instructed to stop the first tape player, start recording with the second one and to use the pictures as a guide while repeating the narration in their own words in Spanish. Image 5 shows the instructions which appeared after the pictures, which the subject read after hearing the story told in English and before retelling it in Spanish.

This narration required the subject to describe a series perfected actions of the past (preterite tense/aspect) and then describe a second set that occurred before the first set (pluperfect tense/aspect). Since the verbs required to retell the story are very common and because the requirement to use the past tense to relate a story is also very common, the speech produced was very close to, perhaps precisely what it would have been, if the subject had been engaged in an actual conversation.

PART B


ORAL SPANISH TEST

Part B

Part B is a continuation of part A and you should follow the same procedures in completing it as you did on Part A. First listen carefully to the story in English and follow the pictures. Then tell the story in Spanish. Remember that the story is still taking place <u>at some time in the past</u>.

Image 6: part B, beginning instructions.

The approximation of the story as the subject heard it told in English, before recording their own version in Spanish:

> Finally it was eight thirty. He had gotten dressed, eaten breakfast and had read the newspaper. He was ready to go to work. He told his wife that he was going to leave the office at four in the afternoon because he had to visit a friend. He asked her to make dinner and to have it ready at six because he and his friend would arrive and they were going to be hungry. He told her that they needed to have something special to drink and asked her to go to the supermarket to buy some wine.

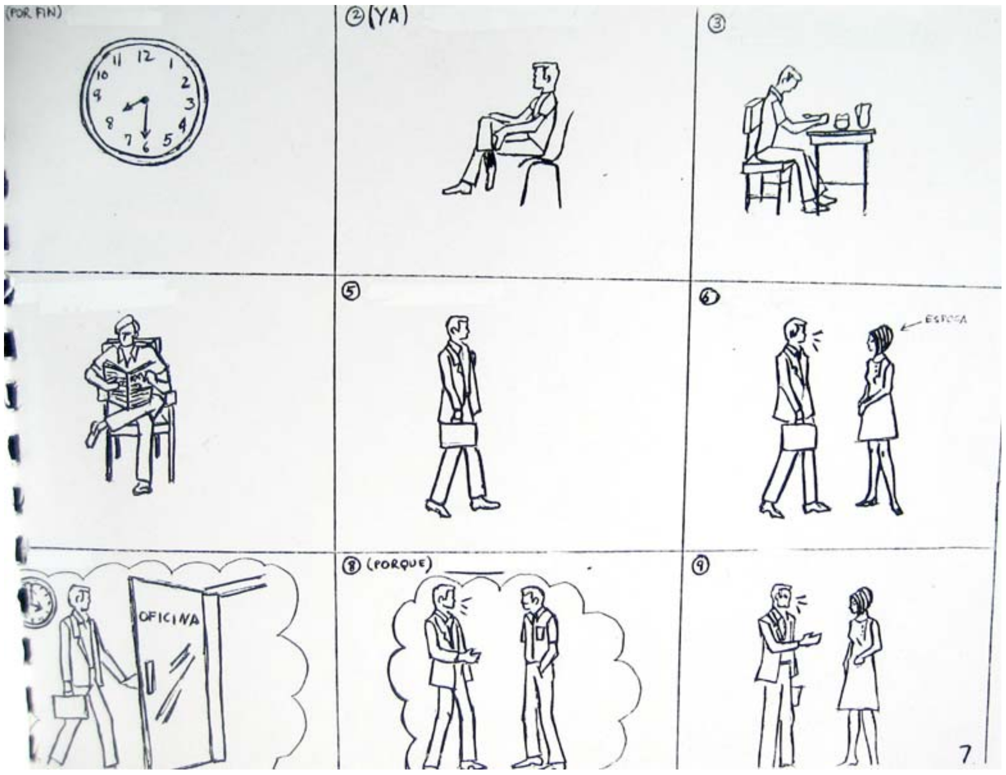


Image 7: part B, first page of pictures.

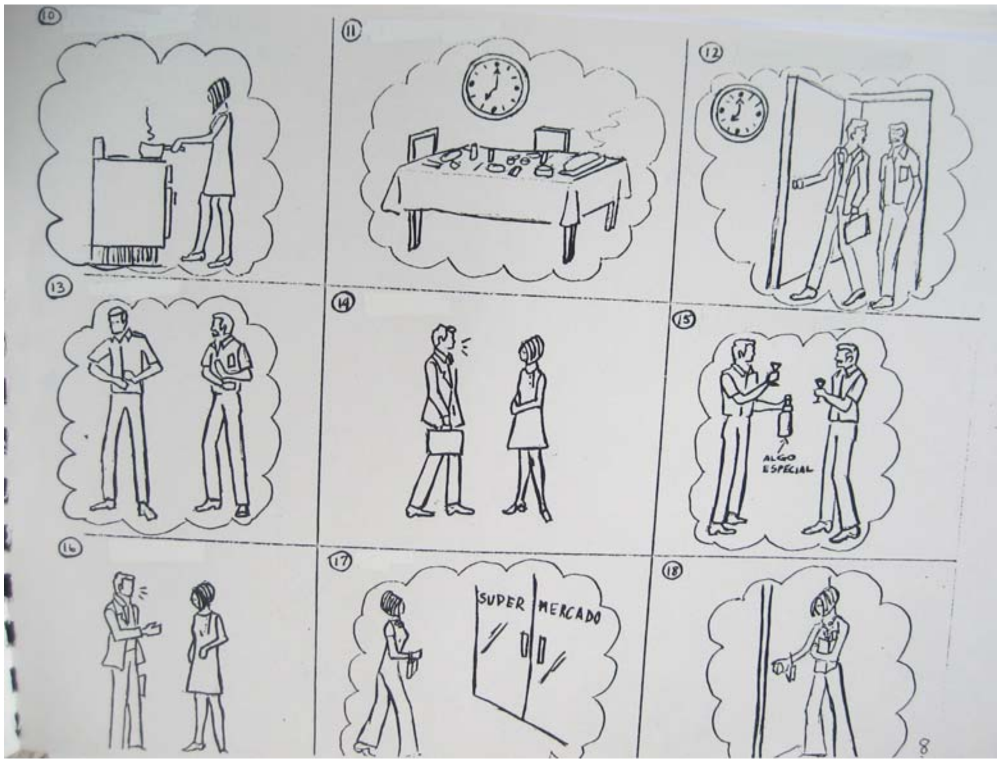
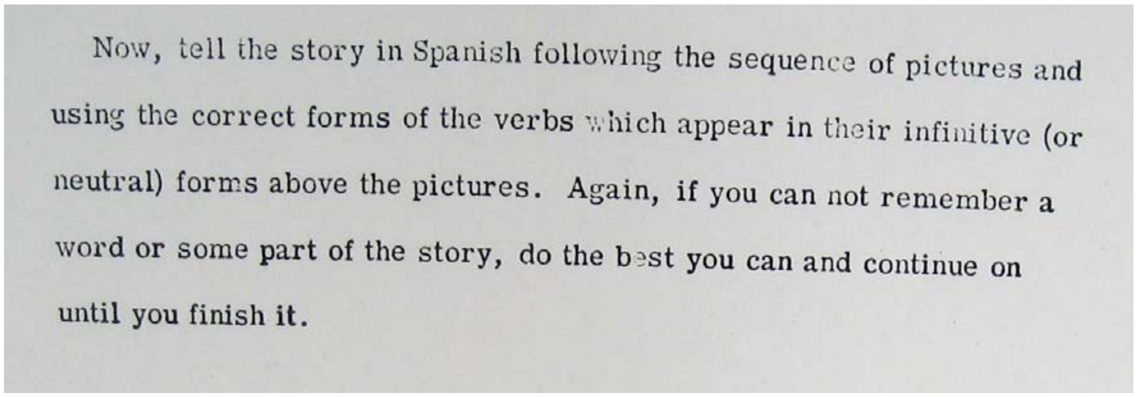


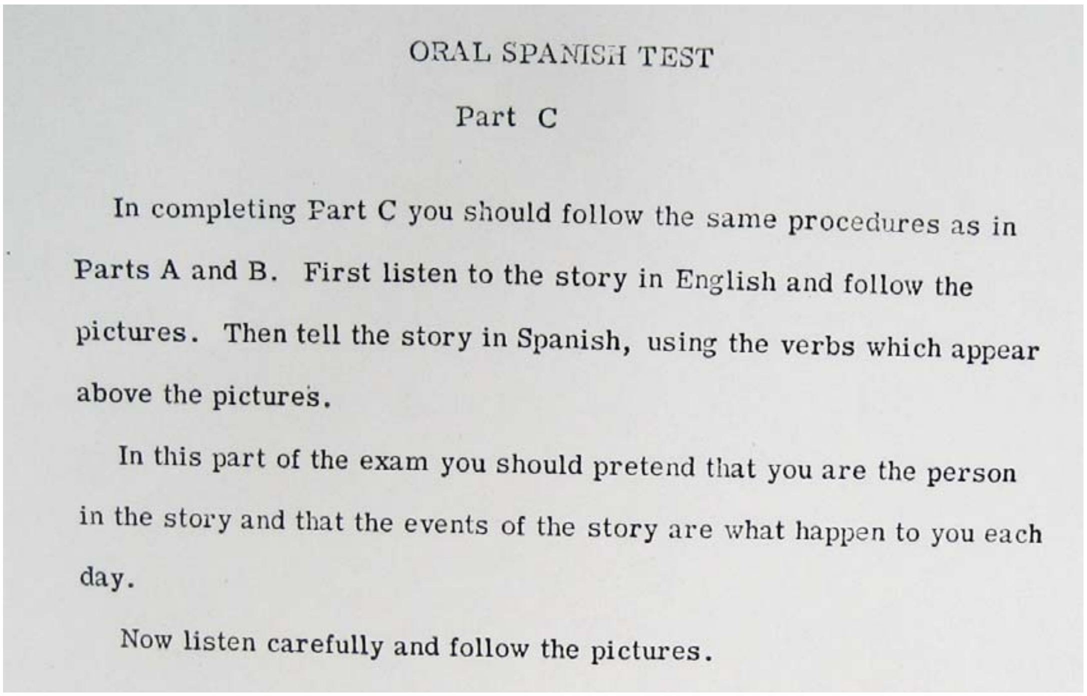
Image 8: part B, second page of pictures.



Now, tell the story in Spanish following the sequence of pictures and using the correct forms of the verbs which appear in their infinitive (or neutral) forms above the pictures. Again, if you can not remember a word or some part of the story, do the best you can and continue on until you finish it.

Image 9: part B, concluding instructions.

PART C



ORAL SPANISH TEST

Part C

In completing Part C you should follow the same procedures as in Parts A and B. First listen to the story in English and follow the pictures. Then tell the story in Spanish, using the verbs which appear above the pictures.

In this part of the exam you should pretend that you are the person in the story and that the events of the story are what happen to you each day.

Now listen carefully and follow the pictures.

Image 10: part C, beginning instructions.

The approximation of the story as the subject heard it told in English, before recording their own version in Spanish:

```
I sleep until six in the morning. Then I wake up. I get up. I get
dressed. I eat breakfast. I read the newspaper and at eight I'm ready
to go to work. I leave and go to the bus stop to catch a bus. I go down
```

Bolivar St. I get off the bus and walk to the office. When I arrive I ask the secretary to make coffee and bring me some, then I start to work. First I find out if there are letters to be written. Then I put all of today's work in one pile and other work in another pile. I work all day long and leave the office at five in the evening. I go to the basketball court and play basketball with my friends. Then I go to my house. I eat dinner and I go to bed.

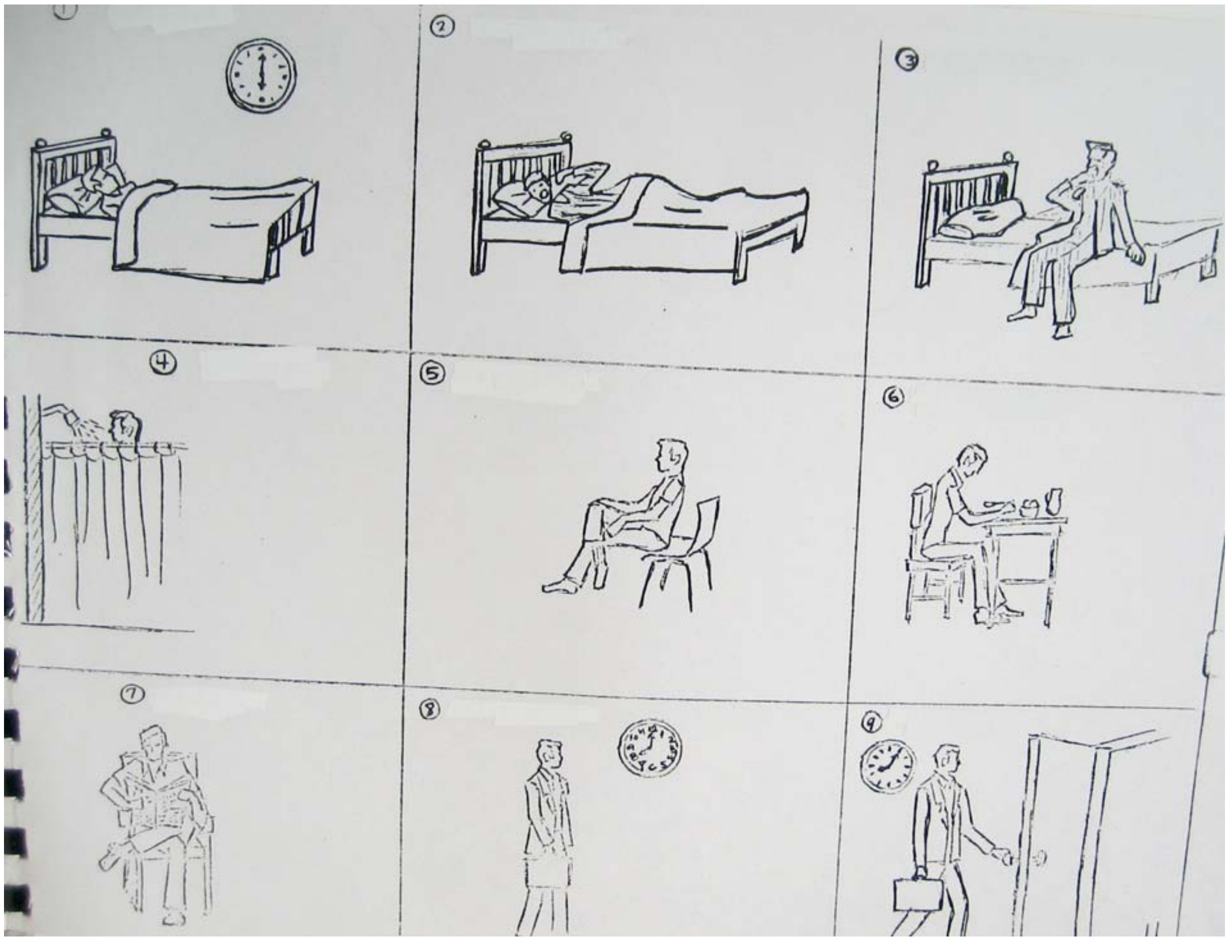


Image 11: part C, first page of pictures.

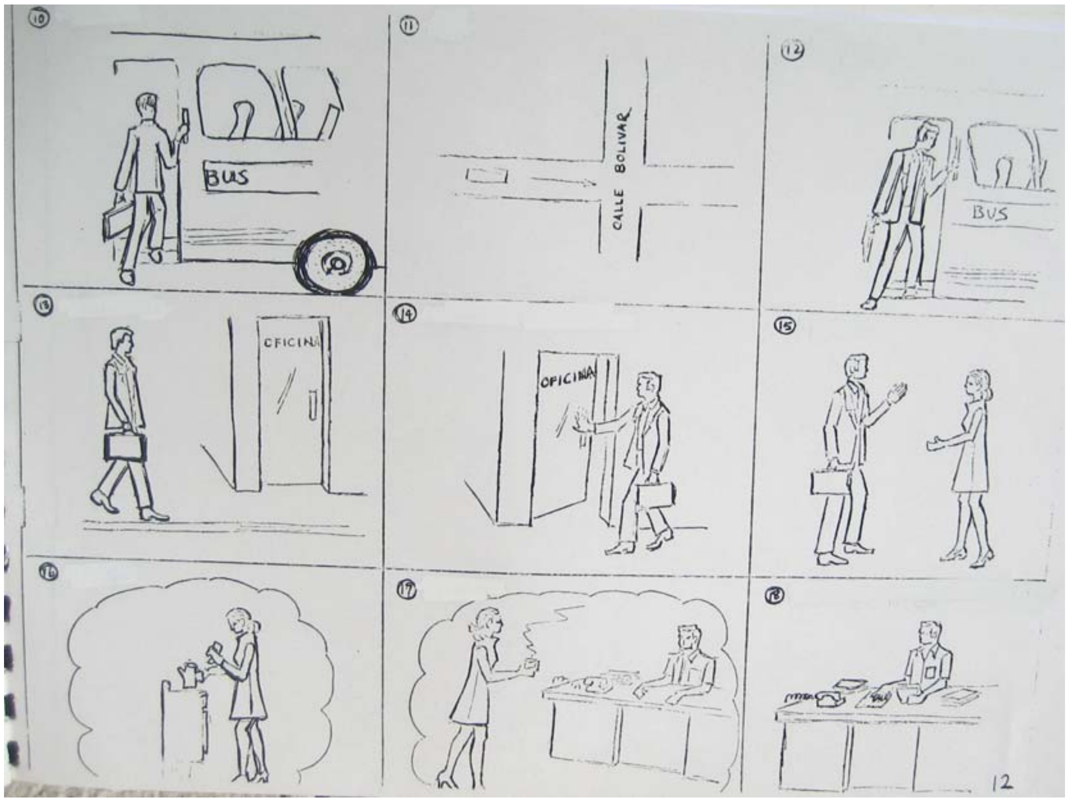


Image 12: part C, second page of pictures.

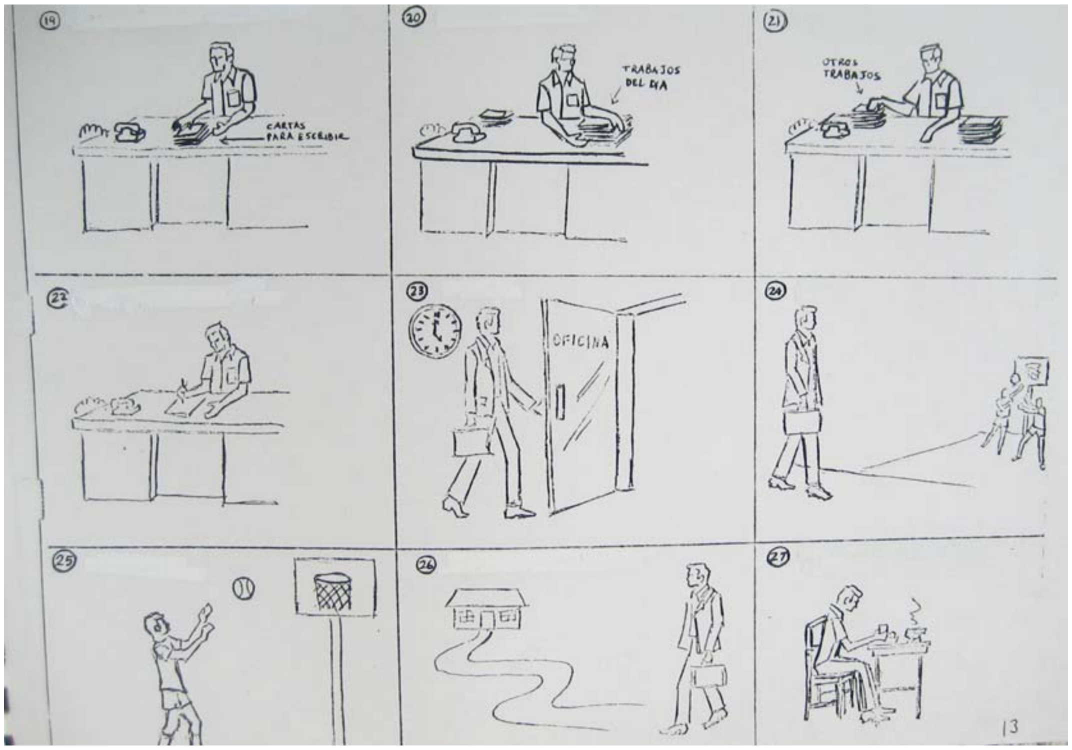


Image 13: part C, third page of pictures.

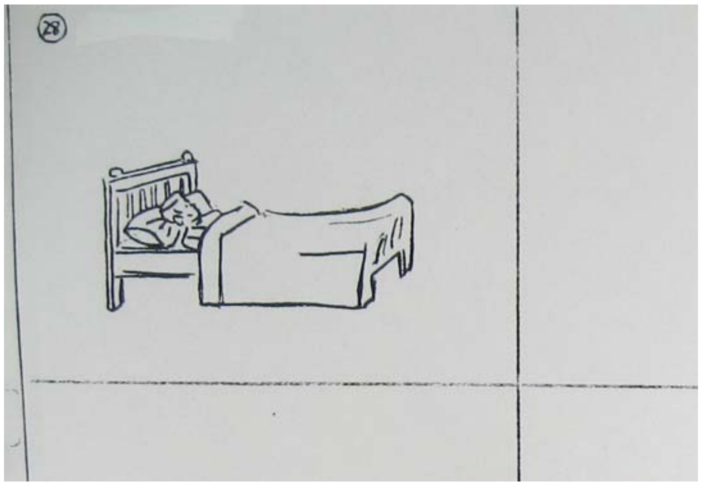
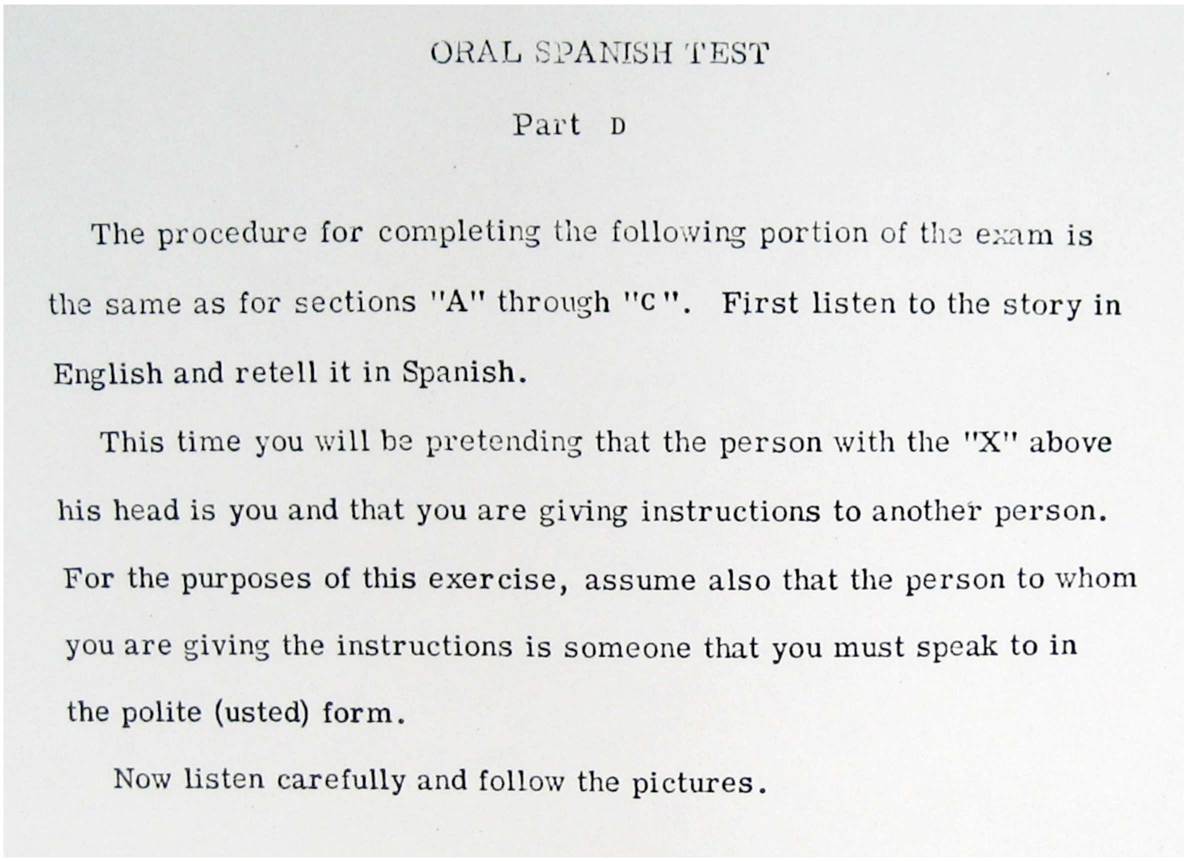


Image 14: part C, fourth page of pictures.

PART D



ORAL SPANISH TEST

Part D

The procedure for completing the following portion of the exam is the same as for sections "A" through "C". First listen to the story in English and retell it in Spanish.

This time you will be pretending that the person with the "X" above his head is you and that you are giving instructions to another person. For the purposes of this exercise, assume also that the person to whom you are giving the instructions is someone that you must speak to in the polite (usted) form.

Now listen carefully and follow the pictures.

Image 15: part D, beginning instructions.

The approximation of the story as the subject heard it told in English, before recording their own version in Spanish:

> I have a sack of money here. I want you to take it to Popular Bank. To get there, leave and shut the door behind you. Then, cross the street and be careful for traffic. Turn left at the corner and go strait three blocks. You will see a big building there with a sign in front that says Popular Bank. Go in the bank and go to the window. You will find a man there named Pedro Lopez. Give him the money and tell him to count it and to put it in my savings account. Tell him to come to my office this afternoon and to bring me a thousand dollars.

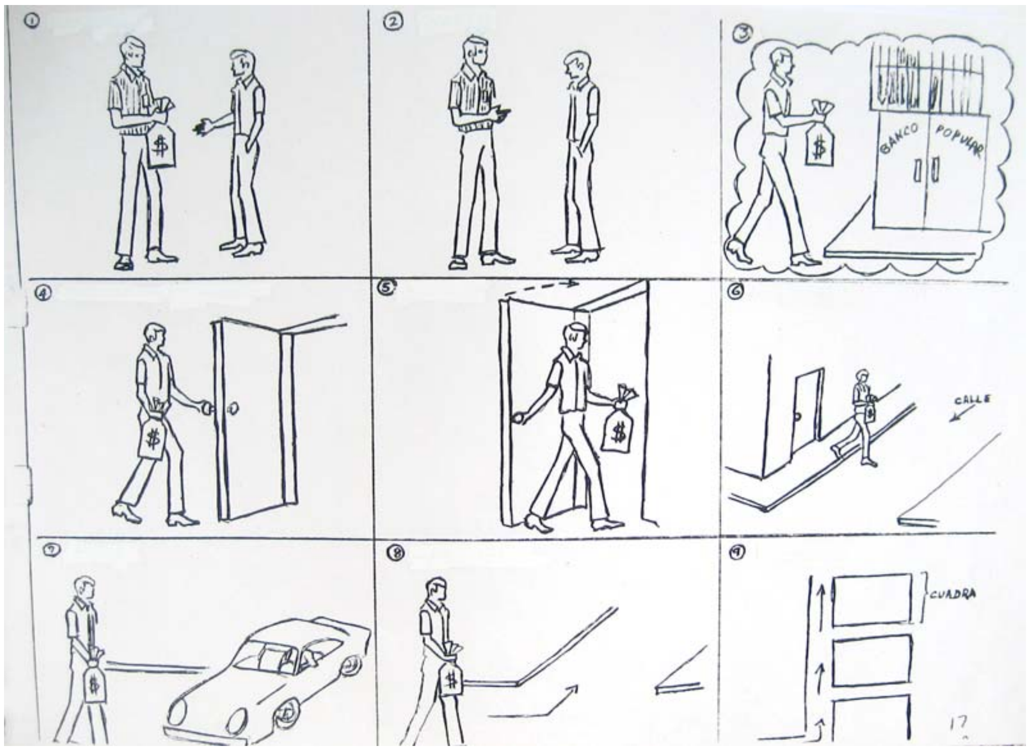


Image 16: part D, first page of pictures.

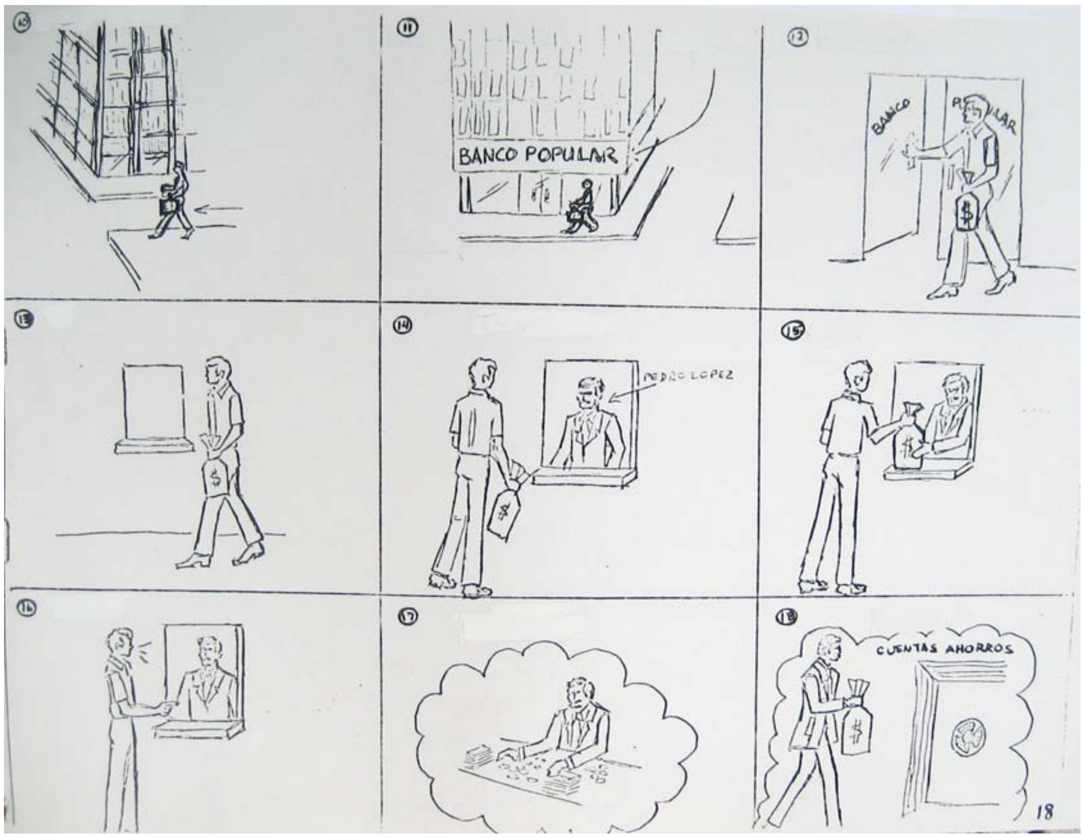


Image 17: part D, second page of pictures.

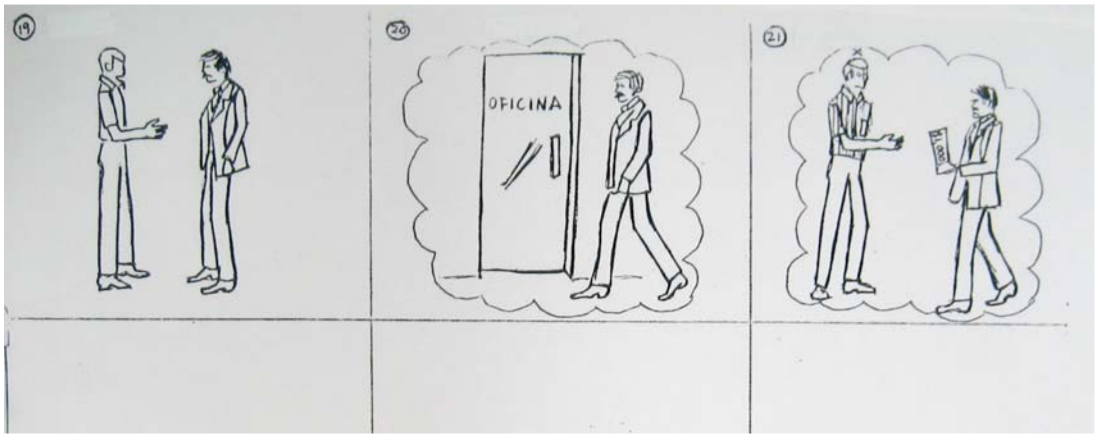
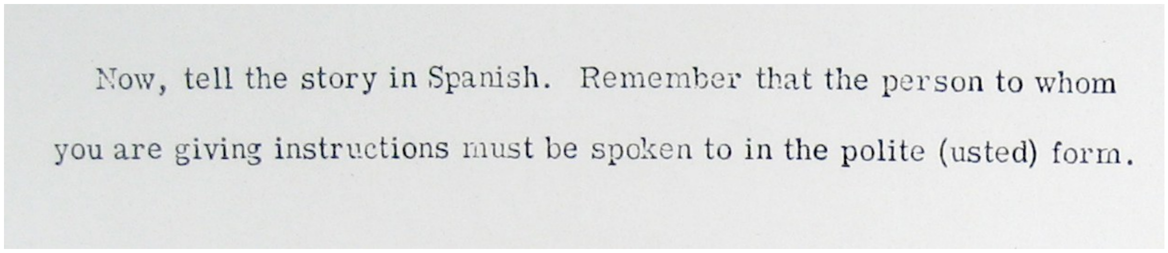


Image 18: part D, third page of pictures.

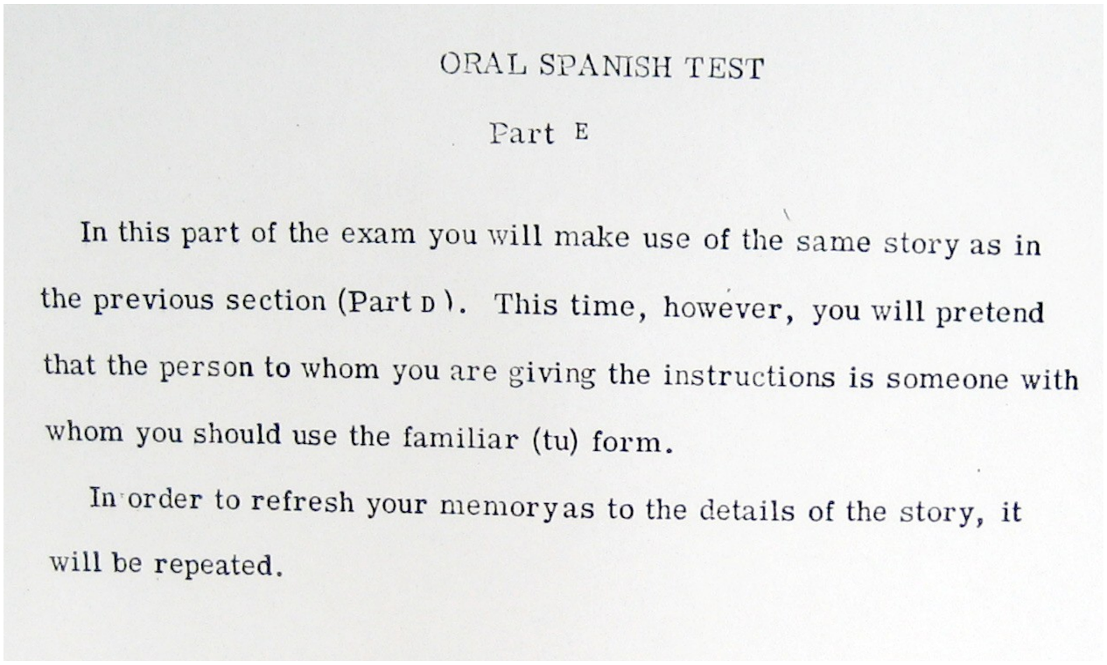


Now, tell the story in Spanish. Remember that the person to whom you are giving instructions must be spoken to in the polite (usted) form.

Image 19: part D, concluding instructions.

PART E

Part E contained the same pictures as part D. The instructions were to use informal rather than formal or polite speech (see Images 19-20.)



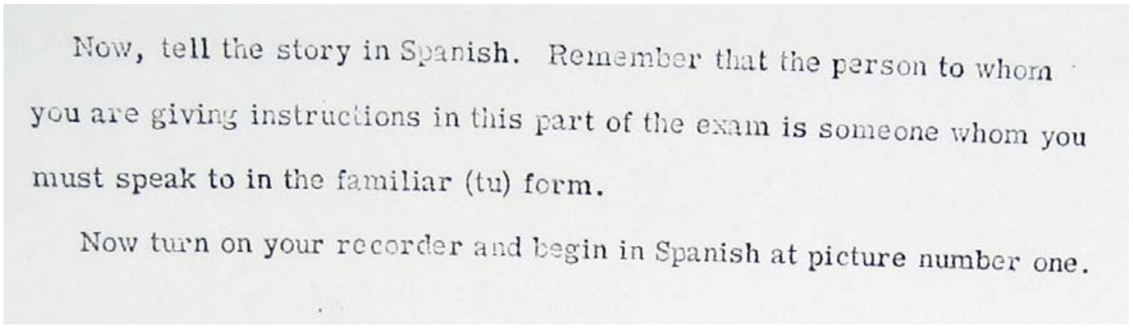
ORAL SPANISH TEST

Part E

In this part of the exam you will make use of the same story as in the previous section (Part D). This time, however, you will pretend that the person to whom you are giving the instructions is someone with whom you should use the familiar (tu) form.

In order to refresh your memory as to the details of the story, it will be repeated.

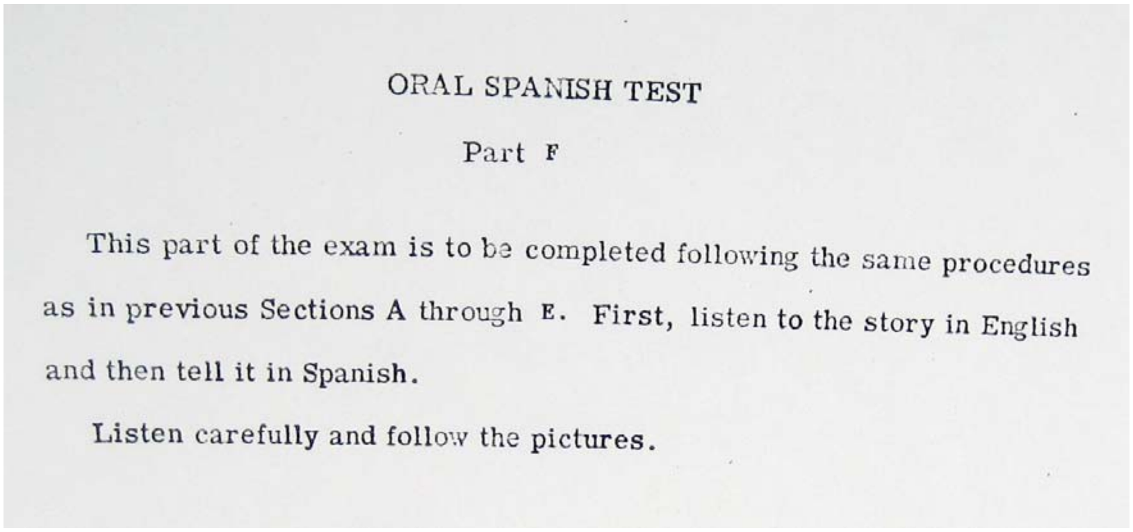
Image 20: part E, beginning instructions.


Now, tell the story in Spanish. Remember that the person to whom you are giving instructions in this part of the exam is someone whom you must speak to in the familiar (tu) form.

Now turn on your recorder and begin in Spanish at picture number one.

Image 21: part E, concluding instructions.

PART F


ORAL SPANISH TEST

Part F

This part of the exam is to be completed following the same procedures as in previous Sections A through E. First, listen to the story in English and then tell it in Spanish.

Listen carefully and follow the pictures.

Image 22: part F, beginning instructions.

The approximation of the story as the subject heard it told in English, before recording their own version in Spanish:

```
Juan is a missionary. He is going to work in Venezuela. He will leave
soon by airplane for Venezuela. When he gets there he will have one
more week of training, then he will leave for his area. It's possible
that he'll go to a remote area, and it's possible that they will give
him a motorcycle. But for sure they won't give him a car. He doesn't
care whether the give him a motorcycle or not. He just wants to have
hot water. It doesn't matter whether the house is old or that it may
not have furniture. He doesn't think he can live without hot water. The
more hot water there is, the happier he will be.
```

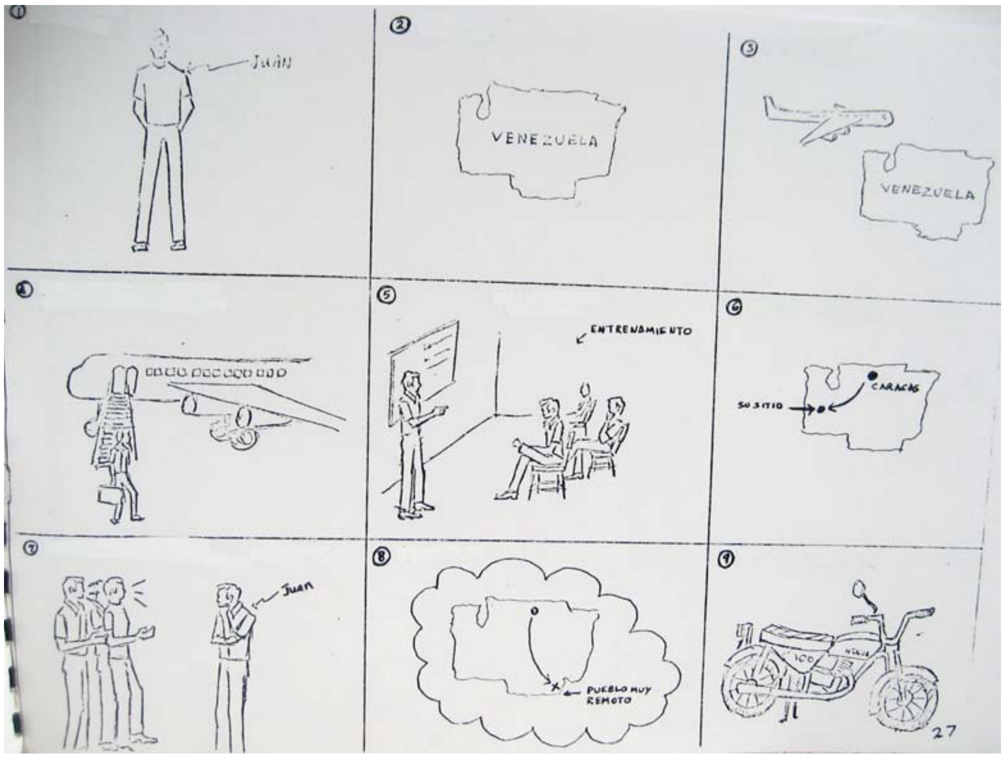


Image 23: part F, first page of pictures.

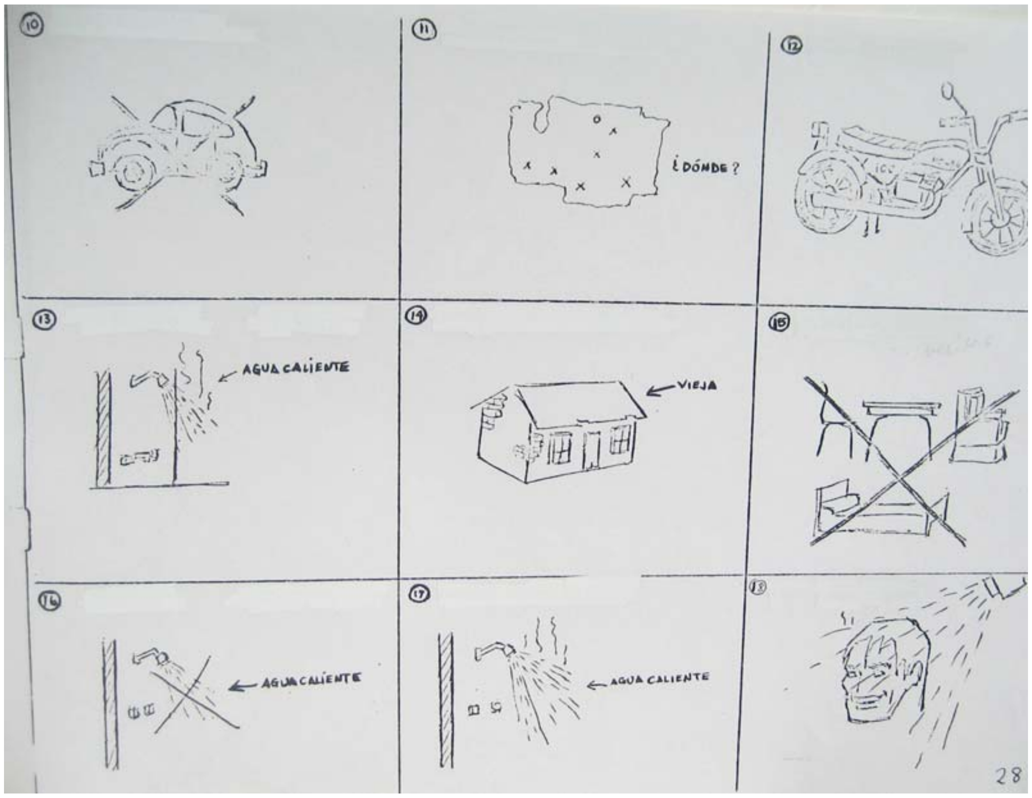
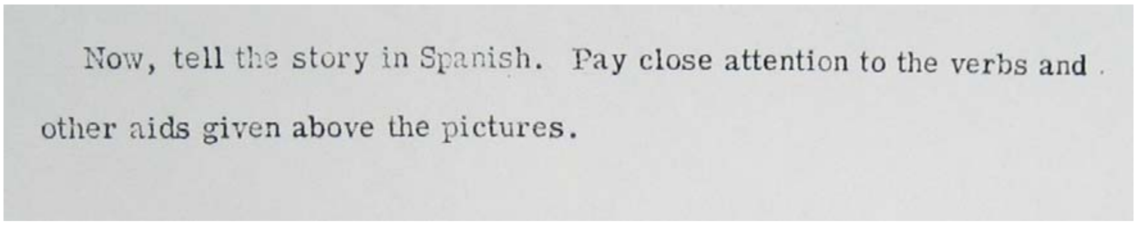


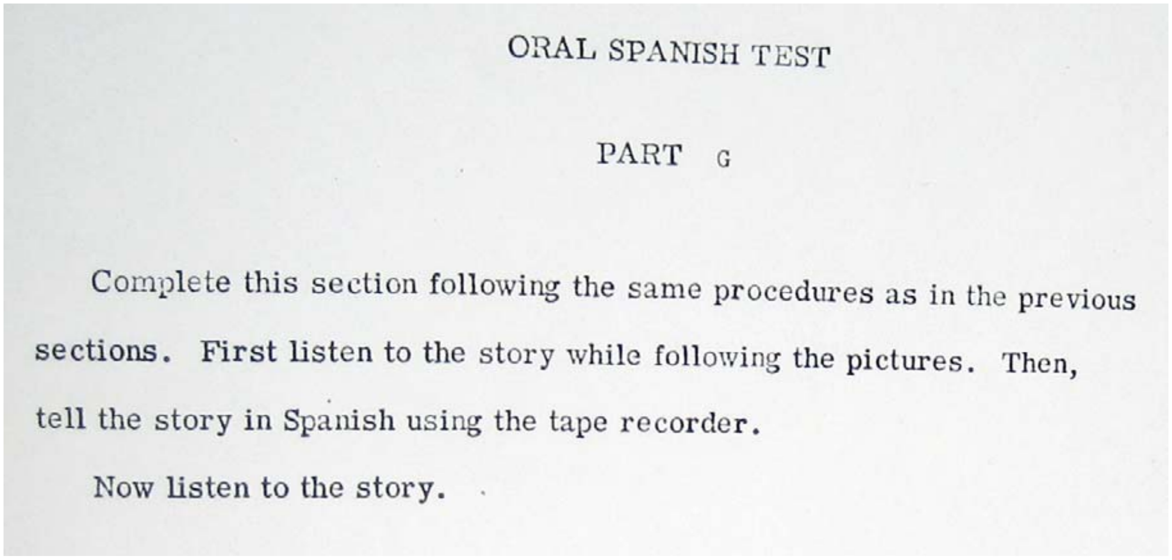
Image 24: part F, second page of pictures.



Now, tell the story in Spanish. Pay close attention to the verbs and other aids given above the pictures.

Image 25: part F, concluding instructions.

PART G



ORAL SPANISH TEST

PART G

Complete this section following the same procedures as in the previous sections. First listen to the story while following the pictures. Then, tell the story in Spanish using the tape recorder.

Now listen to the story.

Image 26: part G, beginning instructions.

The approximation of the story as the subject heard it told in English, before recording their own version in Spanish:

```
Juan brought a cup of coffee to Maria. She took it and said thanks. He
said you're welcome. She tried it, but it needed sugar. So, she put it
on the table and Juan brought some sugar. She put sugar in it and tried
it again. This time she liked it and drank it all. Juan was very happy
and held her hand. Then he left.
```

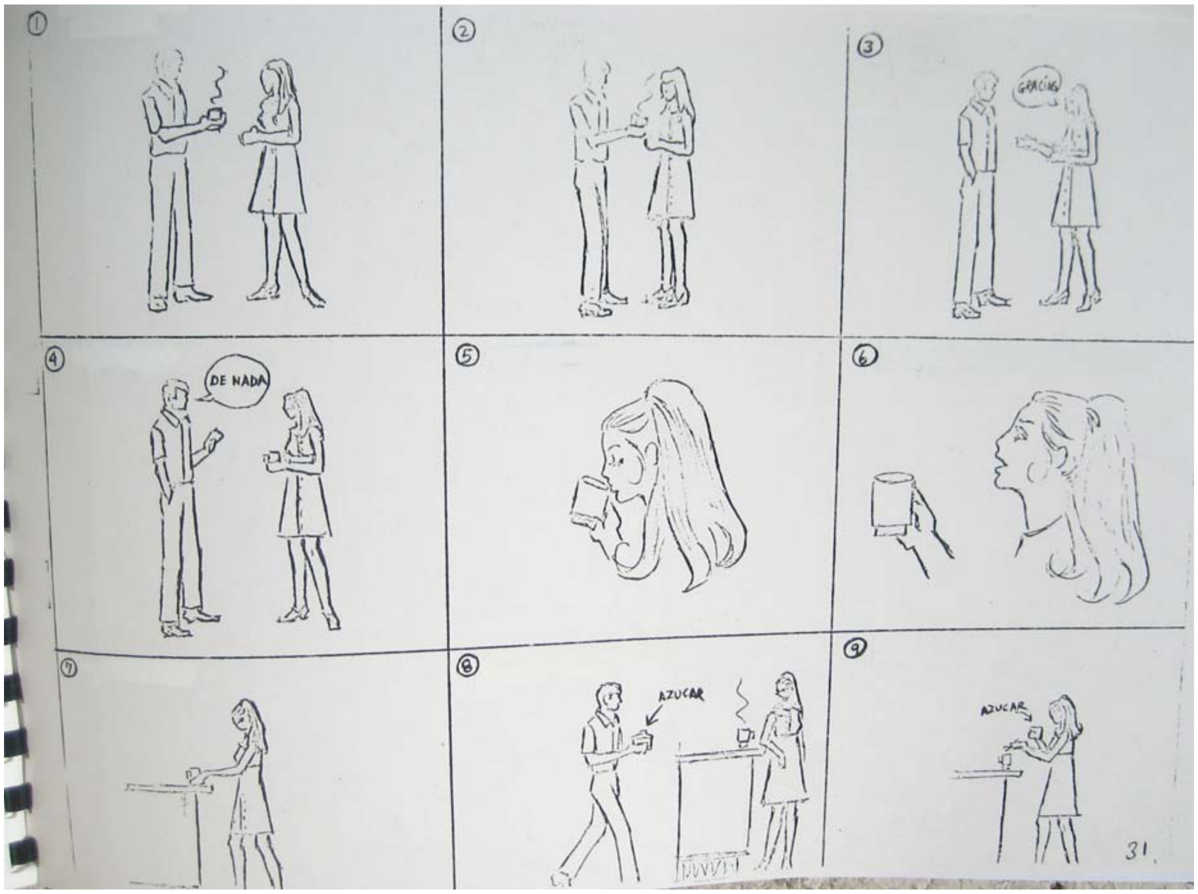


Image 27: part G, first page of pictures.

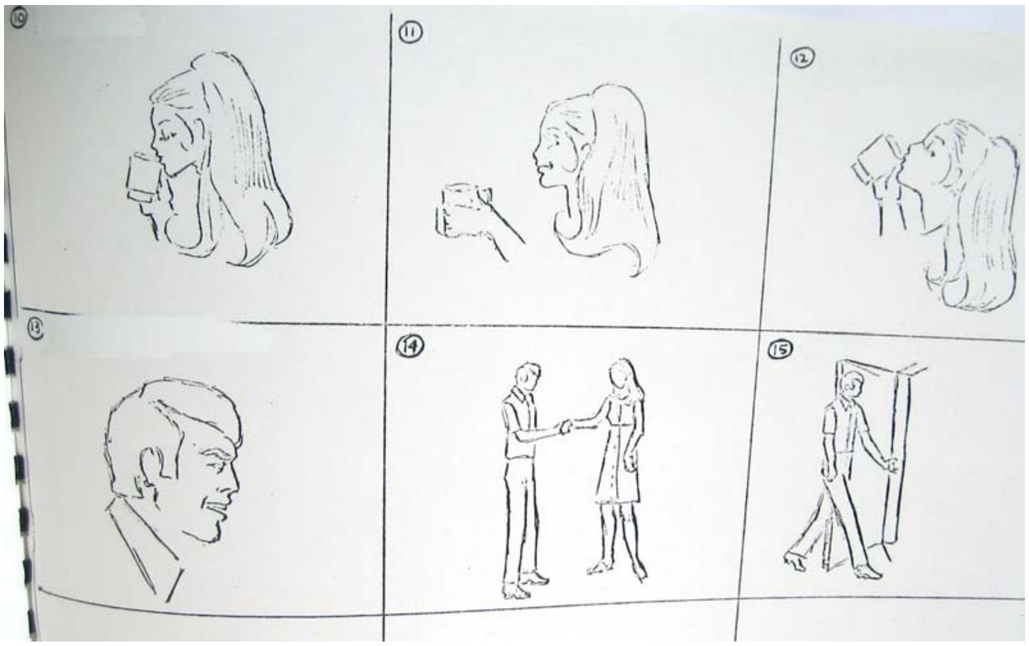
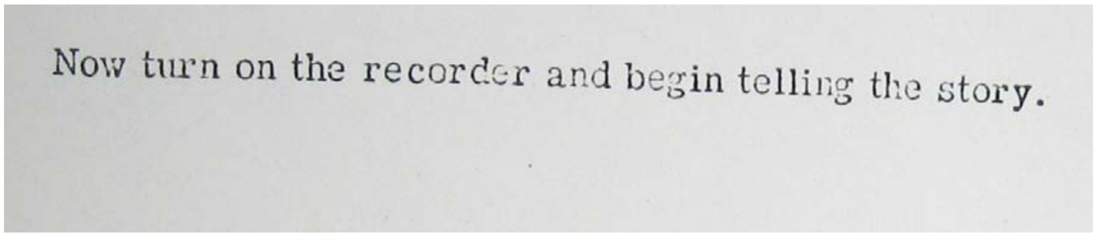


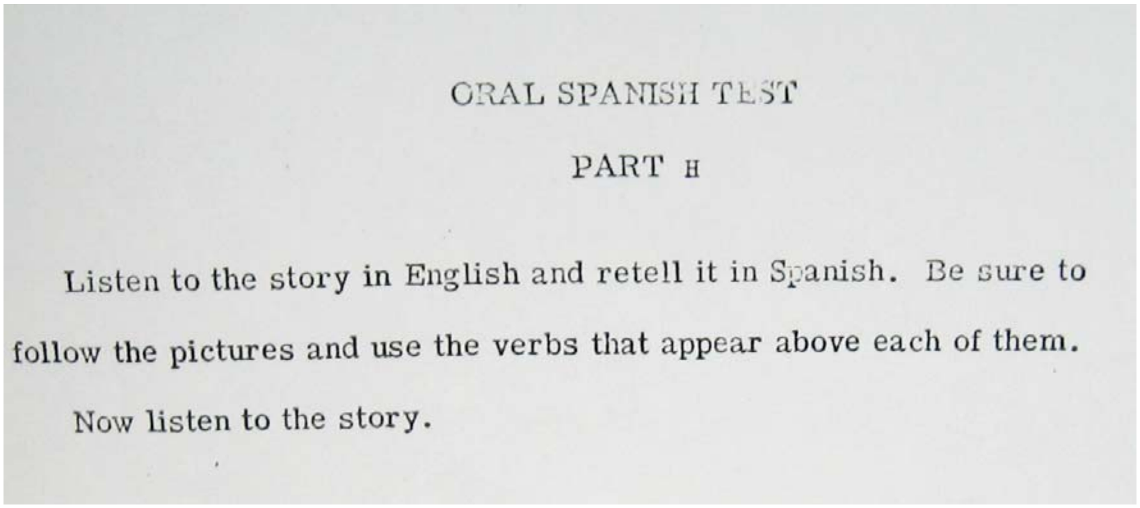
Image 28: part G, second page of pictures.



Now turn on the recorder and begin telling the story.

Image 29: part G, concluding instructions.

PART H



ORAL SPANISH TEST

PART H

Listen to the story in English and retell it in Spanish. Be sure to follow the pictures and use the verbs that appear above each of them.

Now listen to the story.

Image 30: part H, beginning instructions.

The approximation of the story as the subject heard it told in English, before recording their own version in Spanish:

```
Juan chose a book. When he went to pay for it, he realized that he had
left his wallet at home on the dresser. So, he wrote a check and took
the book to read it to his son who was sick. When he was leaving the
bookstore the book fell. He picked it up and got on the bus. The book
fell again. This time a man picked it up and gave it to him, but this
time a page had torn. Then he arrived at home. He began reading the
book. He finished at nine and gave the book to his wife and she put it
on the stand.
```

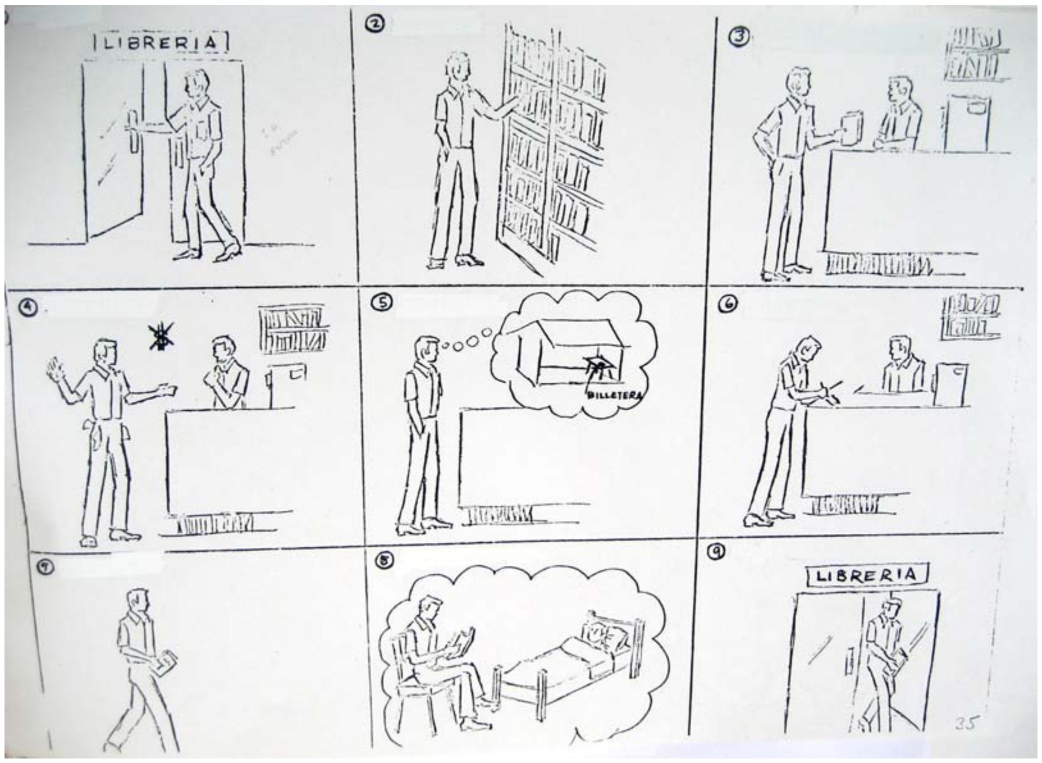


Image 31: part H, first page of pictures.

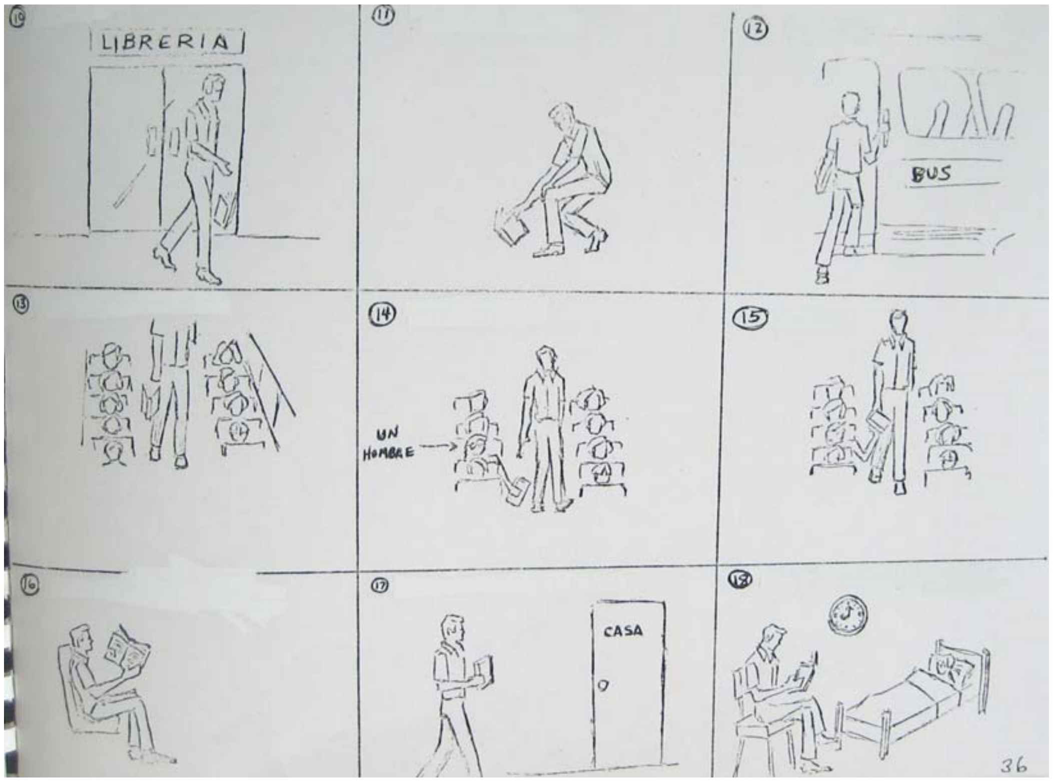


Image 32: part H, second page of pictures.

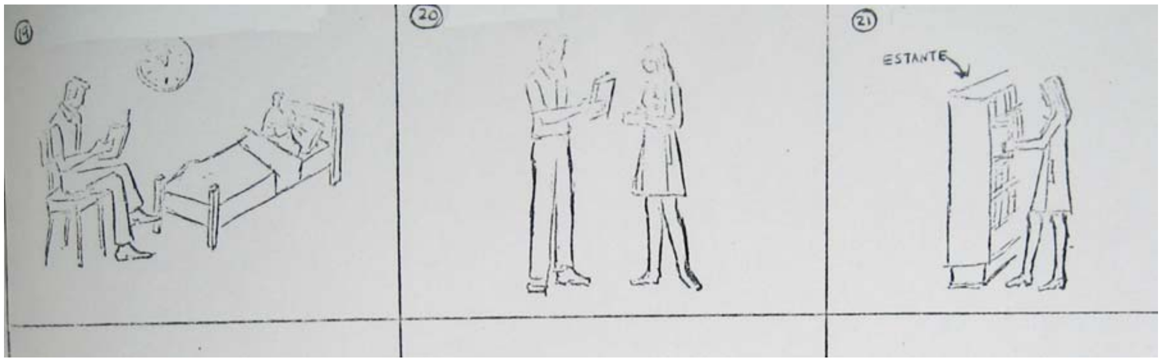
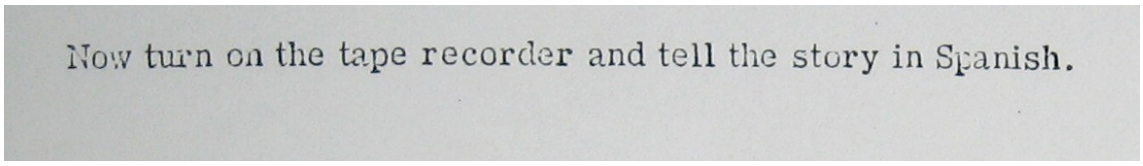


Image 33: part H, third page of pictures.



Now turn on the tape recorder and tell the story in Spanish.

Image 34: part H, concluding instructions.

APPENDIX C: XML TUTORIAL

XML IN GENERAL

XML stands for Extensible Markup Language. It is the most prevalent format for storing and using data on the world wide web and in many other sectors of the digital world. Countless applications exist which rely on the XML format to do their job, and almost every website on the web uses XML in one way or another.

XML structure, or syntax, is standardized. Since any data that is preserved in XML abides by this standard, it can be used or processed by any program that uses XML data.

What makes XML so popular, though, is that its syntax can be adapted or extended for particular purposes, as it has with this corpus. This adaptation of the XML syntax is done by designing a data schema (see appendix E for the schema used to validate this corpus.)

Once you have some data, the process of converting it to valid XML has four steps:

Decide how you want your data to be organized.

Write the schema to reflect that organization.

Organize the data according to the schema.

Use a validator to check your work.

Once this process is complete and your data is valid according to your schema, you can rely on the fact that you know exactly how it is all organized to then start analyzing it.

XML SYNTAX

XML syntax revolves around what are called elements. Each element has three main parts to it. They are:

The start tag

Attributes

The end tag

Here is a line of XML code.

```
<foo>bar</foo>
```

The start tag is the<foo> and the end tag is the</foo>. This element has no attribute in it, but it does have content, the word 'bar'. This is the content because it is contained by or nested in, the element.

If an element has no content, the start and end tags are merged into one element called an empty element.

```
<foo/>
```

Here is the same element, both with and without content, but including an attribute.

```
<foo name="value">bar</foo>
<foo name="value"/>
```

An attribute always has a name and a value. The name of this attribute is name and its value is value. The way to write an attribute is as shown here, the name, followed by an equal sign, then the value in quotes. Single quotes are also acceptable.

As noted, an element can have some content. In the case above the content was the word 'bar' and is called text content. An element's content can also be another element, as in the following example.

```
<foo> <baz></baz> </foo>
```

The containing element is '<foo></foo>' and the nested element is '<baz></baz>'.

Though it is not necessary, XML is usually written with tabs before nested elements to indicate iconically the level of nesting. The following is the syntactic equivalent of the previous example.

```
<foo>
   <baz></baz>
</foo>
```

An element may also have more than one nested elements inside it and those elements may or may not have other nested elements inside them. Here is an example.

```
<foo>
   <baz></baz>
   <sis>
     <boom></boom>
     <bah></bah>
   </sis>
</foo>
```

In the above example, both '<baz></baz>' and '<sis></sis>' are nested inside '<foo></foo>'. Then '<boom></boom>' and '<bah></bah>' are nested in '<sis></sis>'.

There is no limit to the depth of the nesting that can occur with XML. Keep in mind that all XML elements may, irrespective of their nested level, also have text content as well as attributes.

```
<foo name="value">
   <baz>baz content</baz>
   <sis name="value">
     <boom>boom content</boom>
     <bah/>
   </sis>
</foo>
```

In summary, XML syntax revolves around elements. An element is composed of a start and end tag and may or may not have attributes. Attributes come in a name-value pair. Elements may or may not have content. And, content may be text or other elements, which are called nested elements.

XML FILE ORGANIZATION

All 26 of the XML files that make up the corpus are identical in their organization. This section analyzes the sample XML found in appendix D and downloadable from the website by the name *sampleXML.xml*. This sample has all of the structures of the 26 files that make up the corpus but is much smaller, which makes it easier to tease out and explain individual aspects of the organization of the actual corpus files (understanding the organization of these files is key to being able to extract data from the corpus.) The rest of this section reviews the sample XML file line-by-line, introducing and explaining the form and function of the pieces of XML each line. Numberings placed below lines are only for reference and do not appear in actual XML files.

Here is the first line.

```
<?xml version="1.0" encoding="iso-8859-1"?>
                  1                       2
```

This line is called a declaration and appears at the beginning of all xml files. It specifies (1 the XML version and (2 the iso-8859-1 encoding, making Latin characters visible. It is not considered an XML element (see the XML tutorial in appendix C for an explanation of XML elements or other XML jargon.)

The next four lines make up the <file> start element.

```
<file
  xmlns="http://www.null.com"
  xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
  xsi:schemaLocation="http://www.null.com SLAschema.xsd">
```

The file start element is the root element, or the element that the rest of the xml document is nested in. It is not closed until the very last line of the file. It has three attributes. These attributes do not have anything to do with the actual data that came from the tests but are required for validating the XML file.

The next line is the first with information from the actual test, the name of the subject who was tested.

```
<subject name="S1"/>
                    1
```

It is an empty element because of the it has the (1 closing slash at the right side of the element. The value "S1" of the attribute name is used to identify of the subject who was tested.

The next line is the start of the <data> element and uses attributes to identify the source of the data.

```
<data source="oral test transcription" date="" id="Test1">
```

This element uses attributes to specify the source of the test, an oral test transcription, the date of the test (this information is available but has not been added to the corpus yet), and the test id. This was S1's first test. The rest of the file consists of data and is contained in this element and the data element is closed on the second-to-last line of the file, right before the <file> element is closed.

The next line of the sample file is the start of the <sections> element.

```
<sections>
```

Each XML file has one <sections> element which serves to group <section> elements and to separate their data from data which could be added later. It is closed as </sections> on the third-to-last line of the file, just before the <data> tag is closed.

Each time a subject was tested, eight picture-guided narration and five situational response elicitation devices were administered (see appendix A) and each of these responses was placed into a <section> element, so the complete XML files contains 13 <section> elements. Our sample XML file has only two <section> elements.

The next line is the start of the first section of the file.

```
<section type="narration" sectionid="1">
```

The <section> start element has two attributes. The *type* attribute indicates that this section came from a narration. The sections are either of the type 'narration' or 'situation', depending on which type of elicitation device they were responding to. The *sectionid* gives the order in which the elicitation device appeared in the actual test.

The next line in the file is the <transcription> element.

```
<transcription>en la avenida América él era casado y tenía cinco
hijos</transcription>
```

The <transcription> element contains the transcription of the speech the subject produced in response to the test instrument identified by the attributes of the current <section> element. It is closed on the same line as </transcription>.

The next line of the sample file is the start of the <units> element.

```
<units>
```

The <units> element is similar to the <sections> element in that it groups many <unit> elements and is closed as </units> after all of its component <unit> elements have been defined (opened and closed.)

The next three lines show a complete <unit> element.

```
<unit type="T-unit" unitid="1">
                  1           2
  <text size="4">en la avenida América</text>
              3                         4
</unit>
```

The text of the narration was broken down into T-units. I followed the general rule for identifying a T-unit (any verb that is not in a subordinate clause forms a T-unit.) The <unit> elements allow for expressing the complexity of the speech sample in terms of the T-unit.

All of the <unit> elements have a nested <text> element. <unit> elements in this corpus are of the *type* 'T-unit' (1). Each <unit> element has a *unitid* (2), which is the sequential number of the unit within the current <section> element. Each <text> element has a *size*, or the number of words which compose it (3). Each <text> element contains the text of the words spoken by the subject (4). Finally the <text> element is closed as </text> and the <unit> element is closed as </unit> after the <text> element is closed, or after any <error> elements (explained below) have been defined.

Here are the next 14 lines of the file; they have been numbered here for reference.

```
1     <unit type="T-unit" unitid="2">
2      <text size="3">él era casado</text>
3     </unit>
4     <unit type="T-unit" unitid="3">
5      <text size="4">y tenía cinco hijos</text>
6     </unit>
7    </units>
8   </section>
9   <section type="situation" sectionid="2">
10   <transcription>En los Estados muchos tienen carros ...
11   <units>
12    <unit type="T-unit" unitid="30">
13     <text size="6">En los Estados muchos tienen carros</text>
14    </unit>
```

The two remaining <unit> elements are defined in the first six of these lines, three lines per unit, as was the first <unit> element above. Lines 7-8 close the <units> and <sections> elements. Line nine has the start tag of the next <section> element. Its attribute *type* is ‘situation,’ meaning the contents of this section were derived from a situational response instrument rather than a picture-guided narration instrument. Line ten has been truncated and contains the <transcription> element. Line 11 has the start of the next <units> element. The last three lines define a unit element the same way done by the first three lines.

The next seven lines of the file show a <unit> element with a single embedded <error> element.

```
<unit type="T-unit" unitid="3">
  <text size="4">Es lejos al centro</text>
  <error category="319" errorid="1">
    <token>lejos </token>
    <repair>distante</repair>
  </error>
</unit>
```

Each speech errors made by the subject is stored in an <error> element, which are in turn embedded in the <unit> element which contains the text of the speech in which they occur. Each <error> element has an associated *category* attribute to indicate which category the error came from.. This error was classified as type ‘319,’ the catch-all category (for errors which don’t clearly fit into currently defined categories.) Each <error> element also has an *errorid* which is the sequential number of the error within the current <unit> element, as one <unit> element may have more than one embedded <error> element.

Every error also has two nested elements with text content in them. These are the <token> and <repair> elements. The <token> element contains the instance or token of incorrect use and the <repair> contains a correction, or native-like use which, if it had been used, would eradicate the error. The <repair> element is sometimes left empty (without text between the start tag, <repair>, and the end tag </repair>), indicating that native-like speech would have omitted the contents of the <token> element. Finally, each <error> element is closed as </error>, and after the final embedded <error> element has been closed, the following line closes the <unit> element with </unit>.

Here are the next 15 lines of the file; they are numbered here for reference.

```
 <unit type="T-unit" unitid="31">
  <text size="9">y las casas quedan a distante de cada uno</text>
  <error category="210" errorid="1">
   <token>a distante </token>
   <repair>separadas </repair>
  </error>
 <error category="313" errorid="2">
  <token>uno </token>
  <repair>una </repair>
 </error>
 <error category="212" errorid="3">
  <token>uno </token>
  <repair>otra</repair>
 </error>
</unit>
```

These fifteen lines show the last <unit> element in the file. This <unit> element has three embedded <error> elements in it. When multiple <error> elements are embedded in a single <unit> element, each will be fully defined (and closed as </error>) before the next one begins.

Finally, these are the last five lines of the file.

```
       </units>
      </section>
    </sections>
  </data>
</file>
```

The first line closes the most recent <units> element, the second line the most recent <section> element and the last three lines close the <sections>, <data> and <file> elements, which were opened at the beginning of the file.

XML SCHEMAS

Schemas are simply outlines of the way some data is organized. Once the schema is written, it is only used to make sure that all of the XML data is organized, or structured, in the exact same way. Appendix E contains the schema for the XML data of this corpus.

The term for using a schema to check the structure of XML is called validating. Schemas are used to validate an XML file. This simply means that the XML file is compared to the schema file to make sure that it is structured the way the schema says it should be.

An XML schema is saved in an XML Schema Declaration, or an XSD file, with a '.xsd' extension. To validate the XML, use what is called an XML validating parser. Start the application, point it to the XML file you with to validate and to the XSD file you wish to validate against and run the program. The application compares the XML to the XSD and tells you whether the XML is structured as specified by the XSD. If it isn't, the parsing program notifies of an error and provides information regarding what is wrong with the XML file. The detail of this information depends on the chosen parser and maybe its current settings.

Hundreds of XML validating parsers are available. If you would like to try validating an XML file, a simple way is to find one that is web-based, which eliminates the need to install one on your computer.

APPENDIX D: SAMPLE XML FILE

```
<?xml version="1.0" encoding="iso-8859-1"?>
<file
  xmlns="http://www.null.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.null.com SLAschema.xsd">
  <subject name="S1"/>
  <data source="oral test transcription" date="" id="Test1">
    <sections>
      <section type="narration" sectionid="1">
        <transcription>en la avenida América él era casado y tenía cinco
hijos</transcription>
        <units>
          <unit type="T-unit" unitid="1">
            <text size="4">en la avenida América</text>
          </unit>
          <unit type="T-unit" unitid="2">
            <text size="3">él era casado</text>
          </unit>
          <unit type="T-unit" unitid="3">
            <text size="4">y tenía cinco hijos</text>
          </unit>
       </units>
      </section>
      <section type="situation" sectionid="2">
        <transcription>Es lejos al centro En los Estados muchos tienen carros
y las casas quedan a distante de cada uno</transcription>
        <units>
          <unit type="T-unit" unitid="30">
            <text size="6">En los Estados muchos tienen carros</text>
          </unit>
          <unit type="T-unit" unitid="32">
            <text size="4">Es lejos al centro</text>
            <error category="319" errorid="1">
              <token>lejos </token>
              <repair>distante</repair>
            </error>
          </unit>
          <unit type="T-unit" unitid="31">
            <text size="9">y las casas quedan a distante de cada uno</text>
            <error category="210" errorid="1">
              <token>a distante </token>
              <repair>separadas </repair>
            </error>
            <error category="313" errorid="2">
              <token>uno </token>
              <repair>una </repair>
            </error>
            <error category="212" errorid="3">
              <token>uno </token>
              <repair>otra</repair>
            </error>
          </unit>
       </units>
      </section>
    </sections>
  </data>
</file>
```

APPENDIX E: CORPUS SCHEMA

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:target="http://www.null.com"
  targetNamespace="http://www.null.com"
  elementFormDefault="qualified"
  version="0.1.1">
  <element name="file">
    <complexType>
      <sequence>
        <element name="subject" minOccurs="1" maxOccurs="1">
        <complexType>
          <attribute name="name" type="string" use="required"/>
        </complexType>
      </element>
        <element name="data" minOccurs="1" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="sections" minOccurs="1" maxOccurs="1">
            <complexType>
              <sequence>
                <element name="section" minOccurs="1" maxOccurs="unbounded">
                  <complexType>
                  <sequence>
                    <element name="transcription" minOccurs="0"
maxOccurs="1">
                      <complexType mixed="true">
                        <attribute name="size" type="int" use="optional"/>
                      </complexType>
                    </element>
                    <element name="units" minOccurs="1" maxOccurs="1">
                      <complexType>
                        <sequence>
                        <element name="unit" minOccurs="1"
maxOccurs="unbounded">
                          <complexType>
                            <sequence>
                              <element name="text" minOccurs="1"
maxOccurs="1">
                              <complexType mixed="true">
                                <attribute name="size" type="int"
use="optional"/>
                              </complexType>
                              </element>
                              <element name="audio" minOccurs="0"/>
                              <element name="phonetic" minOccurs="0"/>
                              <element name="error" minOccurs="0"
maxOccurs="unbounded">
                              <complexType>
                                <sequence>
                                  <element name="token" minOccurs="1"
maxOccurs="1"/>
```

```
                                  <element name="repair" minOccurs="1"
maxOccurs="1"/>
                                </sequence>
                                <attribute name="category" type="string"
use="required"/>
                                <attribute name="errorid" type="int"
use="required"/>
                              </complexType>
                              </element>
                            </sequence>
                          <attribute name="type" type="string"
use="optional"/>
                          <attribute name="unitid" type="int"
use="required"/>
                        </complexType>
                        </element>
                      </sequence>
                    </complexType>
                  </element>
                  </sequence>
                  <attribute name="type" type="string" use="optional"/>
                  <attribute name="sectionid" type="int" use="required"/>
                </complexType>
              </element>
            </sequence>
            </complexType>
          </element>
        </sequence>
      <attribute name="source" type="string" use="optional"/>
      <attribute name="date" type="string" use="optional"/>
      <attribute name="id" type="string" use="optional"/>
      </complexType>
    </element>
  </sequence>
</complexType>
</element>
</schema>
```

APPENDIX F: WINDOWS REFERENCE

DEFINITIONS

command prompt

You can get to the command prompt by clicking [Start], then choosing [Run...] (square brackets are used in this section to designate keyboard and mouse inputs),usually at the bottom right of the available choices. Type 'cmd' in the field and then click [OK]. A black window should open up with a command prompt and a blinking flat cursor. Just type in the command and hit the [Enter] or [Return] key on your keyboard.

If you don't have the [Run...] button on the Start menu, search for it in the C drive, in the 'WINDOWS' folder, in the 'system32' folder. The program is called 'cmd.exe'.

text editor

A text editor is something you can type and edit text with and save without any formatting. Microsoft Word can do this because it can save files with a '.txt' extension; however, you don't want to use Word or anything similar to write and edit Perl scripts. It is just way too big and bulky.

Instead use an editor like notepad or even WordPad. The one thing to check with notepad is that when you save your file you also select 'All Files' from the 'Save as type' field, immediately below the 'File name' field. This will allow you to type the '.pl' file extension and run the script with Perl. If you don't select this option the file will save with the file extension of '.txt', even if you do give it a name with a '.pl' ending, the '.txt' will be appended and Perl will not recognize it as a Perl script.

If you would like to download a free and easy-to-use editor designed for writing small scripts like this, try *notepad2*, search for it on Google (www.google.com.) It is similar to the 'notepad.exe', which comes standard with Windows, but it has some good features for writing scripts like line numbering and syntax highlighting.

path name

Every file has a name, an easy concept. Well, the 'path name' is simply the file name with the location of the file, or the path, in front of it. The following is an example of a path name for a file.

```
h:\project\data\xml\perl\hello.pl
```

There file is named 'hello.pl' and the path to the file starts in the 'H' drive of the computer, then the 'project' folder, the 'data' folder the 'xml' folder, and finally the 'perl' folder. If you were to open Windows Explorer or My Computer, select the 'H' drive and open the folders in that succession, you would find the file 'hello.pl' in the 'perl' folder.

If the concept still needs some affirmation, use Windows Explorer to open up the last file you remember saving on your computer. Once you get to the file look at the 'Address' bar above the list of contents. You should see the path to the folder in which your file is saved. The path is sometimes called the 'address'.

command history

When you go to run the same Perl scripts as before, there is a slick way to avoid having to type in the whole command by hand. Hold down the [Shift] key and press the up and down arrow keys to cycle through past commands. Up will take you to the most recent command used, down to the oldest one used. When you reach the command you want to run, just hit [Enter].

relative paths

When you type in a path name that starts with a drive letter, like 'h:\project\data\xml\perl\hello.pl' (the H is the drive letter), it is called an absolute path. You can also type in a path name relative to your current location at the command prompt.

Your current location at the command prompt is indicated by the path before the '>' sign. Open your command prompt and take a look at the default location. You can change the current location with the cd command.

In order to run a Perl script with a relative path name you have to know where the script is relative to your current location at the command prompt. If your current location is 'h:\project\data' and you want to run the script called 'hello.pl', which is located in 'h:\project\data\xml\perl', then this is what you enter at the command prompt.

```
perl xml\perl\hello.pl
```

If the current location were instead 'h:\project\data\xml\perl\reference' and you wanted to run that same 'hello.pl' script, here is what you would enter at the command prompt.

```
perl ..\hello.pl
```

One last example, if the current location were 'h:\project\data\xml\perl' and you wanted to run the 'hello.pl' script, you would enter the following at the command prompt.

```
perl hello.pl
```

COMMANDS AND OPERATORS

There are several handy commands to use at the command prompt. The are listed below with a description of each. Most, like 'cd' and '..' can be combined together.

You can get help on a commands at the command prompt by typing 'help' then the name of the command. For example, to get help with the 'cd' command you would type 'help cd', then hit [Enter].

cd

Named ‘change directory'. Changes to the specified directory. Type the name of the new directory after the command, like this 'cd data' or 'cd data/xml', then hit [Enter]. The current directory will change to the one specified in the command.

dir

Named 'directory'. Prints the contents of the current directory. Type 'dir' and hit [Enter]. The directory contents will print to the screen.

..

Named 'directory up'. Changes to the directory above the current directory. When your current location is 'h:\program\data', type 'cd ..' and hit [Enter]. Your new current location will be 'h:\program'.

>

Named 'write to file'. Writes the printed content to the specified file instead of to the screen. Run your 'hello.pl' script and write the contents to a file with this command. Type 'perl hello.pl > hello.txt' and hit [Enter]. Whatever gets printed by the hello.pl script is now in a file called hello.txt. This new file will be located in whatever the current directory was when the 'perl hello.pl > hello.txt' command was given. Warning: If there is already a file called 'hello.txt' in that directory it will be destroyed and replaced by the new file.

>>

Named 'append to file'. Just like the '>' operator, except that it appends the printed content to the end specified file, rather than creating a new file. Type 'perl hello.pl >> hello.txt' and hit [Enter]. Whatever gets printed by the hello.pl script is now appended to the end of the file called hello.txt. If the file 'hello.txt' does not exist before the command was given, it will be created.

APPENDIX G: PERL TUTORIAL

OVERVIEW

Now is a good time for a word of advice - if the thought of using a programming language is intimidating to you, DON'T LET IT BE. The degree of skill required to use Perl for these projects is minimal; it takes about as much effort as learning to say a few phrases in a foreign language. The returns, however, are very rich, like making a friend because you initiated a conversation in their native tongue.

Another thing, this tutorial is written for Windows users and it's been stripped of all but the essentials. If you are a Linux or Unix user (there is a good chance you don't need this tutorial), or for any other reason it is inadequate, a Google search on 'perl tutorial' should give you more results than you can shake a stick at.

Finally, this is a tutorial, meaning that as information is presented incrementally, each new section builds on the knowledge gained from the previous. However, it could also function as a reference, since it has been organized with intuitive headings, etc.

With that, let the fun begin!

GETTING PERL

The first thing to know is that Perl is a program, and if your computer doesn't have it, it needs to be downloaded and installed. Though Perl is free and many other systems are shipped with it already installed, Windows usually does not come with Perl already installed.

But, before you download you can check to see whether it has been installed or not. The way to do this is to open the command prompt and type 'perl --help'. If you do not have Perl installed, a message will print to the screen that says, "'perl' is not recognized as an internal or external command, operable program or batch file." If your machine does have perl installed, a list of stuff will print out, the last, or one of the last items of which should be '-X disable all warnings'.

If you need to download Perl, you can do so at Active State, from their website (www.activestate.com.) Download and install it according to instructions.

Once it has installed, you may or may not have to restart your computer before it will work (if you are running Windows it's a good idea to restart your computer at random times anyway.) Go ahead and restart it before checking to see that it installed successfully. Perform the same check described above to see whether it has installed. If you got that list of stuff, congratulations! The first step is complete.

USING PERL

THE BASICS

The Perl program on your computer does one thing an one thing only, it runs scripts. A Perl script is saved in a file with the file extension of '.pl', so a script named 'hello' would be saved in the file 'hello.pl'.

When Perl runs a script it looks at the file, line by line, starting at the top and performs the commands on each line until it reaches the end of the file, then it stops.

SCRIPTS

The point of writing a script is to have it perform repetitive functions, which would take a human a lot longer. There are quite a few commands that are part of the Perl language, you will only have to learn a few. The first one will be the 'print' command. The following line is an example of an entire Perl script. Yes, just one simple line.

```
print "hello";
```

When Perl reads this line it recognizes the command 'print' and then reads what it is supposed to print, the word 'hello'. Each line of a Perl script ends with the semicolon. Now it's time for you to write your own first Perl script. Open Textpad or some other simple text editor, WordPad will also work, and type in the line. Save it as 'hello.pl' (make sure the .txt is not appended to the file name as Perl will not run a file named ‘hello.pl.txt.’)

Now it's time to run your script.

RUNNING A SCRIPT

Running a script means you start Perl and point it to a script you want it to run. This is done from the command prompt. When you checked whether Perl was installed on your computer you actually ran Perl without pointing it to any script. So you've already won half this battle.

Once the command prompt is open, you type in the command to run the script. Here is what I typed.

```
perl h:\project\data\xml\perl\hello.pl
```

This command has two parts, 'perl', which is a command to the computer to start the Perl program, and 'h:\project\data\xml\perl\hello.pl', which points Perl to the script you want it to run. Notice that you have to be exact about where the script is located. If you are unfamiliar with the syntax of the second part of the command, the Windows reference in appendix F offers a tutorial on pathnames.

Type in the command, specifying where your file is located, then hit [Enter]. The word 'hello' should print to the screen on one line. If that is what happened, congratulations again. If not, go back to the last part of the tutorial that you successfully completed and start from there.

There are some useful things to know about the Perl print command. To make future reference easier, they have been placed in a section below.

VARIABLES

Variables store information and are used to recall that information elsewhere. To make a variable in a Perl script, use the '$' sign. The following line shows a variable being assigned a value.

```
$name = "Abraham Lincoln";
```

The variable could, just as well, have been '$foo' or '$bar', as long as it had the '$' sign in front of it. Below is an example of how a variable is created and used in Perl.

```
$name = "Abraham Lincoln";
print "Hello,\n";
print "My name is $name.\n";
print "I was named after $name.\n";
```

Running this script gives the following output.

```
Hello,
My name is Abraham Lincoln.
I was named after Abraham Lincoln.
```

The value of a variable can, well, vary. Take a look at the next example script.

```
$day = 4;
print "Yesterday was June $day, 2007.\n";
$day = 5;
print "Today is June $day, 2007.\n";
```

This script gives the following output.

```
Yesterday was June 4, 2007.
Today is June 5, 2007.
```

In sum, variables store information, which can be retrieved and used later.

COMMANDS AND OPERATORS

We have already seen one Perl command, 'print'. The next one to learn is the ‘split’ command. This command will chop up a sequence of characters and store the results in variables. Take a look at the following script. The numbers below the first line are for reference and are not part of the actual script.

```
($first,$second) = split /:/, "number one:number two";
               1   2     3  4  5                       6
print "$first\n";
print "$second\n";
```

Running this script gives the following output.

```
number one
number two
```

Broken into five parts, 1) the variables where you want the results stored, placed in parentheses, 2) an equals sign, 3) the ‘split’ command, 4) the value to “split on” inside forward slashes, in this case a colon, 5) a comma, 6) the item to be split.

There are a few more peculiarities about the split command. Look closely at the following script and its output to figure them out. The ‘print’ command interprets the ‘\n’ as a ‘newline’ character and starts a new line in the printed output wherever ‘\n’ is found.

script

```
$string = "one two three four";
($first,$second,$third) = split / /, $string;
print "first: $first\nsecond: $second\nthird: $third\n";
print "**********************\n";
print "string: $string\n";
```

output

```
first: one
second: two
third: three
**********************
string: one two three four
```

If the item can be split into more values than the number of variables specified to hold them, the rest of the values are not included. However, the split command does not change the value of the item it splits. It just extracts info from that item and stores it.

The next command is the 'if' statement. It has three parts, 1) the 'if' command, 2) a set of parentheses to hold the conditional test and 3) a set of curly braces to hold the commands to run, when the conditional test is true. (The numbers below the first line in this script are for reference and not part of the actual script.)

script

```
if ( 1 ) { print "hello\n"; }
 1    2     3
if ( 0 ) { print "goodbye\n"; }
```

output

```
hello
```

In Perl a '1' means true and a '0' means false. Since only the first conditional test was true, only the first print command was executed. Notice that, though command statements inside the braces end with a semicolon, no semicolon is used after the if statements.

Usually, though, you will want to compare two things in the conditional. To compare numbers use either a double '==' sign ‘is equal to,’ the greater than sign, '>', and less than sign, '<'. Here is another sample script and its output.

script

```
if ( 100 == 100 ) { print "A\n"; }
if ( 100 < 100 ) { print "B\n"; }
if ( 100 > 0 ) { print "C\n"; }
```

output

```
A
C
```

The conditional test can also be applied to variables as shown here.

script

```
$a = 15;
$b = 16;
if ( $a == $b ) { print "$a equals $b\n"; }
if ( $a < $b ) { print "$a is less than $b\n"; }
if ( $a > $b ) { print "$a is greater than $b\n"; }
```

output

```
15 is less than 16
```

Finally, if the commands to be executed are long, white space is used to keep things readable.

script

```
if ( 100 > 99 ) {
  print "THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM\n";
  print "THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM\n";
  print "THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM\n";
  print "THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM\n";
  print "THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM\n";
  print "THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM\n";
}
```

output

```
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM
```

The last command to look at for now is the 'm' or match command. This command checks a string of characters to see whether a smaller string of characters is found inside it. The match command evaluates to true or false depending on whether or not it contains the smaller string. For example, you could use the match command to ask whether your first name appeared in a line of text, if your first name was anywhere in the line of text, the command would evaluate to true.

The match command has four parts, first the item you want to check, then an equals-tilde, then the 'm' command, last, the item to match inside forward slashes. Here is an example of a script using the match command and its output.

script

```
$line = "This line has the name Roger in it";
if ( $line =~ m /Roger/ ) {
  print "The following line has the name Roger in it:\n";
  print "    $line\n";
}
```

output

```
The following line has the name Roger in it:
This line has the name Roger in it
```

Just as the searched element may be a string or a variable, variables may also be used inside the forward slashes as the item to be searched for.

ARRAYS

In this section you will learn about an extremely useful, yet simple Perl structure called an array. An array is merely a list of things, usually variables. Each item in the list is numbered, starting with zero. This number is called the index of the item.

Here is an example of how to make an array of names.

```
@names = ("john","michael","peter");
```

This assignment puts the three names into the array called @names. The array could just as well have been named @foo, so long as it had the '@' in front of it. The first name, 'john', has the index '0', the second name, 'michael', has the index '1' and the third name, 'peter', has the index '2'. The items can be retrieved from the array by using their index as shown in the following example.

```
$name = @names[0];
```

This assignment gives the variable '$name' the value of 'john', since 'john' is has the index of '0'. Take a look at the following script and its output.

script

```
@names = ("john","michael","peter");
print @names[1];
print "\n";
$size = @list;
print $size;
```

output

```
michael
3
```

Two significant things. First, the item indexed '1' was printed without assigning it to a variable. The number of items in the array extracted when the array is made the value of the variable $size. The size, 3, is then printed out.

There are four other Perl commands that are very useful for arrays. They are 'shift', 'foreach', 'join' and 'while'. We will take a look at each in turn.

The 'shift' command extracts from the array the item with the index of '0', then lowers the index of each item by one. Take a look at the following script and its output.

script

```
@names = ("john","michael","peter");
print "@names[0]\n";
shift @names;
print "@names[0]\n";
```

output

```
john
michael
```

Next, the 'foreach' command. This command takes each item, one by one, from the array and assigns it to a variable and lets you do something with the variable before moving on to the next item in the array. The syntax of the 'foreach' command is in this order, first the 'foreach' command, then the variable to assign to, then the array inside parentheses, last, commands to execute inside curly braces. Individual commands inside the curly braces are terminated with a semicolon. Here is an example of the 'foreach' command in action.

script

```
@names = ("john","michael","peter");
foreach $name (@names) { print "$name\n"; }
```

output

```
john
michael
peter
```

The 'join' command is similar to the 'foreach' command in that it goes through an array, one item at a time. The difference is that the join command joins all of the items in the array by a specified string of characters and makes out of it one large string. Here is an example of how the 'join' command works.

script

```
@names = ("john","michael","peter");
$string = join " > ", @names;
print "$string\n";
```

output

```
john > michael > peter
```

In the example, each item in the array was joined to the next item this sequence of characters, 'space, greater than, space'. The resulting string of characters was stored in the variable '$string' and then printed out.

The last item to consider is the 'while' statement. The 'while' statement is very similar to the 'if' statement in its syntax. First is the 'while' command, then parentheses with a conditional test, then a set of curly braces, which contain the commands to be executed, over and over, as long as the while statement remains true. Take a look at the following script to see what is going on. There is a flaw to this script. See if you can identify it.

```
while ( 1 ) {
  print "forever\n";
}
```

The flaw in the script is that the conditional is never false, so it can't ever stop. If you ran this script it would continue printing 'forever' to your screen until your computer's temporary memory was filled up. Then Perl would stop running the script. This wouldn't harm the computer; temporary memory is always getting filled up and emptied as you open and close and use different programs. It could be annoying though.

Whenever you use the 'while' command, you have to make sure that each time it executes the commands in the curly braces that it gets closer to the condition of being false, i.e. a stopping point. Consider the following example script and its output.

script

```
$i = 10;
while ( $i > 0 ) {
  print "$i ";
  $i = $i - 1;
}
```

output

```
10 9 8 7 6 5 4 3 2 1
```

In this example the variable $i starts with a value of 10. When the while command initiates, it first tests the conditional in the parentheses, since $i equals ten and ten is greater than zero, it executes the commands in the braces, first printing the value of $i, with a space after it, and then decrementing the value of $i by one. After that it goes back to the test the conditional again.

If you want to cycle through the contents of an array, just stick the array in the parentheses in place of the conditional to be tested. As long as the array has at least one item in it, the conditional test will be true. Take a look at how to use this feature along with the 'shift' feature.

script

```
@names = ( "Jim", "Joe", "Bob", "Bill");
while ( @names ) {
  $name = shift @names;
  print "$name\n";
}
```

output

```
Jim
Joe
Bob
Bill
```

Since the array gets smaller each iteration, and the conditional is testing whether the array has items in it, this while command will eventually stop.

There are a few more Perl commands which are useful but not essential. These are included in the Perl reference section below.

PASSING INFO TO PERL SCRIPTS (ARGUMENTS AND PARAMETERS)

Sometimes you want to get at information that is not inside the Perl script. Perl makes this very easy. For starters, lets say you want to set a variable in a script to a different value each time you run the script. This takes two steps. 1.) Set up the script to catch an *argument*, also called a *parameter*. 2.) Pass the argument to the script from the command prompt. For number 1. the script looks like this.

```
$value = @ARGV[0];
print "$value\n";
```

The array called '@ARGV' is standard in Perl. It captures the arguments that are passed into the Perl script when the script is run from the command line. The second step is to run the script and pass it an argument. If the script above were saved in a file named 'args.pl' you could call it and pass in an argument like this.

```
perl args.pl hello
```

This would print the word 'hello' to the screen. If the argument you want to pass to the script has spaces in it, just use quotation marks. Here is another example of running the same script with an argument passed to it and the resulting output.

run

```
perl args.pl "hello, joe"
```

output

```
hello, joe
```

More than one argument can be passed into the script at one time. Here is an example of the script, running it and the output.

script

```
$value1 = $ARGV[0];
$value2 = $ARGV[1];
print "$value1\n";
print "$value2\n";
```

run

```
perl args.pl Frank, "Hello. This is Joe."
```

output

```
Frank,
Hello. This is Joe.
```

READING AND WRITING FILES WITH PERL (I/O)

Now, lets say you want to get the contents of a file and use it inside a script. This is done quite simply in Perl. It takes two steps. 1.) Open the file. 2.) read the contents into an array. The next script does just that.

```
open(FILE,"h:/project/data/xml/file.txt");
@lines = <FILE>;
```

In the above script the file 'file.txt', located in the directory 'h:\project\data\xml' is opened with a filehandle called 'FILE'. A filehandle just kind of handles the file, does stuff to it. The filehandle does not have to be in all caps, and it can be named whatever you want to name it. Once the file has been opened into the filehandle, it gets read into the array named '@lines' by assigning the value '<FILE>' to the array. After the assignment is made, the lines from the file are stored in the array named '@lines', each line being an item in the array.

This is a good time to test these new ideas. Make a new file called 'file.txt'. Type a few lines into it. Here is an example of what your 'file.txt' file might look like.

```
this is line one
this is line two
this is line three
this is line four
```

Now write a script like the one below. The one thing you need to change is the location of the your file on your computer. Also notice that Perl uses forward slashes between directories, Windows uses backslashes.

```
open(FILE,"h:/project/data/xml/file.txt");
@lines = ;
foreach $line ( @lines ) {
  print $line;
}
```

Its time to run the script. Just run it from the command line and the contents of the 'file.txt' file should print to the screen. If it doesn't do what you expect, go back to the last part of the tutorial that you successfully completed and start over. If it does work, congratulations. You are speaking Perl.

Now lets combine the use of arguments and these file commands. The idea here is to make the argument you pass to the Perl script be the name of the file you are going to work with. Here is a text file, script, entry for the command line and output. Lets say the text file is saved as 'file.txt' and the script is saved in a file called 'printlines.pl' and both are located in the current directory.

text file

```
line one
line two
line three
```

script

```
$file = $ARGV[0];
open (FH,$file);
@lines = <FH>;
        foreach $line ( @lines ) {
            print "$line";
}
```

run

```
perl printlines.pl file.txt
```

output

```
line one
line two
line three
```

One thing to remember about passing in the name of a file to a Perl script is that if the path name specifying the file location is relative, it has to be relative to the current directory and not to the location of the script. For example, if the script is saved in 'c:\perl\scripts' and the file you are passing in as an argument is named 'file.txt' and is saved in 'c:\perl\textfiles' and the current directory is 'c:\perl', then the path name of the file that is sent as an argument to the script is textfiles/file.txt and not ../textfiles/file.txt.

Now is a good time to practice writing scripts that will open files and do stuff with the data in those files. Below is an example of a script that opens the file specified in the first argument to the script and prints all of the lines in that file that have the word 'the' in them.

```
$file = $ARGV[0];
open (FH,$file);
@lines = <FH>;
foreach $line ( @lines ) {
  if ( $line =~ m/the/ ) {
    print "$line";
  }
}
```

Here is a script that prints all of the words in a file, one to a line.

```
$file = $ARGV[0];
open (FH,$file);
@lines = ;
while (@lines) {
  $line = shift @lines;
  foreach $word ( split / /, $line ) {
    print "$word>";
  }
}
```

This is the end of the basic Perl stuff. If any of it is hazy, go back and do it again. A couple of times through the examples and it should start to be pretty easy.

PROCESSING XML WITH PERL

Perl makes parsing and processing XML very easy. You need to specify at the beginning of each script that you will be processing XML. You do that with the following line.

```
use XML::Simple;
```

Just put that line in at the top of the script. This allows Perl to use an internal library called ‘Simple XML’, which stores pre-written scripts for processing XML and allows you to use those scripts without writing them yourself.

After this line, there are two steps to start processing an XML file. 1.) Create an XML parser. 2.) Read an XML file in with the parser. Here is how that looks in a script.

```
$parser = new XML::Simple;
$document = $parser -> XMLin( $file );
```

The first line creates a new parser called '$parser'. The second line uses the new parser to read in an XML file stored in the variable '$file', the parsed xml is stored in a variable called '$document'. The variable names '$parser', '$file' and '$document' could have been anything else; these are just easier to remember.

Once the XML file has been stored in the $document variable by the parser, you can start to extract data from the $document variable. If you want to practice using the same XML file I’ve used here, it is shown in appendix D. and available for download from the website by the name *sampleXML.xml* (use the mouse to right-click on the link, select "Save Link As...", save it into the directory you use as your current command line directory.) This XML file has same structure as the full ones in the corpus, but it only has two sections, instead of 13, and the number of units in each section have been reduced. Take minute to review it and get familiar with its structure. This will make the following explanations clearer.

The manner of getting at the data that is stored in the '$document' variable is straightforward. The following line looks into the $document variable to find the element named 'subject', then looks inside the subject element to find the attribute called 'name' and stores the value of that attribute in the variable called '$name'.

```
$name = $document->{subject}->{name};
```

You can think of the arrow as a command that says “look inside the first thing for the second thing.” So, `'{A}->{B}'` would mean “look inside A for B”. In these commands XML elements are designated by curly braces The example script below shows how to combine what we've learned about Perl XML, Perl arguments (parameters) and the perl ‘print’ command. The script can be downloaded from the website by the name *perlscript-c.pl.*

Under ‘run’ is the command you need to type at the command prompt and has three parts, 1) the command to start Perl, 2) the script to run, 3) the first parameter, the name of the file we will use in the Perl script. (Notice that under 2) and 3) relative path names are used, so the script and the XML file must be in the current directory. See the Windows reference in appendix F.)

run

```
perl perlscript-c.pl sampleXML.xml
```

script

```
$file = $ARGV[0];
use XML::Simple;
$parser = new XML::Simple;
$document = $parser -> XMLin( $file );
$name = $document->{subject}->{name};
print "$name\n";
```

output

```
S1
```

In the sample file we are using, within the <subject> element there is an attribute called *name* with the value of *S1*. This script extracts that value and stores it in the variable '$name,' then prints it.

Write the same script and name it 'perlscript-c.pl'. Then run the command. If your output is the same, congratulations, you just parsed your first XML file with Perl.

The next thing to learn is how to deal with arrays in Perl XML. The script below is an augmentation of the previous and is available by download from the website by the name *perlscript-d.pl.* Run this script as you did the last one.

script

```
$file = $ARGV[0];
use XML::Simple;
$parser = new XML::Simple;
$document = $parser -> XMLin( $file );
$name = $document->{subject}->{name};
print "$name\n";
$array = $document->{data}->{sections}->{section};
print "$array\n";
```

output

```
S1
ARRAY(0x1add2e4)
```

The characters in the parentheses after 'ARRAY' may be different for your output. The point is that when you did this command, '$document->{data}->{sections}->{section}', you told Perl to "look into $document for an element called 'data', into 'data' for an element called 'sections', then into 'sections' for an element called 'section'." Since the element 'sections' contains two elements called 'section', an array containing both of those sections was returned. So, if we want to get at the data inside the 'section' elements, we have to turn them into Perl arrays.

There are two steps to doing this. Put the sections into an array that Perl can deal with, then use the 'foreach' command to deal with the array elements one at a time. Here are the two steps.

```
@array = @{$document->{data}->{sections}->{section}};
foreach $section (@array) {
  print "$section->{transcription}\n";
}
```

The command '@array = @{$document->{data}->{sections}->{section}}; ' tells Perl to take the result of '$document->{data}->{sections}->{section}' and force it into an array. Then you name that array '@array' so you can use it on the next line. When this stuffing into an array occurs you use the '@' symbol and then enclose the 'stuffing' in curly braces, as shown. Two lines could actually have been combined into one line, as shown here.

two lines

```
@array = @{$document->{data}->{sections}->{section}};
foreach $section (@array) {
  print $section->{transcription}, "\n";
}
```

one line

```
foreach $section ( @{$document->{data}->{sections}->{section}} ) {
  print $section->{transcription}, "\n";
}
```

The difference is that the array resulting from the command '@{$document->{data}->{sections}->{section}} ' is never given a name. Combining the two lines also makes the script less readable.

The next example is a script that uses two 'foreach' commands to print the content from each text element in the file.

```
@array1 = @{$document->{data}->{sections}->{section}};
foreach $section ( @array1 ) {
  @array2 = @{$section->{units}->{unit}};
  foreach $unit ( @array2 ) {
    print "$unit->{text}->{content}\n";
  }
}
```

The first line puts all of the sections into an array called '@array1'. The second line starts the 'foreach' command which, one by one, puts each section the variable '$section'. Each time one the '$section' variable is given a new section, two things occur. First, all of the units in that section are put into a new array called '@array2'. Second, another 'foreach' command is started. The second 'foreach' takes each 'unit' in '@array2' and prints the context from each of its 'text' variables. The second 'foreach' command continues until all of the units in '@array2' have been processed, then it goes back to the first 'foreach' command and, if there is another 'section' in '@array1', the process is repeated.

To test these new concepts. Alter your script so that it prints out the value of the 'unitid' attribute for each of the units that gets processed. The script below shows how to do this. Try it before looking at how it's done.

```
$file = $ARGV[0];
use XML::Simple;
```

```
        $parser = new XML::Simple;
        $document = $parser -> XMLin( $file );
        @array1 = @{$document->{data}->{sections}->{section}};
        foreach $section ( @array1 ) {
            @array2 = @{$section->{units}->{unit}};
                foreach $unit ( @array2 ) {
                  print "$unit->{unitid}\n";
              }
        }
```

In this last example we will alter the script to print out the information for each error in the file. Here is the script that does the job. It is available for download from the website under the name *perlscript-e.pl*.

```
$file = $ARGV[0];
use XML::Simple;
        $parser = new XML::Simple;
        $document = $parser -> XMLin( $file );
@array1 = @{$document->{data}->{sections}->{section}};
  foreach $section ( @array1 ) {
    @array2 = @{$section->{units}->{unit}};
      foreach $unit ( @array2 ) {
        if ( ref( $unit->{error} ) eq "HASH" ) {
          print "context: $unit->{text}->{content}\n";
          print "  category: $unit->{error}->{category}\n";
            print "  token: $unit->{error}->{token}\n";
            print "  repair: $unit->{error}->{repair}\n";
         }
         elsif (ref( $unit->{error} ) eq "ARRAY" ) {
            @array3 = @{$unit->{error}};
            print "context: $unit->{text}->{content}\n";
            foreach $error ( @array3 ) {
              print "  category: $error->{category}\n";
              print "  token: $error->{token}\n";
              print "  repair: $error->{repair}\n";
            }
        }
      }
    }
```

This is a larger script, and actually has some new complexity to it. First, it has three nested 'foreach' commands. Up to now we have only dealt with two.

Second, it has an 'if' and an 'elsif' command. The 'elseif' is like saying "otherwise if". It works just like the 'if' statement, only it must come right after an 'if' statement, and its conditional test is only evaluated if the first 'if' statement is false.

The last bit of complexity concerns that which is being tested inside the 'if' and 'elsif' conditionals. Remember when we first saw that when the command $document->{data}->{sections}->{section} is given it indicates an array of items, since there are more than one 'section' elements inside the 'sections' element. Since some 'unit' elements with errors have only one error and others have more, when the $unit->{error} command is given it may or may not be looking at an array of 'error' elements. If there is only one error then it is looking at a 'HASH' and if there are more than one errors it is looking at an 'ARRAY'. We don't need to know anything else about hashes.

Look back at the conditional tests for the 'if' and 'elsif'. In each one a command called 'ref' is given. This command looks at something and tells whether it is an 'array' or a 'hash'. After the 'ref' command comes the 'eq' which means "equals" and is just like using the double equals, '==', except the double equals is for comparing two numbers and the 'eq' is for comparing two strings of characters. So the whole conditional tests whether '$unit->{error}' is looking at an array or a hash.

In the case that it is a hash, meaning there is only one error, the information we want is extracted from the error with print commands. In the case that it is an array the context of the unit is printed out then another 'foreach' command is issued to process each of the errors in the array individually.

PERL COMMENTS

Everything you need to know to process XML with Perl has now been covered. There is one more thing, though, that turns out to be pretty important.

When writing your scripts, sometimes you will want to give an explanation of what parts of the script are supposed to be doing. This documentation makes it easier for other people to borrow from or use your scripts for their own purposes, and helps yourself remember what is going on in the script when you have not read it for some time.

This is done with 'comments'. Anything on a line in a Perl script that has a pound sign, '#', in front of it is ignored by the Perl program when the script is run. If you tried to run the script below nothing would happen.

```
# print "hello\n";
```

Sometimes you may want to use the 'comment' function to prevent certain lines of your script from running. This can be handy when you want to test only certain lines of your script and you don't want to delete a bunch of lines and then type them in again later.

It is a good idea to develop a habit of using comments to document your work. You and others will end up appreciating them.

This concludes the Perl tutorial. You should be able to understand and use all of the scripts that are part of the example project posted on the website, titled "Substituting for Two Aspectual Morphemes", as well as begin to write new ones for your own projects.

PERL REFERENCE

PERL OPERATORS

=

Assigns a value to a variable.

```
$name = "Joe";
```

This line makes the variable '$name' have the value of "Joe".

==

Tests two number to see if they are equal.

```
if ( $a == $b ) { print "equal\n";
```

This line will print the word "equal" if the values for $a and $b are equal.

>

Tests whether one number is greater than another.

```
if ( $a > $b ) { print "greater\n";
```

This line will print the word "greater" if the value of $a is more than the value of $b.

<

Tests whether one number is less than another.

```
if ( $a < $b ) { print "less\n";
```

Will print the word "less" if the value of $a is less than the value of $b.

eq

Tests whether one string of characters is the same as another.

```
if ( $a eq $b ) { print "same\n";
```

Will print the word "same" if the value of $a is exactly the same as the value of $b.

PERL COMMANDS

print

Lets say you wanted Perl to print something onto multiple lines, like in the following.

```
Hello,
My name is mud.
```

```
Goodbye.
```

The following piece of Perl script would do it for you.

```
print "Hello,\nMy name is mud.\nGoodbye.";
```

The special character '\n' means 'newline'. There are two other ways of getting the same result. Here is the first.

```
print "Hello
My name is mud.
Goodbye.";
```

As mentioned in another section. The Perl line ends with the semicolon. Here is the second way of getting that same printout.

```
print "Hello\n";
print "My name is mud.\n";
print "Goodbye.\n";
```

Now lets say you wanted to print with tabs as well, like in this example.

```
Day         Monday 4
Month       June
Year        2007
```

The following piece of Perl script would do it.

```
print "Day\tMonday 4\n";
print "Month\tJune\n";
print "Year\t2007\n";
```

The list of special characters for formatting your print commands includes the following:

\n for printing a new line.

\t for printing a tab.

Try altering your Perl script that you saved as 'hello.pl'. Include these special characters for practice. To save some keystrokes in running the same script again and again, click here.

split

```
($first,$second) = split /:/, "one:two";
```

This line puts the value 'one' in the variable '$first' and the value 'two' in the variable $second.

```
@items = split / /, "one two three four";
```

This line puts the value 'one', 'two', 'three' and 'four' into an array named @lines.

if

```
if ( 100 > 99 ) { print "this is true\n"; }
```

This line tests the conditional inside the parentheses, since true, it prints out 'this is true'.

@

```
@names = ("Jim","George","Mary","John");
```

This line creates an array called '@names' with four items.

```
print @names[3];
```

This line prints the item with the index of '3', or the fourth item from the array called '@names'.

```
$size = @names;
```

This line stores the size of the array called '@names' in the variable '$size'.

shift

```
$name = shift @names;
```

This line puts the zero-index value from the array named '@names' into the variable called '$name'. The rest of the elements are shifted to have an index value one less than before.

foreach

```
foreach $name ( @names ) { print "$name\n"; }
```

This line prints each item from the array called '@names' on a separate line.

join

```
print join "-", @names;
```

Prints one long string made of the individual items from the array called '@names', each item being separated by a '-' character.

while

```
$i = 10;
while ( $i > 0 ) {
  print "$i ";
}
```

This script prints the value of '$i' ten times, each separated by a space.

chomp

```
chomp ( $line );
```

This line removes any '\n', newline, characters from the end of the variable.

open

```
open(FH,$file);
```

This line opens the file named in the variable $file and establishes a handler for that file named FH.

<filehandle>

```
@lines = <FH>
```

This line puts the contents of the file handled by the handler 'FH' into the array called '@lines'. Each line in that file will be stored as an indexed item in the array.

```
->
```

```
$elemntA->{elementB}
```

This line looks in '$elementA' for 'elementB'.

ref

```
ref( $something )
```

This line tells whether '$something' is an array or a hash.

APPENDIX H: ERROR PRINTOUTS

```
Counts for category 315:
    Test1:  60
    Test4:  67
S1: Test1
1     hacía :hizo
S1: Test4
1     necesitó :necesitaba
S2: Test1
S2: Test4
1     se levantó :se había levantado
2     fue :había ido
3     había puesto :puso
4     visita :visitar
S3: Test1
1     he vivido :viví
S3: Test4
1     estaba :estuvo
2     habría :habrá
3     decía :dijo
4     pidió :pedir
S4: Test1
1     gustó :gustaba
2     preparó :preparaba
3     estaba leyendo :estuvo leyendo
S4: Test4
1     fue :era
2     estaba limpiando :había limpiado la casa
3     haciendo :había hecho
4     poniendo :puesto
5     estaba :estuvo
6     tenía :tuvo
S5: Test1
1     dormía :durmió
2     averiguar :averiguo
3     probaba :probó
4     tomaba :tomó
5     olvidó :habia olvidado
6     continuar :continuo
S5: Test4
1     dormir :durmió
2     limpió :había limpiado
3     hizo :había hecho
4     preparó :había preparado
5     leyó :había leido
6     estaba casado :me casé
S6: Test1
1     dormía :durmió
2     hubo :había
3     hubo :había
4     hubo :había
5     hubo :había
6     hubo :había
7     tendrán :tendrían
S6: Test4
```

```
1     vivió :vivía
2     limpió :había limpiado
3     hizo :había hecho
4     preparó :había preparado
5     comió :había comido
6     tendrán :tendrían
7     necesitarán :necesitarían
8     sabía :supo
S7: Test1
S7: Test4
1     hacía :había hecho
2     comía :había comido
3     leyó :había leido
4     tendrán :tendrían
5     parecer :parece
6     sigue :seguir
7     embarazamos :embarazarnos
S8: Test1
1     fue :era
2     tendría :tendrá
3     saldría :saldrá
4     gustaba :gustó
5     podemos :podríamos
6     estamos casados :hemos estado casados
S8: Test4
1     se levantó :se había levantado
2     hizo la cama :había hecho la cama
3     le gustaba :gustó
4     estaba leyendo :leyó
5     tenía :tuvo
S9: Test1
1     sabía :supo
S9: Test4
1     fue :era
2     gustó :gustaba
3     limpió :había limpiado
4     hizo :había hecho
5     hubo desayunado :había desayunado
S10: Test1
1     fue :era
2     puesto :puso
3     tenía :tuvo
S10: Test4
1     había levantarse :se había levantado
2     había preparando :había preparado
3     leyó :había leido
4     bañarme :me baño
5     olvidó :había olvidado
6     llamado :llamaba José Smith
S11: Test1
1     fue :era
2     vivió :vivía
3     fue :era
4     empezó a barrer :había barrido
5     hizo :había hecho
6     preparó :estaba preparando
7     fue :era
8     olvidó :había olvidado
```

```
9     dijo :decía
10    fue :era
11    posee :poseer
S11: Test4
1     vivió :vivía
2     estaba :estuvo
3     limpió :había limpiado
4     gustaba :gustó
5     estaba :estuvo
6     aceptó :aceptar
7     aceptó :aceptar
8     podría :podía
S12: Test1
1     hubo agua :había
2     hizo :hacía
3     preparó :preparaba
4     hubo :había
5     hubo :había
6     hubo :había
7     fueron :eran
8     gustaba :gustó
9     fue :era
10    dirá :diría
11    tuvo :tenía
12    tuvimos :teníamos
S12: Test4
1     preparó :estaba preparado el desayuno
2     podía :podría
3     enseñando :enseñar
S13: Test1
1     hubo :había
2     fue :era
3     hubo :había
4     alegraba :alegró
5     hubo :había
6     dijo :decía
7     creciendo :crecer
8     aprendiendo :aprender
9     podrá :podría
S13: Test4
1     fue :era
2     acostado :acostó
3     levantada :haberse levantar
4     gustaba :gusto
```

APPENDIX I: AUGMENTED ERRORS

```
Counts for category 315:
    Test1:  60
    Test4:  67
S1: Test1
1     hacía: hizo: imp: pret (irr)
S1: Test4
1     necesitó: necesitaba: pret: imp
S2: Test1
S2: Test4
1     se levantó: se había levantado: pret: past perf imp
2     fue: había ido: pret (irr): past perf imp (irr)
3     había puesto: puso: past perf imp (irr): pret
4     visita: visitar: PRESENT: inf
S3: Test1
1     he vivido: viví: past perf imp (irr): pret
S3: Test4
1     estaba: estuvo: imp: pret (irr)
2     habría: habrá: cond: FUTURE
3     decía: dijo: imp: pret(irr)
4     pidió: pedir: pret: inf
S4: Test1
1     gustó: gustaba: pret: imp
2     preparó: preparaba: pret: imp
3     estaba leyendo: estuvo leyendo: imp (irr): past prog pret (irr)
S4: Test4
1     fue: era: pret (irr): imp (irr)
2     estaba limpiando: había limpiado la casa: past prog: past perf imp
3     haciendo: había hecho: prog: past perf imp (irr)
4     poniendo: puesto: prog: perf (irr)
5     estaba: estuvo: imp: pret (irr)
6     tenía: tuvo: imp: pret (irr)
S5: Test1
1     dormía: durmió: imp: pret (irr)
2     averiguar: averiguo: PRESENT: inf
3     probaba: probó: imp: pret
4     tomaba: tomó: imp: pret
5     olvidó: habia olvidado: pret: past perf imp
6     continuar: continuo: PRESENT: inf
S5: Test4
1     dormir: durmió: imp: pret (irr)
2     limpió: había limpiado: pret: past perf imp
3     hizo: había hecho: pret (irr): past perf imp (irr)
4     preparó: había preparado: pret: past perf imp (irr)
5     leyó: había leido: pret: past perf imp (irr)
6     estaba casado: me casé: past perf imp (irr): pret
S6: Test1
1     dormía: durmió: imp: pret (irr)
2     hubo: había: pret (irr): imp
3     hubo: había: pret (irr): imp
4     hubo: había: pret (irr): imp
5     hubo: había: pret (irr): imp
6     hubo: había: pret (irr): imp
```

```
7     tendrán: tendrían: FUTURE: cond
S6: Test4
1     vivió: vivía: pret: imp
2     limpió: había limpiado: pret: past perf imp (irr)
3     hizo: había hecho: pret (irr): past perf imp (irr)
4     preparó: había preparado: pret (irr): past perf
5     comió: había comido: pret (irr): past perf
6     tendrán: tendrían: FUTURE (irr): cond
7     necesitarán: necesitarían: FUTURE: cond
8     sabía: supo: imp: pret (irr)
S7: Test1
S7: Test4
1     hacía: había hecho: imp: past perf imp (irr)
2     comía: había comido: imp: past perf imp (irr)
3     leyó: había leido: pret: past perf pret (irr)
4     tendrán: tendrían: FUTURE: cond
5     parecer: parece: inf: PRESENT
6     sigue: seguir: PRESENT: inf
7     embarazamos: embarazarnos: PRESENT: inf
S8: Test1
1     fue: era: pret (irr): imp (irr)
2     tendría: tendrá: cond (irr): FUTURE
3     saldría: saldrá: cond (irr): FUTURE
4     gustaba: gustó: imp: pret
5     podemos: podríamos: PRESENT: cond (irr)
6     estamos casados: hemos estado casados: pres perf: pres perf perf
S8: Test4
1     se levantó: se había levantado: pret: past perf imp
2     hizo la cama: había hecho la cama: pret: past perf imp (irr)
3     le gustaba: gustó: imp: pret
4     estaba leyendo: leyó: past perf imp: pret
5     tenía: tuvo: imp: pret (irr)
S9: Test1
1     sabía: supo: imp: pret (irr)
S9: Test4
1     fue: era: pret (irr): imp (irr)
2     gustó: gustaba: pret: imp
3     limpió: había limpiado: pret: past perf imp
4     hizo: había hecho: pret (irr): past perf imp (irr)
5     hubo desayunado: había desayunado: past perf pret (irr): past perf imp
S10: Test1
1     fue: era: pret (irr): imp (irr)
2     puesto: puso: perf (irr): pret (irr)
3     tenía: tuvo: imp: pret (irr)
S10: Test4
1     había levantarse: se había levantado: imp inf: past perf imp
2     había preparando: había preparado: imp prog: past perf imp
3     leyó: había leido: pret: past perf imp
4     bañarme: me baño: inf: PRESENT
5     olvidó: había olvidado: pret: past perf imp
6     llamado: llamaba: perf: imp
S11: Test1
1     fue: era: pret (irr): imp (irr)
2     vivió: vivía: pret: imp
3     fue: era: pret (irr): imp (irr)
4     empezó a barrer: había barrido: pret: past perf imp
5     hizo: había hecho: pret (irr): past perf imp (irr)
```

```
6     preparó: estaba preparando: pret: past prog imp
7     fue: era: pret (irr): imp (irr)
8     olvidó: había olvidado: pret: past perf imp
9     dijo: decía: pret (irr): imp
10    fue: era: pret (irr): imp (irr)
11    posee: poseer: PRESENT: inf
S11: Test4
1     vivió: vivía: pret: imp
2     estaba: estuvo: imp: pret (irr)
3     limpió: había limpiado: pret: past perf imp
4     gustaba: gustó: imp: pret
5     estaba: estuvo: imp: pret (irr)
6     aceptó: aceptar: pret: inf
7     aceptó: aceptar: pret: inf
8     podría: podía: cond (irr): imp
S12: Test1
1     hubo agua: había: pret (irr): imp
2     hizo: hacía: pret (irr): imp
3     preparó: preparaba: pret: imp
4     hubo: había: pret (irr): imp
5     hubo: había: pret (irr): imp
6     hubo: había: pret (irr): imp
7     fueron: eran: pret (irr): imp (irr)
8     gustaba: gustó: imp: pret
9     fue: era: pret (irr): imp (irr)
10    dirá: diría: FUTURE (irr): imp (irr)
11    tuvo: tenía: pret (irr): imp
12    tuvimos: teníamos: pret (irr): imp
S12: Test4
1     preparó: estaba preparado el desayuno: pret: past prog imp
2     podía: podría: imp: cond (irr)
3     enseñando: enseñar: prog: inf
S13: Test1
1     hubo: había: pret (irr): imp
2     fue: era: pret (irr): imp (irr)
3     hubo: había: pret (irr): imp
4     alegraba: alegró: imp: pret
5     hubo: había: pret (irr): imp
6     dijo: decía: pret (irr): imp
7     creciendo: crecer: prog: imf
8     aprendiendo: aprender: prog: imf
9     podrá: podría: FUTURE: imp (irr)
S13: Test4
1     fue: era: pret (irr): imp (irr)
2     acostado: acostó: perf: pret
3     levantada: haberse levantar: perf: inf perf
4     gustaba: gustó: imp: pret
```

APPENDIX J: GLOSSARY

case assignment

Case theory says that every noun in a sentence must receive case or the sentence will be unacceptable. Nouns typically get case from verbs and prepositions. In this sentence, 'The cat gave the dog a biscuit in a bowl.', the noun 'cat' receives *nominative case*, the noun 'dog' receives *accusative case* and the noun 'biscuit' receives *dative case* from the verb 'saw'. The noun 'bowl' receives *oblique case* from the preposition 'in'. If the sentence had instead read, 'The cat gave the dog a biscuit a bowl.', it would be unacceptable because 'bowl' is not getting case. In the corpus an error is marked category 318 when a noun is not getting case.

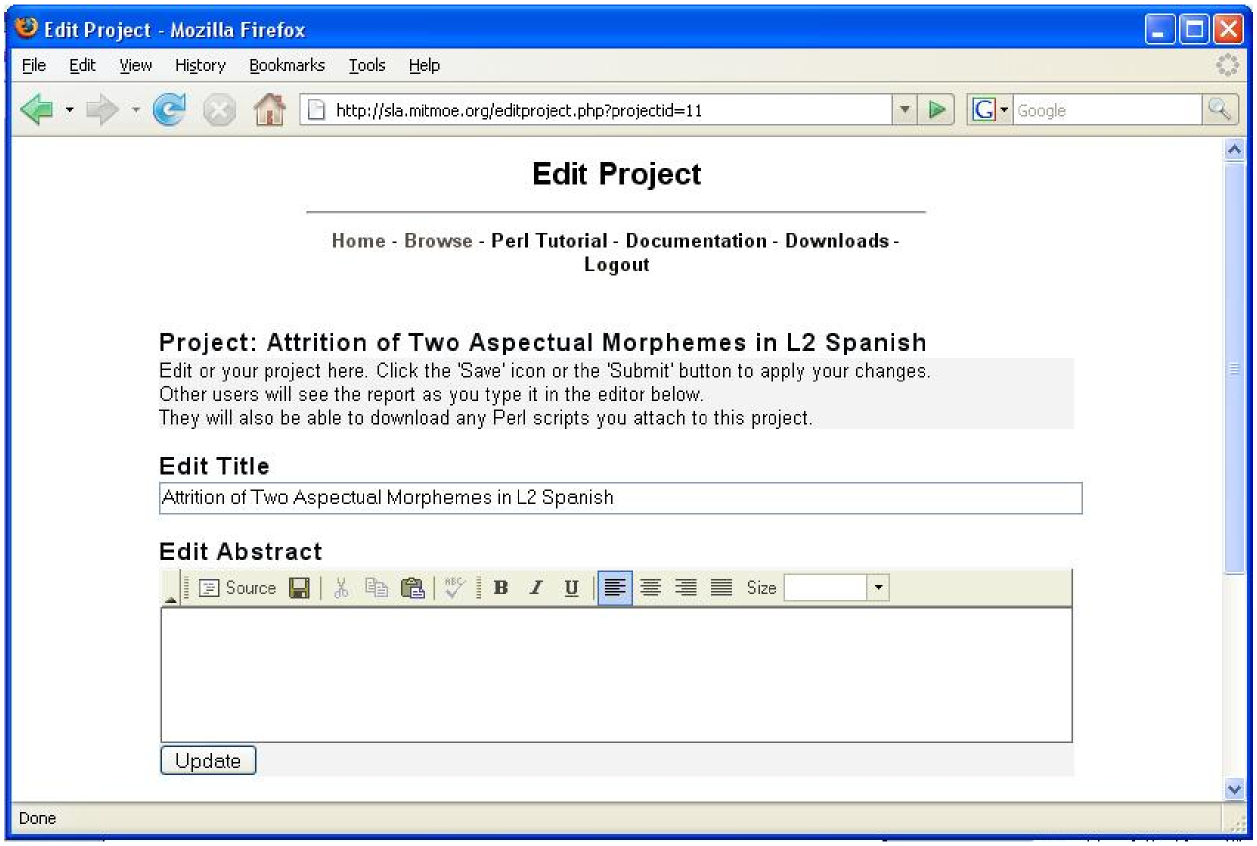
theta roles

A theta role or θ-role is the semantic role a noun phrase plays in a sentence, as opposed to the syntactic roles such as Subject and Object. The interpretation used here is that verbs assign theta roles to nouns. An error in the corpus will be marked with category 317, an error in theta role assignment, when a verb assigns a theta role and there is no noun to receive it. For example, if the speaker were to say 'he saw' without specifying what was seen.

T-units

A t-unit, or *minimal terminable unit*, is an independent clause with all of its dependent elements. For example, the sentence 'The cat gave the dog a biscuit when the dog asked for it.' has one independent and one dependent clause; therefore, it has one t-unit. The sentence 'The cat meowed and the dog barked.' has two independent clauses and two t-units.

APPENDIX K: WEBSITE SCREENSHOTS


Edit Project - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://sla.mitmoe.org/editproject.php?projectid=11
Google
Source
Size
Update
Done

Edit Project

Home - Browse - Perl Tutorial - Documentation - Downloads - Logout

Project: Attrition of Two Aspectual Morphemes in L2 Spanish

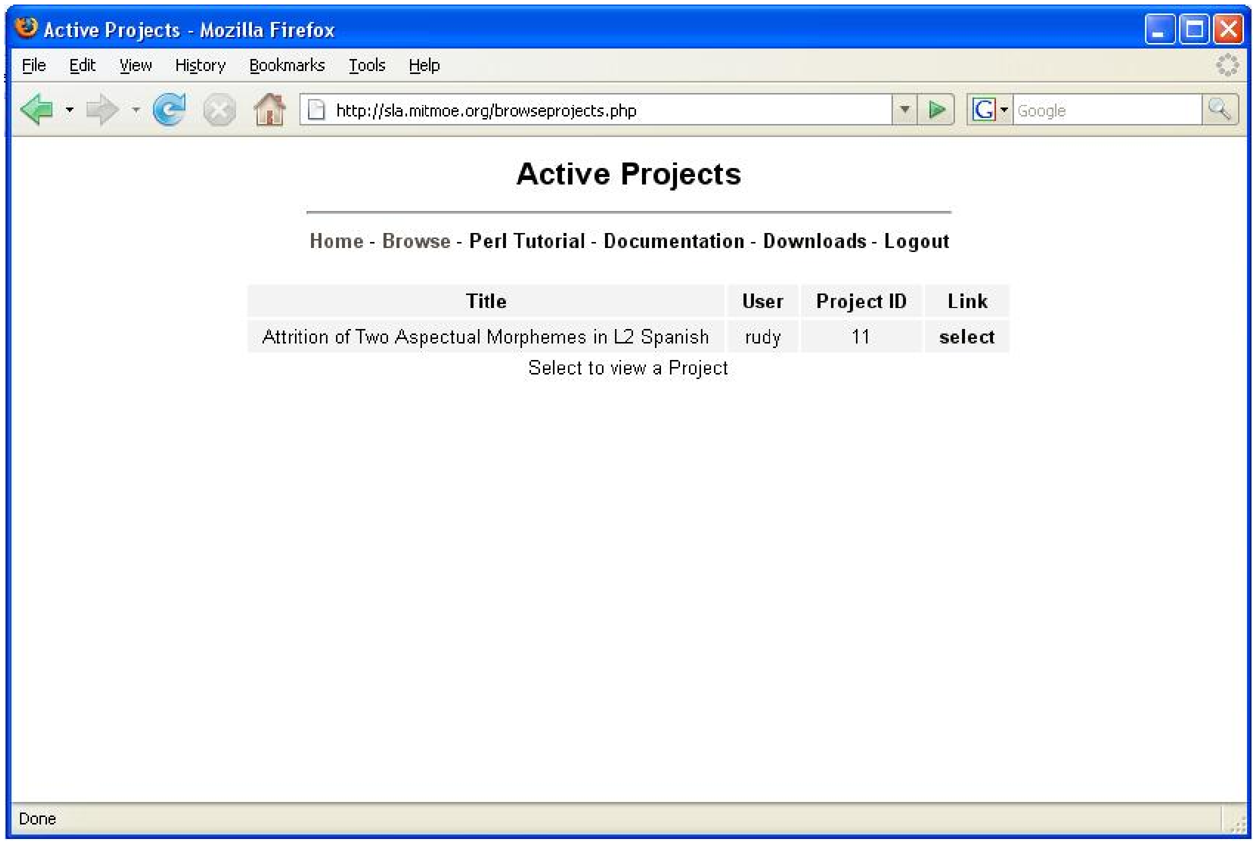
Edit or your project here. Click the 'Save' icon or the 'Submit' button to apply your changes.
Other users will see the report as you type it in the editor below.
They will also be able to download any Perl scripts you attach to this project.

Edit Title

Attrition of Two Aspectual Morphemes in L2 Spanish

Edit Abstract

Image 35: website, edit project page.


Active Projects - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://sla.mitmoe.org/browseprojects.php
Google
Done

Active Projects

Home - Browse - Perl Tutorial - Documentation - Downloads - Logout

Title	User	Project ID	Link
Attrition of Two Aspectual Morphemes in L2 Spanish	rudy	11	**select**

Select to view a Project

Image 36: website, browse projects page.