SCHWA

A DICTIONARY PRONUNCIATION DATABASE SYSTEM

by

ERIC ROCHESTER

(Under the direction of William A. Kretzschmar, Jr.)

ABSTRACT

Although pronunciations have been an integral feature of dictionaries for over two hundred years, they have never been the most important aspect of lexicography. Consequently, when dictionary editors adopted computer technology, pronunciations benefited from these new tools less than did other aspects of dictionary production. Recently, however, new technologies such as Unicode and XML have made it possible to work with pronunciations more easily and effectively. Schwa, a database system for managing and editing lexicographical pronunciations, incorporates these and other technologies to facilitate working with pronunciations. This dissertation describes Schwa, the design decisions that went into creating it, and some of the history and theory of lexicography that lie behind it.

INDEX WORDS: Lexicography, Dictionaries, Pronunciation, Phonetics, Phonology, Orthoepy, Computers, Unicode, XML, Databases

SCHWA
A DICTIONARY PRONUNCIATION DATABASE SYSTEM

by

ERIC ROCHESTER

B.A., Southern Adventist University, 1993

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2004

© 2004

Eric Rochester

All Rights Reserved

Schwa

A Dictionary Pronunciation Database System

by

Eric Rochester

Approved:

Major Professor:	William A. Kretzschmar, Jr.
Committee:	Michael A. Covington Nelson Hilton Stephen Ramsay

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2004

DEDICATION

This work is dedicated to my wife, Jackie, who has shown extraordinary patience throughout this process.

ACKNOWLEDGMENTS

The production of a project of this size takes a lot of work and patience, not just from those directly doing the coding and writing, but also from friends, relatives, and associates.

First, I thank my parents for their support and my wife, Jackie, for her unconditional love, support, and patience.

I also thank my dissertation committee: William A. Kretzschmar, Jr., Nelson Hilton, Michael Covington, and Stephen Ramsay. Without their input and guidance, this would not have been. Also, I want to thank all the other professors at the University of Georgia with whom I have had the privilege of studying.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

Chapter 1

Introduction

"*Glorious*," said Steerpike, "is a dictionary word. We are all imprisoned by the dictionary."

— *Mervyn Peake*, Titus Groan

From the first generally acknowledged English dictionary, Robert Cawdrey's *A Table Alphabeticall*, to the voluminous *Oxford English Dictionary*, these reference works have been major undertakings. The task of selecting entries, writing definitions, and gathering illustrative quotes for tens or hundreds of thousands of words is itself a staggering achievement. At that point, however, the logistics of layout, typography, printing, marketing, and distribution have only begun.

To manage these tasks and to make them feasible, lexicographers have developed tools, the most notable of which is the note card. Note cards allow lexicographers to break the entries and the information in them into manageable pieces. Then these smaller pieces of information could be combined and rearranged into the most appropriate organization for the specific task at hand.

Dictionary editors used some ingenious means to impose order on the chaos that threatened to overwhelm them. In the first building that James A. H. Murray used when he started work on the *Oxford English Dictionary* (*OED*), the walls were lined with pigeon holes that he could use to organize the citation slips, while still making them easy to move around. To process the citation slips as they poured in at the rate of 1,000 a day, first someone would look over the slips to make sure there were no egregious errors. Next, someone, often his children, would alphabetize them so that a more skilled assistant could sort the words by part of speech and by sense. Next, the quotes within each group were arranged chronologically. Then a sub-editor would look over each group to determine if a word's senses needed to be further subdivided. Finally, Murray himself would look at the bundles, fine-tune the subdivisions, and add the etymologies and pronunciations (Murray 186–87). The editors of the *Middle English Dictionary* used another tool to help sort through the citations for each word. Oscar Johnson, the associate editor for the *MED* in the 1930s and 1940s, fashioned a "sorting board," which looked like a greeting card display rack. This board made it easy to view the various citations for a word and to group them spatially in two dimensions. Citations that had the same sense could be stacked, and those with related senses could be clustered. Likewise, the stacks and clusters could be easily split back apart, as the editor's understanding of a word's meanings developed (Kuhn 27–29).

Once computers were available, however, lexicographers quickly recognized the importance of them to their work. Because computers could manage huge amounts of data and perform repetitious tasks without complaining or losing focus, they were the perfect tool to use in managing the voluminous amount of information that goes into a dictionary and in doing much of the mindless drudgery involved in lexicography.

However, one aspect of lexicography that has not benefited from computerized tools as much as other aspects is pronunciation. Traditionally, pronunciation has been a secondary, or even tertiary, consideration of lexicography. No dictionary even attempted to provide any guide to pronunciation for over 100 years after Cawdrey's *Table Alphabeticall*. Even the first dictionary that did attempt pronunciations, Nathan Bailey's *Universal Etymological English Dictionary*, only showed the word's primary stress. Since then, although the way that dictionaries have approached and handled pronunciations has improved, they still raise a number of issues—social and linguistic—beyond what the rest of the entry raises.

Because of pronunciations' relative lack of importance and the problems they entail, computerized tools for pronunciations have tended to be less sophisticated than those used for the rest of lexicography.

However, the application of computers to other aspects of dictionary production has provided enormous benefits. Some parts of producing dictionaries are significantly easier: producing, managing, and searching the citation file is far faster with a computer. Other aspects of dictionary production can be radically improved with the use of computers: through applying techniques of computerized corpus linguistics, the production of definitions can be moved from a largely intuitive activity to one with some empirical foundation. Also, computerizing the publishing process has made that aspect of dictionary production far less costly and labor-intensive.

It is important to remember that new tools, computers especially, are not a panacea, and this is true in applying computers to lexicography. First, computers have only partially addressed some of the previous problems with creating dictionaries. This was particularly true when dictionary production was just beginning to be computerized from the 1960s through the 1980s. Although dictionaries make use of a wide variety of typographic symbols, particularly in their pronunciations, computer systems of the time had trouble providing these characters, so typographic production and printing often relied upon non-standard, ad hoc systems. These difficulties in representing pronunciations helped to sidetrack and retard advances in how pronunciations are handled in dictionaries. Other problems are more theoretical. Certain aspects of dictionaries—definitions and citations, especially—are easily improved by computers, corpora, and databases, while other aspects—such as pronunciations—are less easily helped by computer processing. Because of this, dictionary editors may be tempted to pay less attention to pronunciations because they require more work. This is exacerbated because definitions have always been *the* primary aspect of dictionaries anyway. Fortunately, as computers have developed over the last few years, many of the particular problems presented by dictionary pronunciations have been addressed by the computing world at large. However, the dictionary industry already has a considerable amount invested in its current systems, and this has prevented it from taking advantage of these developments.

In Chapter 2 of this dissertation, I will examine the history of lexicography, particularly how pronunciations have been handled, including the social and theoretical issues involved in choosing a pronunciation and the balance between linguistic realities and the expectations of dictionary users. Aside from these larger issues of how dictionaries fit into society, there are also the more prosaic, yet still very important, issues of how the pronunciations should be represented within the dictionary. Over time and even today, different dictionaries use very different schemes to represent pronunciations. Although many dictionaries in Britain and Europe use the International Phonetic Alphabet (IPA) to represent pronunciations, American dictionaries still use diacritic respellings of the head words. How well these systems work is a question that should be considered in determining how to handle this data with a computer. In Chapter 3, I will evaluate a number of possible technologies and how they apply to the specific problems involved in managing and producing pronunciations. In Chapter 4, I will suggest solutions to the problems raised by dictionary pronunciations in light of the technical resources available. I will describe how these solutions contribute to the design and development of Schwa, a computer application that accompanies this dissertation and that facilitates working with lexicographical phonetic data. Finally in Chapter 5, I will explore some of the issues this tool raises concerning both the technologies involved and the advantages and drawbacks this program might present in the work of writing dictionary pronunciations.

Chapter 2

Lexicographical and Phonetic Issues

Lexicographer — A writer of dictionaries, a harmless drudge.

— *Samuel Johnson,* Dictionary

To understand dictionaries as they are today, it is necessary to look at their history and at the social forces that influenced them and were, in turn, influenced by them. In this chapter, I look first at the general history of lexicography, with a particular emphasis on how dictionary entries have developed. Then, I summarize current practices in writing pronunciations. Finally, I look at how dictionary writers have used computer technology to ease their work load and to improve their products.

Although simple dictionaries have been available for thousands of years in the form of bilingual glossaries, the first English dictionary is usually considered to be *A Table Alphabeticall*, published by Robert Cawdrey in 1604. The information it provided for its 3,000 entries was rudimentary, although he does include some simple etymological information: he uses a "g" to indicate that a word derives from Greek. However, he included no information on the pronunciation of its entries (Landau, *Dictionaries* 48). Another early dictionary was Henry Cockeram's 1623 *The English Dictionarie: or, An Interpreter of Hard English Words*. At this time, most dictionaries only covered difficult words, and this dictionary and others like it acted as a kind of bilingual simple English-difficult English dictionary. One section of Cockeram's dictionary listed difficult words and simple synonyms; another listed simple words and their difficult synonyms. Since many of the difficult words were based on classical languages and the sciences, he also included a section on mythology, nature, geographical terms, and such. This began the tradition of including encyclopedic information in dictionaries. He also tried to include usage information on words, such as whether a term was vulgar or cultivated (Landau, *Dictionaries* 50).

Henry Blount's 1656 *Glossographia: or, A Dictionary Interpreting all such Hard Words . . . as are now used in our refined English Tongue* also broke new ground in several respects. First, while all dictionaries writers at the time borrowed liberally from their predecessors, Blount went further and supplemented the word lists he took from other dictionaries with words he found in his own readings. He also altered the definitions he took from previous dictionaries and even rejected some entries he considered unsuitable. Second, he included two woodcuts, thus making his dictionary the first to boast illustrations (Landau, *Dictionaries* 51).

In 1658, Edward Phillips published *The New World of English Words*. Although this was nearly a copy of Blount's *Glossographia*, it also included some innovations. The primary one was including a list of specialists and giving the impression that they had been consulted on and contributed to the dictionary, although it is doubtful that they actually did so. Phillips also indicated the subject field of each term and continued the practice of indicating the language of origin (Landau, *Dictionaries* 52).

The first dictionary to go beyond the hard words tradition and include common terms as well as difficult ones was John Kersey's 1702 *A New English Dictionary*. Although its definitions tended to be inadequate because they were too short, this dictionary represented a break from those were modeled on Latin-English dictionaries and started a new trend in which dictionaries were based upon spelling books. Later, Kersey went on to revise Phillips' *New World of English Words*. This work was important because it included a list of multiple meanings for the same word (Landau, *Dictionaries* 53).

Up to this point, no dictionaries had attempted to indicate how their entry words should be pronounced. The first to do so was Nathan Bailey's *Universal Etymological English Dictionary, Supplement II*, published in 1721. It accented the stressed syllable in some head words. He also included extended etymological information. Previous dictionaries had, at most, indicated the language terms were immediately borrowed from. Bailey also included earlier forms in other languages. Later editions also included symbols to indicate whether words' were of impeccable or dubious correctness. This may have been more necessary than in other dictionaries because, unlike many of his contemporaries, he also included terms considered vulgar or taboo, such as *shite* and *fuck* (which he only defined in Latin) (Landau, *Dictionaries* 53–55; Bronstein 137). The first dictionary to include pronunciations as a standard feature in all of its entries was Thomas Dyche's *New General English Dictionary*, published in 1735. It also included an extensive commentary on pronunciation in its front matter (Bronstein 137).

Also, at this time definitions began to take their modern, multi-part form in Benjamin Martin's *Lingua Britannica Reformata*. He used numbered senses in his definitions and put more work into his distinctions than did the lexicographers before him (Landau, *Dictionaries* 60–61).

The first landmark English dictionary, of course, was written by Samuel Johnson. His plans for the dictionary were ambitious beyond what had been done before. In his 1747 "A Short Scheme for Compiling a New Dictionary of the English Language," he intimates that a dictionary should include information on pronouncing words (Congleton 60). Toward this goal, he proposes listing words beside words they rhyme with. For example, *tear* (cry) with *peer* and *tear* (rip) with *dare* (Congleton 61). However, by 1755 when Johnson published his *Dictionary of the English Language*, this system had been set aside. Instead, he used four ways to indicate the pronunciation of a word. First, he continued Bailey's system of marking the primary stress with accents. Second, he illustrated the sounds of letters and combinations of letters using common words. Third, he gave rules for determining a word's pronunciation from its orthography. And fourth, he provided further directions for irregular pronunciations by using alternate spellings, cross references, accents, directions, and respellings (Congleton 62). How innovative Johnson's handling of pronunciations was and how much influence it may have had on his successors is debatable. However, Johnson's dictionary did include a number of important features. His handling of definitions and illustrative quotations, while not original, was masterful (Landau *Dictionaries*, 64–65).

After Johnson, the eighteenth-century featured a number of pronunciation dictionaries, which were dominated by the work of two men: Thomas Sheridan and John Walker. Both made important contributions to lexicography and dictionary pronunciations. However, the extent to which they each balanced their own natural prescriptivist tendencies with real, descriptive pronunciations varied, as did how well they each fulfilled the public's expectations for dictionary pronunciations.

Thomas Sheridan contributed the most to lexicographic pronunciations and how they were technically represented, and his *Complete Dictionary of the English Language*, published in 1789, included a number of innovations. He used a system of detailed respellings, diacritics, and stress marks that continued to influence dictionary writers into the twentieth century. Also, of the two, his pronunciations were the most descriptively accurate, both for the standard forms and in the variant pronunciations he listed (Bronstein 139).

The other influential dictionary writer was John Walker, with his *Critical Pronouncing Dictionary and Expositor of the English Language*. It made use of many of the technical innovations that Sheridan introduced. However, he took a radically different approach to deriving pronunciations, as Walker's comments on Sheridan in the preface to Walker's dictionary illustrate:

> It must, indeed, be confessed, that Mr. Sheridan's Dictionary is greatly superiour to every other that preceded it; and his method of conveying the sound

> of words, by spelling them as they are pronounced, is highly rational and useful.—But here sincerity obliges me to stop. The numerous instances I have given of impropriety, inconsistency, and want of acquaintance with the analogies of the Language, sufficiently show how imperfect I think his Dictionary is upon the whole, and what ample room was left for attempting another, that might better answer the purpose of a Guide to Pronunciation. (qtd. in Sheldon 131–32)

This quote illustrates how Walker made use of Sheridan, how he deviated from him, and what his primary criteria were: consistency with spelling and analogy with other words (Sheldon 143). He regarded the spelling of a word as sacred and considered movements to reform spelling to make it more consistent with pronunciation as being exactly backward from how the process should work. A number of his pronunciations—in conflict with Sheridan's and other dictionaries of the time—matched the spelling of the words. For example, a number of contemporary dictionaries list the pronunciations of *super* (and many other words that begin with *su*-) as being /ʃupɚ/. Walker, however, gives only the modern pronunciation of /supɚ/. (Modern examples of words in which *su*- is still pronounced as /ʃu-/ would include *sure* and *sugar*.) Another example of his heavy-handed approach is his handling of pronunciations for jargon and other special terms. Modern dictionaries go to great lengths to secure the pronunciations of, for example, lawyers for legal terms and sailors for nautical terms. Walker explicitly did not do this, and in fact provided pronunciations that "corrected" those used by specialists (Sheldon 141).

Walker's dictionary and its pronunciations were incredibly influential. As Esther Sheldon states, "There can be no doubt that, if any one single person were to be named as the greatest influence on English pronunciation, that person would have to be Walker" (146). There are a number of reasons for his success. First, English teaching was based upon Latin, and sometimes English grammars were little more than translations and adaptations of Latin ones. Because English was considered to be an inferior, debased descendant of Latin or Greek, English syntax that conflicted with rules derived analogously from Latin were condemned. Moreover, because those writing the English grammars were often retired clergy, they expressed their disapproval in strongly moral terms (Landau, *Dictionaries* 244). This was the way the public expected linguistic authorities to talk, and Walker fulfilled those expectations well.

Another reason for his success was the social mobility of the time, which created a ready audience for such advice. After all, "It is the person who is moving to what he thinks is a superior station who fears that he will be stigmatized by the kind of language he learned at home—and often is" (McDavid 24). For the immigrants coming to the United States and for the upwardly mobile middle class, "Walker's advice was a much appreciated help" (Landau, *Dictionaries* 68). Those who were looking for instruction, not on how words *were* pronounced, but on how words *should be* pronounced, could turn to Walker and find advice that was logical and consistent, and therefore easy to follow, even if it was not wholly accurate.

In America, Walker was also successful because of the political and linguistic realities of colonization. In 1779, Benjamin Franklin said that a pronunciation dictionary would be useful in America because there were a number of words that British writers used, and which all Americans could read and understand, but which Americans did not know how to pronounce (qtd. in Read, "Social Impact" 69–70). This attitude of colonial linguistic insecurity continued in America and helps to explain Americans' continued need for linguistic guidebooks and authorities. William Dean Howells perhaps best explains and expresses this insecurity:

> If one has moved in good English society, one has no need even to ask how a word is pronounced, far less to go to the dictionary; one pronounces it as one has always heard it pronounced. The sense of this gives the American a sort

> of despair, like that of a German or French speaking foreigner, who perceives that he never will be able to speak English. (qtd. in Read, “Social Impact” 73)

In fact, Walker was so popular and influential in America that Noah Webster complained about Americans’ being “misled” by him and that at least one British person visiting here mentioned that Americans generally spoke well, except for where they had adopted Walker’s pronunciations (Read, “Social Impact” 71, 72).

The final reason why Walker’s dictionary was so successful was the influence it had on other dictionaries. Because the pronunciations were careful, logical, and consistent, they appealed to other editors. His dictionary went through a number of editions, more than Sheridan’s. Moreover, when a new edition of Sheridan’s dictionary was published in 1788 after his death, its new editor, Stephen Jones, completely reworked the pronunciations, bringing them more in line with those of Walker (Sheldon 146). In fact, Walker continues to influence us today, not only in his individual pronunciations, but even his idea about what constitutes a correct pronunciation (Sheldon 130). Esther Sheldon, in studying Walker’s pronunciations, sums up the causes of his success: “The reason for this is, I believe, that while Sheridan reflects the speech of his time better, Walker satisfies the temper of his time better, and its demand for linguistic regulation and reform” (Sheldon 146).

On the other side of the Atlantic, the first American English dictionary was by Samuel Johnson, Jr. It was a slim volume, only 198 pages, designed to be used by school children, not as a general-purpose American dictionary. The task of writing a general American dictionary was left to Noah Webster. His two dictionaries—the 1806 *Compendious Dictionary of the English Language* and the 1828 *American Dictionary of the English Language*—were part of his project to provide a distinctly American dictionary, one that reflected the linguistic realities of American speech, not merely recommending British usage. Webster spent ten years working on etymologies for the *American Dictionary*, but because of his rejection of the discoveries of Joseph Grimm and other philologists and because of his preconceived ideas about the history of languages, his etymologies were bad, even for his time. Some of his other, more fortunate innovations included listing the principle parts of irregular verbs and appending tables of weights and measures U. S. population figures (Landau, *Dictionaries* 69). He also included information about pronunciations; however, he often relied upon Jones's edition of Sheridan and, hence, upon Walker's pronunciations. In general, however, his *Dictionary* was more descriptive than Walker's was, although it missed the mark of being a general American dictionary, since much of his usage and pronunciation do not reflect the usage of America in general, but only of New England (Krapp 366).

Webster's major competitor was Joseph Worcester. He produced three dictionaries: *Comprehensive Pronouncing and Explanatory Dictionary of the English Language* (1830), *Universal and Critical Dictionary of the English Language* (1846), and *Dictionary of the English Language* (1860). In general, his dictionaries were better than Webster's. Although the definitions were generally shorter, the coverage of the vocabulary was better. Only the last of these included etymologies, but these were also better than Webster's. Also, he included quotations to illustrate his definitions (Landau, *Dictionaries* 72–74). In Worcester's dictionaries, pronunciations were handled more carefully than in Webster's dictionaries. Pronunciations are more clearly indicated, and Worcester listed variations based upon twenty-six different dictionaries and linguistic treatises (Krapp 371). However, his dictionaries did have a distinctly British bias to them (Bronstein 139).

This bias toward British pronunciations was also evident in the first pronunciation dictionaries in America. R. S. Coxe produced *A New Critical Pronouncing Dictionary of the English Language* in 1813, B. Allison, *The American Standard of Orthography and Pronunciation and Improved Dictionary of the English Language* in 1815, and W. Bolles, *A Phonographic Pronouncing Dictionary of the English Language* in 1846 (Bronstein 139).

One curious experiment in pronunciation dictionaries was attempted in 1855. *The American Phonetic Dictionary of the English Language, Adapted to the Present State of Literature and Science; with pronouncing vocabularies of Classical, Scriptural and Geographical Names* was designed by Nathaniel Storrs and compiled by Dan Smalley. This dictionary used a phonetic alphabet invented by Benn Pitman, Elias Longley, A. J. Ellis, and others for the dictionary. The words were listed in alphabetical order, but the headwords were given as transcribed into the phonetic alphabet. The definitions were also printed using phonetic transcriptions. Although the phonetic alphabet was good, its use in both headwords and definitions proved to be too much. As George Philip Krapp points out, "It proves that even a phonetic alphabet does not hold the mirror up to nature, but that after all it is only an approximate, therefore conventional, representation of real speech, like the traditional alphabet" (374).

In England in the late nineteenth century, the Philological Society sponsored one of the largest, most impressive dictionary projects in the world. The new dictionary would be historical in focus and would not be merely a revision of any previous dictionary, but would instead be original, hence its first title, *A New Dictionary on Historical Principles*. Although he was the third editor, work on this dictionary really began when James A. H. Murray became editor in 1879. When this dictionary—eventually to be called the *Oxford English Dictionary* (*OED*)—was finally finished in 1928, nearly half of its contents had been edited by Murray. The *OED* provides the historical development of each word, with illustrative quotations from various sources for each of the many, detailed senses that it includes. Its etymologies are as a whole still the most complete and authoritative of any dictionary. It represented an outstanding achievement of lexicography, in spite of some imperfections. One such imperfection was its pronunciations and the system of pronunciations used. Although they were sufficient, they were not up to the standard of the rest of the dictionary (Landau, *Dictionaries* 80–81).

After the *OED*, throughout the twentieth century, in England a series of authoritative pronunciation dictionaries were produced by Daniel Jones. He was the first to use the International Phonetic Alphabet (IPA). In 1913 he wrote the *Phonetic Dictionary of the English Language* (with H. Michaelis) and in 1917 the *English Pronouncing Dictionary* (*EPD*). This described “Received Pronunciation” (RP), a dialect of British English that was based upon educated pronunciation in London and the Home Counties, but in the nineteenth century was spoken by the upper class throughout the country. This went through a number of editions and was revised by A. C. Gimson in 1967. In 1997 a new *EPD* was published, based upon the Daniel Jones’ original and edited by Peter Roach and James Hartman. It includes new entries and American pronunciations, and it continues to be an authoritative dictionary for RP today. Since the 1997 edition of the *EPD*, however, the *Longman Pronunciation Dictionary* and the *Oxford Dictionary of Pronunciations for Current English* also provide authoritative pronunciations for RP (Bronstein 140–41; Landau, *Dictionaries* 37).

The only major pronouncing dictionary for the United States in the 20th century is J. S. Kenyon and T. A. Knott’s 1944 and 1945 *A Pronouncing Dictionary of American English* (*PDAE*). Kenyon also authored the extraordinary preface on pronunciations given in the front matter to *Webster’s Second New International Dictionary* (*NID2*), which contain information on basic phonetics and phonology, the IPA, and numerous other topics. The *PDAE* was explicitly descriptive. It included a number of variants for each entry and incorporated data from the Linguistic Atlas project and other sources (Bronstein 142–43). Unfortunately, the *PDAE* has never been updated, so that Sidney I. Landau cites *Webster’s Third New International Dictionary* (*NID3*) as “the best source for American English pronunciations” (*Dictionaries* 37). However, since Landau wrote this, *The Oxford Dictionary of Pronunciation for Current English* (*ODP*) has been published, which covers both British and American pronunciations and uses IPA for both sets (Upton, Kretzschmar, and Konopka).

One specialized kind of pronunciation guide on both sides of the Atlantic is targeted to broadcasters. In England, *Broadcast English* (1928–39) and the *BBC Pronouncing Dictionary of British Names* (1971, 1983) serve this function. In America, W. Cabell Greet's 1948 *World Words* served as a pronunciation guide for CBS. It primarily contained foreign words. NBC, however, used James F. Bender's 1943 *NBC Handbook of Pronunciation*, which used both diacritic respelling and IPA (Bronstein 143–45).

Another source of pronunciations, more familiar to the general public, is standard desk dictionaries. In Britain, these have used a number of systems, including diacritic respelling, IPA, and IPA variants, although recently all have moved to using the IPA. In the United States, however, dictionaries only use diacritic respelling. Beginning with Random House's *American College Dictionary,* which introduced the schwa character to its pronunciations in 1947, many dictionaries use the schwa (ə) and eng (ŋ) characters, although it appears that these adoptions in no way represent a move toward using the IPA generally.

The most influential general and desk dictionaries in the United States have been produced by Merriam-Webster, the commercial inheritors of Noah Webster's dictionaries. They produce two lines of dictionaries: the *Webster's New International Dictionaries,* also known as the "unabridged," and the *Merriam-Webster's Collegiate Dictionaries,* which typically follow the policies of the unabridged dictionary on pronunciations and other matters. In the twentieth century, the *Second New International Dictionary*'s (*NID2*) major contribution to phonetics was the preface by J. S. Kenyon. The *Third New International Dictionary* (*NID3*) went further, however. Its pronunciation editor, Edward Artin, tried to make systematic use of linguistic research and greatly improved the quality of the pronunciations. Unfortunately, the transcription system used in *NID3* marred its usefulness by being overly complex.

In general, dictionary pronunciations have been poorly handled, more so than any other aspect of the dictionary (Hulbert 55), and analyzing how dictionaries do work with pronunciation is muddied by the apparent lack of a well articulated theoretical approach to pronunciations and by a poorly defined audience and goals for the pronunciations (Gimson 115). When pronunciations have been written in the past, usually they are based upon either prescriptivist rules, personal idiolect, or observation (Read, “Theoretical Basis” 87).

In analyzing the handling of pronunciations in lexicography, the starting point and center of the inquiry should be the usefulness of the pronunciations for the user. As Landau points out, this is the emphasis of Johnson’s 1747 *Plan* and it should still be “the first rule of good dictionary making” (*Dictionaries* 359). Of course, this immediately raises a problem, since at least one study has indicated that only a minority of regular dictionary users even want pronunciations and even fewer occasional users do (Quirk 81). Overlooking that, however, there are two identifiable objectives that influence how decisions involving dictionary pronunciations are made. The first is the target audience, whether they be specialists, the general public, or both, native speakers or foreign language learners. The second is the purpose of the pronunciations, whether it be to describe in detail how words are spoken, using many variations, or to indicate how a word is pronounced within the context of the user’s own dialect or, alternatively, within the standard form of the language.

Whatever any individual dictionary publisher and editor decides those goals to be, there are essentially two main tasks involved in handling dictionary pronunciations: first, gathering possible pronunciations and choosing one or more from the many variants, and second, communicating that pronunciation in print (Gimson 116).

The first issue—choosing a pronunciation—is multifaceted. It first involves gathering evidence for pronunciations. Historically, this has often involved the pronunciation editor deciding how he or she pronounces words. This contributed to Noah Webster’s dictionaries’ New England bias. Later, Edward Artin tells how his predecessor at Merriam-Webster would spend much of his time muttering words to himself (126). The next more sophisticated way of gathering pronunciations involves surveys. These can range from informal ones in which the pronunciation writer polls colleagues for their pronunciation to the *Longman Pronunciation Dictionary* Pronunciation Preference Survey to the sophisticated surveys conducted for the various Linguistic Atlas projects (Wells). Obviously, the quality of data obtained by each of these kinds of surveys will be similarly variable. Both of these methods contribute to the dictionary's working collection of pronunciations. This may simply be the previous edition of the dictionary or a more elaborate database (electronic or on cards) of pronunciation entries or transcriptions.

For example, Edward Artin, the pronunciation editor for *NID3*, says that his primary concern during his work there was to gather evidence for pronunciations. When he began his work, the earliest pronunciation evidence he could find in the Merriam-Webster files primarily involved transcriptions taken from other dictionaries (Artin 125), so he began making transcriptions from radio and television, and he made use of data from the *Linguistic Atlas of New England* (*LANE*) (128). As a result of the project he began, *Merriam-Webster's Collegiate Dictionary*, 11th edition, (*MW11*) says that "The pronunciations in this dictionary are informed chiefly by the Merriam-Webster pronunciation file" (33a), which is a collection of 3x5 cards recording pronunciations transcribed from radio, television, and speeches since the 1930s, and which is the authority *MW11* relies upon for its pronunciations.

Unfortunately, a citation file of pronunciations has many of the same weaknesses as a standard citation file. First, it is limited to what those contributing to the file are exposed to. In the case of the Merriam-Webster file, this and the use of *LANE* could perpetuate their New England bias, although the citations from radio and television might introduce some pronunciations from other parts of the country. Either way, it is possible that much of the country is underrepresented in their files. Second, citation files tend to be targeted: a transcription is made only if support for a specific word is needed or if the pronunciation for some reason catches the transcriber's attention. Both of these factors make the authority of the pronunciation citation file less than perfect.

However, while definition writers now have large computerized corpora to give their definitions a more empirical basis, this development is unlikely to happen for pronunciation writers in the near future. For one thing, corpora of spoken language, even without phonetic transcriptions, are time-consuming and costly to produce (Landau, *Dictionaries* 324). The spoken corpora that do exist—the London-Lund corpus, the British National Corpus, the American National Corpus, Santa Barbara Corpus of Spoken American English, and others—tend to be too small for lexicography, which requires very large corpora just to get sufficient coverage of the lexicon. Also, the data from the various Linguistic Atlas projects are of limited utility for lexicographers, since the aims of these projects are different than the needs of lexicographers for writing pronunciations. For example, lexicographers need fairly complete coverage of the lexicon, which would be a waste of resources for most linguistic studies.

In the end, pronunciation writers wind up roughly where they began, intuition, but hopefully they have learned some lessons. First, they need to take many pronunciations into account consciously, and to work to train their ear and intuition to be aware of variant pronunciations. Also, the writer must be aware of the incredible variation in how words are pronounced and to be humble in his or her knowledge of pronunciations. In short, the dictionary pronunciation writer should work to become an informed, trained expert in pronunciations. All of this is simply to avoid providing incorrect or biased pronunciations. As Artin wrote, “How does one go about avoiding this sort of thing? The best answer I could think of was, by doing all the listening one possibly can” (127).

Once the evidence has been gathered, it must be evaluated and one or more pronunciations chosen to be included in the dictionary. In British dictionaries, this decision is somewhat simpler. They present Received Pronunciation (RP), which is the dialect of educated speakers in and around London. (The issues and variation involved in RP will not be covered here, since this treatment deals more specifically with American Standard pronunciation and its variants.) On the other hand, pronunciations in American dictionaries generally reflect what is referred to as “Standard American.” This is supposed to represent a form of American speech that is devoid of any distinctive regional features and is representative of educated speakers. The problems with this are well summarized by Arthur J. Bronstein when he compared the problems faced by American lexicographers to those faced by their British peers:

> one must recognise that the indication of pronunciation in North America does present any lexicographer who plans to enter only a single ‘type’ of American English pronunciation with a very difficult task. There is no single ‘prestigious’, ‘educated’, ‘acceptable’ or ‘standard’ dialect that exists throughout the continent, even if one did exclude the northeastern and southern United States. However, if not as homogeneous as supposed earlier, the regional differences among most educated speakers are relatively minor (when compared to the differences used by British English speakers). Thus, despite the large number of local and regional variant forms, communicative interferences are not typical. These differences do not easily permit a single label to embrace all of them, unless the term (label) used is understood to represent no single, preferred dialect of the language spoken in North America. (142)

This difficulty is exacerbated because dictionaries rarely define the group whose pronunciations they represent, beyond socially and linguistically vague terms like “cultivated,” “literate,” and “educated” (Pederson 130). For example, the “Guide to Pronunciation” in *MW11* states that it “attempts to include—either explicitly or by implication—all pronunciation variants of a word that are used by educated speakers of the English language,” although it also warns, “Among such speakers one hears much variation in pronunciation” (33a). In practice, this involves representing the prestige dialect, which Landau defines as “a dialect widely accorded respect by all social levels in a community because it is identified with well-educated people of high social and economic standing" (*Dictionaries* 220). In actuality, the prestige dialect is a useful abstraction, since it is not based upon any well-defined, coherent dialect and since there are no speakers who do more than approximate it to varying degrees. Instead, it is defined by a lack of recognizable regional markers (Upton, Kretzschmar, Konopka xiii–xiv). And although it is an abstraction, being able to reproduce it may be useful to those who wish to move into a higher social class and avail themselves of opportunities there.

Because the prestige dialect is an abstraction, how many variants to list is an issue. Should the lexicographer list more than one pronunciation if more than one is current, and if so, how does he or she choose which one to list? Simply listing all variants is not an option because of the high premium placed upon space in a dictionary. One way to approach this problem is to choose a method of transcription that is intentionally ambiguous, so that one grapheme represents more than one phonetic value simultaneously. However, as we shall see, this also has its drawbacks.

After deciding upon which pronunciations to include in the dictionary, they must still be represented in a graphical, printable form. There are two issues here, although they are interrelated and often confused. However, for the sake of clarity, I will consider them separately, insofar as possible, before considering them together.

The first issue is the level of representation to use in pronunciations. On the one hand, there are *phonetic* transcriptions, which are represented graphically by enclosing the transcriptions in square brackets ([. . .]). These describe the pronunciation in terms of points of articulation, that is, in terms of what the teeth, lips, tongue, and so forth are doing when the sound is produced. The problem with this is that the more specificity and detail in description of sounds, the more variants that need to be listed to capture all of the possible standard pronunciations. For example, one commonly cited variation in standard American speech is the vowel in *cot*, which can be realized as either [ɑ] or [ɔ]. This is systematic in American English for a certain class of words, and a decision to list both variants for all such words would take much of a dictionary's precious space.

On the other end of the spectrum are *phonemic* transcriptions, which are represented by enclosing the transcriptions between slashes (/. . . /). These are based upon phonological theory that posits abstract underlying sounds, called *phonemes*, which are actually realized in speech by different sounds, called *allophones*. In the example of *cot* above, [ɑ] and [ɔ] are phonetic allophones for the *phoneme* /ɑ/. Phonemes capture the fact that, for the purpose of distinguishing words, multiple actual speech sounds are used to represent, and are heard as, one sound. Phonemic transcriptions are good for saving space, but they intentionally hide a certain amount of phonetic variation. Unfortunately, some of the differences that they cover can be differences between prestige and marked regional forms, which would theoretically be allophones of the same phoneme. Since one purpose of dictionary pronunciations is to differentiate marked regional forms from the prestige forms, hiding these differences is a major drawback.

Today, American dictionaries use a phonemic system to represent pronunciations. Although the actual system of transcription—what symbols represent what sounds—varies from dictionary to dictionary, all the systems are phonemic in nature: they rely upon the native speaker's knowledge of how to pronounce words to define the symbols. This results in the symbols' definitions corresponding to the speaker's idiolect, not to the prestige dialect.

Fortunately, a compromise between these two forms of transcription is possible. This involves using a broad phonetic transcription. At its core the transcription is still phonetic, but it redefines some symbols to represent more than one sound in some contexts. The transcriptions are still essentially phonetic, but these redefinitions allow them to handle regular, predictable, standard variants more gracefully (Landau, *Dictionaries* 123; Sledd 136). For example, the *ODP* uses broad phonetic transcriptions to handle the variation between stressed and unstressed central vowels: that is, [ə] is used in both unstressed environments (*sofa*; [ˈsoʊfə]) and in stressed environments (*sun*; [sən], which other linguists might transcribe as [sʌn]) (*ODP* xvi). This allows the one symbol to cover both pronunciations in a regular, systematic manner, and it simplifies the transcription system. However, it does not prevent the reader from reading the transcription and discovering the standard pronunciation.

After considering the level of representation, the next issue is the system of pronunciation transcription to use. To represent pronunciations, American dictionaries currently use diacritic respelling. This makes use of the standard English alphabet, some digraphs based upon those characters, a few phonetic symbols (usually schwa and eng), and diacritics to modify those symbols. Respelling systems are not defined by sounds or points of articulation, but by key words. For example, in *American Heritage College Dictionary*, 4th edition, (*AHCD4*), *ă* is defined using "p**a**t."

This kind of system is based upon the assumption that the users are native speakers who are not experts in linguistics, that they will primarily only be in question about the words' stress, and that they want to know how to pronounce words within the context of their own dialect (Neufeldt 111–12; Pearsons 115). It provides no help for those, such as foreign language learners, who have little or no intuitive knowledge of English pronunciations. Furthermore, it also provides no help to native speakers who wish to know the pronunciation of a word in the prestige dialect. The primary virtue of respelling systems is that they leverage the native speaker's intuitive knowledge of English pronunciation, not just as a system, but also in the correspondences between pronunciation and spelling (Neufeldt 111). At heart, this system is phonemic, because the defining key words used, such as *pat* above, represent word classes. Thus an *a* in a pronunciation will represent a sound that many, if not all, native speakers will associate with the grapheme *a*; diacritics will simply specify which of the sounds it actually is.

While this ease of use would appear to be a major advantage for respelling systems, in practice, it is not. Even proponents of respellings admit that "a great many—possibly most—people who use dictionaries do not know how to use even the simple respellings found in current dictionaries” (Neufeldt 112). This situation is exacerbated by the number of respelling systems, since each dictionary uses its own.

Looking at the extended explanation for \t\[1] in *MW11* illustrates several things about respellings:

> \t\ as in **t**ie, a**tt**ack, la**te**, la**t**er, la**tt**er (IPA [t]). In some contexts, as when a stressed or unstressed vowel precedes and an unstressed vowel or \ᵊl\ follows, the sound represented by *t* or *tt* is pronounced in most American speech as a voiced flap produced by the tongue tip tapping the teeth-ridge (IPA [ɾ]). In similar contexts the sound represented by *d* or *dd* has the same pronunciation. Thus, the pairs *ladder* and *latter*, *leader* and *liter*, *parody* and *parity* are often homophones. At the end of a syllable \t\ often has an incomplete articulation with no release, or it is accompanied or replaced by a glottal closure. When \t\ occurs before the syllabic consonant \ᵊn\ as in *button* \ˈbə-tᵊn\, the glottal allophone is often heard. This may reflect a syllabication of \t\ with the preceding stressed syllable (i.e., \ˈbət-ᵊn\).
>
> Many speakers pronounce \t\ like \ch\ when it occurs before \r\ in the same syllable. (35a)

The first thing to notice is that, judging from the list of key words containing that sound, \t\ corresponds to at least two different sounds: *tie* [taɪ] and *latter* [ˈlæɾɚ]. However, the next item in the explanation is an equivalent symbol in IPA ([t]). Next, the explanation lists a set of exceptions to the symbol \t\ being pronounced either [t] or [ɾ]. It specifically mentions the phones [ɾ] and [tʃ], and it implies the phone [ʔ] (the “glottal allophone”). Obviously, the user is expected to have either an intuitive or a learned understanding of how /t/ is realized in various environments, since the symbol \t\ does not

[1] *MW11* uses \...\ to indicate the beginning and end of a pronunciation transcribed in their respelling system, and I follow that convention here.

actually provide this information. This quote also illustrates how complex a respelling system can be and how overwhelming.

The main alternative to respelling systems is the IPA. This is a system of symbols that is widely used by linguists, foreign dictionaries, and foreign-language dictionaries to describe pronunciations. The IPA is a phonetic system, that is, its symbols are defined in terms of sounds and points of articulation, not key words. Generally, pronunciation editors prefer to work in IPA. In fact, Edward Artin, the pronunciation editor for *NID3*, says that having to use respelling systems instead of IPA is one of the griefs of the job (Landau, *Dictionaries* 125).

The main arguments against using the IPA is that its symbols are Eurocentric, that is, that its vowel symbols especially more closely resemble graphemes from European languages, for example *i* and [i]. Even worse, English words transcribed into IPA can end up more closely resembling the orthographic spellings of other English words: *feet* is [fit] (Neufeldt 113; Pearsons 115). Another argument is that IPA requires listing too many variants, because of its phonetic nature.

However, while some may argue that the IPA symbology is biased toward European languages, the system is essentially arbitrary, as are the respelling systems (Bladon, et al., 126). Also, as has been mentioned, respelling systems' resemblance to English graphemes has evidently not contributed to their intelligibility. If American general dictionaries began the process of moving to IPA, it would not be moving from a system that works to a (possibly better) unknown; the current respelling system does not work, so moving to a system that is more theoretically sound, is more descriptively accurate, and is successfully used elsewhere will in all likelihood be a benefit.

So why do American dictionaries still use respelling systems? Aside from the reasons listed above, tradition appears to be a factor, since systems of this type have been used in dictionaries since Thomas Sheridan's 1789 *Complete Dictionary of the English Language*. Another reason is the cost of educating dictionary users, who expect diacritic respelling because of its long tradition. The “Guide to Pronunciation” for *Webster’s New World College Dictionary,* fourth edition, states it this way:

> In general the PRONUNCIATION SYMBOLS used in *Webster’s New World* are of the type that has been used in American dictionaries for many years. They are familiar symbols that most Americans learned in school. The set of symbols used in this dictionary is unique, however.
>
> The INTERNATIONAL PHONETIC ALPHABET (IPA) has not been used. The IPA has many symbols for showing a wide range of sounds with great exactness. Many British and foreign language dictionaries use the IPA, as do language specialists. Most Americans, however, are not familiar with the IPA, so this dictionary does not use it. (xxii)

Also, the American dictionary market is competitive and hence very conservative, and any dramatic change, such as the use of IPA would be, even if implemented gradually, would be too risky from the publishers’ perspective. Combine this with the perception that dictionary users would be unable to understand IPA and do not want a change and dictionary publishers have good reason to maintain their current, conservative stance. Sidney I. Landau summarizes the situation well:

> The fundamental question we have to ask is: Would we rather represent pronunciation precisely though few understand it or represent it imprecisely to be grasped in a general way by the many? Would we prefer that a few be well informed while many others remain benighted or that a great many people be only slightly misinformed? (“Should We Change?” 119)

So far, American dictionaries have elected to provide partial information to many, and thus have shunned using IPA.

One advance that dictionary writers have made use of, however, is computers. Although the use of computers is recent in the larger context of the history of dictionaries, publishers adopted these new tools for data management and printing quite early. Laurence Urdang, working on *The Random House Dictionary of the English Language, Unabridged*, first reported using computers in dictionary production in the early 1960s. He set up a database that allowed the lexicon to be assembled according to subject, which permitted greater consistency in handling the information. Along with the subject area, the database also contained all other information about the items: “main entry word, pronunciation, definition(s), variant(s), etymology, run-in entry, illustration” (155). Unfortunately, he was unable to produce a print-image directly from this database. He instead had to use technology that was still under development to produce microfilm images.

Besides *Random House*, during the sixties a number of other dictionaries began using computers: *Trésor de la Langue Française*; *Dictionary of the Older Scottish Tongue*; *Dictionary of Old Spanish*; *Dictionary of Old English*; and *American Heritage Dictionary* (Sedelow 97). The last of these was the first to be typeset by computer in 1969 (Logan 352). Also, *Merriam-Webster’s Seventh New Collegiate Dictionary* and *Merriam-Webster’s New Pocket Dictionary* were early dictionaries that were prepared for computational applications (Logan 352).

In all, computers are used in lexicography in a number of ways. The first way that they are used is in production, printing, and logistics. For example, at the *OED*, computers have been used for on-screen editing since the early 1980s, when they first starting the process of computerization (Simpson and Weiner 15; Hultin and Logan; Weiner 2). Also, as John Simpson describes, the Internet has revolutionized communications between editors and consultants:

> Editors benefit from being able to maintain a range of handpicked specialist advisors around the academic world who respond by e-mail to lexicographical enquiries both authoritatively and, in some cases, almost instantaneously. In Murray's day, it could take weeks for a response on a crucial word to arrive in Oxford from, say, Australia or China, although, of course, Murray had a pillar box specially installed outside his house in Oxford to facilitate the flow of correspondence between himself and his own extensive stable of collaborators, both at home and abroad. (Simpson 11)

In the late 1960s and early 1970s, lexicography began a revolution as it incorporated methodologies and tools created for the field of corpus linguistics, which uses large collections of computerized texts to study language. In one sense, lexicographers have always used corpora. Johnson's *Dictionary* was based upon a corpus of probably well over 1 million words, stored on slips of paper. Likewise, the *OED*'s corpus had perhaps over 50 million words (Kennedy 14–15). However, computers provided new powerful tools for dealing with such voluminous amounts of information. The first to make use of computerized corpus tools was the *American Heritage Intermediate (AHI) Corpus*. It was created from publications widely read by 7–15 year old American school children and contained 5.09 million words. It was designed as a citation database for the *American Heritage School Dictionary* (Kennedy 34). "The *AHI Corpus* was one of the first computer-based databases for lexicographical purposes and the resulting dictionary a forerunner of a number of innovative, commercial, corpus-based dictionary projects which were published from the 1980s, including the *Longman Dictionary of Contemporary English* and the *Collins Cobuild English Language Dictionary*" (Kennedy 34–35). The *Collins Cobuild English Language Dictionary* especially represented a landmark, since it was the first to use a so-called "megacorpus" and the first to use it from its inception, rather than simply updating a previously existing dictionary. Also, John Sinclair, its editor, is particularly committed to the field of corpus linguistics, so it is more “theoretically pure” than other dictionaries (Landau, *Dictionaries* 287). The *Collins Cobuild English Language Dictionary* is based upon the COBUILD corpus, later renamed the Bank of English. This is a monitor corpus, one to which material is continually added, begun in 1991 by COBUILD and the University of Birmingham. On the other hand, the *Longman Dictionary of Contemporary English* is based on the Longman/Lancaster corpus, which contains 30 million words and was created through a partnership between Longman Publishers and Lancaster University. These two projects have encouraged fruitful collaborations between universities and commercial interests to construct solid, balanced corpora that can be used both for academic research and for lexicography. Today, using corpora has become the norm. Since *Webster’s Third*, no major English dictionary has not used an electronic database, and most lexicographers base their dictionaries on some corpus, although how much they actually use them is open to debate and American dictionaries—including *Merriam-Webster*, which primarily uses citation files—have been slower to adopt this technology than their British counterparts (Kennedy 15, 91; Simpson 9; Jost, et. al, 16–18).

Originally, lexicographers mainly used key-word in context (KWIC) displays from text concordancing software. More recently, however, they have taken using software made for handling large corpora (Simpson 9). They have also found the Internet—and particularly full-text databases like Lexis/Nexis, JSTOR, and the Middle English Compendium—to be invaluable (Simpson 7).

Corpora can be used for a number of purposes in lexicography. First, a monitor corpus, one to which text samples are constantly added, such as the Bank of English, can be useful in identifying neologisms (Kennedy 91). Another use is in identifying collocates, or words that are regularly used in conjunction with each other. While all dictionaries include these, their selection of them is inconsistent, and they regularly omit important ones (for example, M. Benson mentions *acceptable to*). Using a corpus would help to determine which collocates should be explicitly mentioned, based upon their frequencies, upon their semantic and syntactic importance, or upon irregularities in their use.

Probably the area where corpora have proved themselves the most useful, however, is determining a word's meanings. In the past, determining a word's senses has relied upon instinct and the citation file. Paradoxically, citations still work better for newer or more unusual words, for which readers are more likely to fill out a citation card and which are likely to have fewer senses than more common words. But corpora can provide evidence for how a word is used, what contexts it is used in, and both its denotation and connotation. Thus, while using a corpus does not replace a traditional reading program, it does augment it in useful ways (Simpson 8). Michael Stubbs gives a number of compelling examples of how corpora can provide useful, interesting insights into semantics in his book on this, *Words and Phrases*.

For example, at one point he looks at the occurrences of *undergo* in the COBUILD corpus and builds a lexical profile for it. *MW11* defines *undergo* this way:

> **1** : to submit to : ENDURE
>
> **2** : to go through : EXPERIENCE *undergo a transformation*
>
> **3** *obsolete* : UNDERTAKE
>
> **4** *obsolete* : to partake of

However, Stubbs looks at the top 20 collocates—at the words that occur near the word—for *undergo*, which are listed in Table 2.1. He begins by noting that these collocates seem to present a simple pattern: "people involuntarily *undergo* serious and important events, such as medical procedures" (89). Checking other corpora confirmed this characterization, except that when *undergo* occurs in technical writing, it lacks the negative connotations. Finally, he presents this analysis of *undergo*:

> In summary, the main semantic patterns are simple: (1) In general English, people are forced to undergo unpleasant experiences, especially medical pro-

Collocate	**Frequency**	**Collocate**	**Frequency**
surgery	108	women	31
tests	67	forced	26
treatment	62	further	25
change	53	testing	25
training	43	major	24
test	41	examination	23
medical	40	extensive	31
before	37	heart	20
changes	35	required	19
operation	34	transformation	17

Table 2.1: Top 20 Collocates for *undergo* in COBUILD

> cedures, or tests and (often arduous) training. (2) People and things undergo (usually radical and unpleasant) changes. (3) In scientific and technical English, the word is usually neutral. (92)

He also notes that *undergo* has a typical syntactic usage pattern, which is used in the majority of occurrences, shown in Figure 2.1. Comparing the *MW11* definition with Stubbs' definition makes the advantages of using corpora to study a word's semantics clear: it makes explicit a number of aspects of the word's connotations that are, at best, implicit in the *MW11* definition, but which are common enough to consider including in the word's denotations, for example, the clear negative pattern and the strong medical bias (89–95).

However, like all tools, using corpora does have its disadvantages. For one thing, the software currently used to analyze corpora works best with modern texts. Nonstandard spellings and nonstandard or obsolete syntax can create problems for identifying words and for automatically tagging parts of speech. Also, corpora do not act as a replacement for the writer of definitions. "I also think that the 'art' of lexicography, the work of analyzing passive or modal + *undergo* + adjective + abstract noun

forced to	typical	typical
required to	adjectives	lexical fields
must		
etc.	*further*	*medical procedure*
	extensive	*testing*
	major	*training*
	severe	*change*
	etc.	*a trauma*
		etc.

Figure 2.1: Typical Usage Pattern for *undergo* from Stubbs' Lexical Profile (92)

data and framing definitions which takes place in the lexicographer's head, remains essentially unchanged since Murray's day and is likely to remain fundamentally unchanged for the foreseeable future" (Simpson 10).

Perhaps the deepest, most troubling, and most interesting problems in corpus use are raised by Charles J. Fillmore and B. T. S. Atkins. They examined the definition for *risk* in ten dictionaries and found a number of discrepancies. They had hoped that, by using a corpus to study the word in use, they could write a better, more consistent, and more complete definition for the word. Instead, they say, "we found that the challenge of dealing with the corpus material convinced us that it was impossible to analyse and describe the word within the constraints of the classical dictionary entry design" (350). As a solution, they used frame semantics to provide a better paradigm within which to create a dictionary definition. In it, they drew a schema of the actors and actions involved in the concepts of *risk* and identified the parts of the diagram with the parts of the sentences that used that word. From there, after putting the words and identifications into a database, they could extract patterns of how *risk* is used in different ways to create different meanings. In the end, they determined that this kind of analysis and definition is not currently practical, but that it could be in the not-too-distant future. Right now the time and resources necessary to analyze words in this way, and particularly the space required to print it, are too expensive. However, as more sophisticated computer resources are applied to lexicography and as electronic printing becomes more widespread, using frame semantics to analyze words could become more practical. Their criticism, however, is a decade old at this point, and the analysis of a word's semantics, if not the requirements of publishing a more space-consuming type of definition, is more practical now, as Stubbs demonstrates.

Another way that computers are useful in lexicography is in automating various tests and performing verifications on the entries. For example, a computer can check definitions as they are modified to ensure that they do not contain words that are not in the dictionary (that is, definitions that break the Word Not In principle).

A final way that computers are used in lexicography is to change the nature of dictionaries themselves. As John Algeo noted in remarking on the *OED2*, the real change between the first *OED* and the second edition was not in the merging of the supplemental material, the use of IPA, or any of the other visible changes. The biggest difference was that the new edition was primarily an electronic text. Seen this way, the *OED* can be constantly updated, corrected, and supplemented. "The real Second Edition is, at least in potential, a fluid text, which its editors can make respond as quickly as they wish to new information or new interpretation" (Algeo 139). John Simpson, chief editor of the *OED*, also confirmed this in an interview with *English Today* right after the *OED Online* was opened on March 14, 2002. In this interview, he was asked if there would ever be a third edition printed, and his reply demonstrates that he realizes exactly the watershed the *Online*'s publication was:

> *OED Online* will *be* the Dictionary in future. I am sure it will be the version that most people will consult. A dictionary of perhaps forty volumes will be rather unwieldy, but the present hardback has many fans and the *OED* in traditional book form is by no means out of the question. (Simpson and Weiner 13)

And in another place, he says that the dictionary is moving from "text to dictionary, back to text, and on to bibliography database, picture, sound, graphical analysis, or whatever" (Simpson and Weiner 12). Like all forms of print culture, dictionaries are on the brink of some exciting changes. And although the early stages of their voyage into electronic form will surely reflect their print past, what they might look like further down the road makes for interesting speculation.

In all, whether the early hard-word references or the modern electronic lexicographical databases, dictionaries are an amazing achievement. Although complicated by problems with authority, logistics, and the vagaries of the publishing business, they represent the remarkable product of vast amounts of hard work, drudgery, and scholarship. And the end result is a product that is at once misunderstood by many of its users, but extremely useful, containing information that is unavailable anywhere else. This is the tradition into which this product, Schwa, steps, trying to lighten the work load somewhat in working with pronunciations, which have been, as described above, one of the most neglected parts of the dictionary.

Chapter 3

Technical Issues

Some people, when confronted with a problem, think, "I know, I'll use a computer." Now they have two problems.

— paraphrasing Jamie Zawinksi, "marginal hacks"

Aside from the theoretical issues concerning lexicography and phonetics, planning an application for processing lexicographical pronunciations also raises technical problems. These include how the IPA phonetic information is encoded, how to represent the dictionary entries, how to store the information, and how to process phonetic data. Looking at how these issues have been handled in the past, as well as what options are available today, allows us to arrive at the best solution for the current problem.

The first technical issue to consider is how to represent IPA pronunciations in a form the computer understands. Theoretically, this is no different than asking a computer to work with the English alphabet, the Hebrew alphabet, or Chinese ideographs. To work with any script, the computer needs four components: a character set defining correspondences between character codes and the characters in that alphabet, a font defining how the characters in the character set should be displayed on the monitor or printed (the glyphs in the font), an input method for that character set, and operating system or application software support (Harold 181).

The first component, the character set, is how computers deal with textual data. Internally, computers represent all information as positive integers within a specified range (often 0–65,535). Computers maintain a table defining correspondences between these numeric codes and letters, punctuation, digits, and other characters. This set of correspondences is referred to as the character set. The characters in the character set represent all the characters and only the characters that the computer can deal with and the characters that applications expect to be available. Thus, to use some characters on the computer, for instance the IPA, there needs to be a character set defined for it.

Numbers for character codes are usually given in base sixteen, or hexadecimal. Computers store these numbers using base two, or binary. A single binary digit is a bit, and eight binary digits are a byte. Just as “round” numbers in decimal (base ten) are powers of ten, so round numbers in base two are powers of two. Thus, the numbers that computers deal with most naturally and are used to specify ranges of numbers and characters are often powers of two. The “specified range” mentioned in the preceding paragraph is a good example. 65,536 is 2^{16}. Moreover, sixteen is used here because it also is a power of two (2^4). While round numbers in binary rarely correspond to round numbers in decimal, they always correspond to round numbers in a base that is itself a power of two. For this reason, base eight (2^3, called octal) and base sixteen (2^4, called hexadecimal) are often used to represent numbers in computers. Hexadecimal is particularly efficient for this, because each hexadecimal digit represents four binary digits, so two hexadecimal digits represent one byte. For an example of how some round binary numbers fail to correspond to round numbers in decimal, but do correspond to round numbers in hexadecimal, see Table 3.1. Hexadecimal numbers are often written with a “0x” preceding them, to differentiate them from decimal numbers. Of course, hexadecimal requires six more digits than decimal does, and these are supplied using the first six letters of the alphabet, so in hexadecimal, counting starts with 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, which corresponds to decimal 0–16.

Converting hexadecimal characters to decimal is fairly simple. Starting from the right, each digit is multiplied by 16 raised to the count of the digit, from the right, minus one. The

Power of Two	**Binary**	**Decimal**	**Hexadecimal**
2^2	100	4	0x4
2^5	100000	32	0x20
2^8	100000000	256	0x100
2^{10}	10000000000	1,024	0x400
2^{16}	10000000000000000	65,536	0x10000

Table 3.1: Numbers in Binary, Decimal, and Hexadecimal

$$\begin{aligned}
\mathrm{0x8F1C} &= (8 \times 16^3) + (\mathrm{F} \times 16^2) + (1 \times 16^1) + (\mathrm{C} \times 16^0) \\
&= (8 \times 16^3) + (15 \times 16^2) + (1 \times 16^1) + (12 \times 16^0) \\
&= (8 \times 4096) + (15 \times 256) + (1 \times 16) + (12 \times 1) \\
&= 32768 + 3840 + 16 + 12 \\
&= 36636
\end{aligned}$$

Figure 3.1: Converting Hexadecimal 0x8F1C

digits A–F are replaced by their decimal equivalences: 10–15. These are added together to get the total. As with most mathematical concepts, this process is easier to illustrate than to explain. Figure 3.1 walks through converting 0x8F1C. First, the equation is set up according to the process above: each digit is multiplied by sixteen raised to the number of the digit, counting from the right, minus one. Next, the digits F and C are replaced by their decimal equivalences, 15 and 12. From this point on, the equation is simplified using standard algebraic processes. While this is not difficult, fortunately most scientific calculators, including the calculator applications included with most computer operating systems, will automatically convert between hexadecimal and decimal.

In summary, base sixteen is used for character codes. For example, in most character codes, *a* is character code 97 (decimal), which is 0x61 (hexadecimal; $6 \times 16^1 + 1 \times 16^0$). For the rest of this discussion, hexadecimal will be used for all character codes, while decimal will be used to specify ranges of character codes.

The earliest widely used character set is the American Standard Code for Information Interchange (ASCII). ASCII defined correspondences for thirty-two non-printing codes used to control the computer, printer, or teletype, and ninety-six codes for digits, punctuation, and the American English alphabet. It defined these in the range 0–127 (decimal). While ASCII worked very well for American English, it did not work for Spanish, French, or German, which used additional characters, much less for Hebrew, Russian, or Japanese, with their entirely different character sets.

To address these deficiencies, a babel of other encodings arose. In 1984, the Macintosh computer introduced the MacRoman encoding (Harold 193). This extended the ASCII character set by defining a number of other characters in the range 128–255. These were mainly accented Latin letters, Latin digraphs, and Greek letters used for mathematical symbols.

Of course, MacRoman did not include enough characters to represent all the European languages, much less the languages for the entire world. To handle European and Arabic alphabets, the International Standards Organization (ISO) defined fifteen other character sets. All of these were supersets of ASCII: these character sets used the characters ASCII defined in the range 0–127, and they extended ASCII by defining other characters in the range 128–255. The most common of these extended character sets was ISO 8859-1, also known as Latin-1. It defined many of the same characters as MacRoman, but associated the characters with different codes. The other character sets (ISO 8859-2 through ISO 8859-15) defined slightly different characters in the 128–255 range. For example, ISO 8859-3 defined the ASCII set plus characters for Esperanto, German, Maltese, and Galician; ISO 8859-6 defined the ASCII set plus Arabic characters. These character sets could not be used together. Thus, if ISO 8859-3 was used, the document could contain English and German; if ISO 8859-6 was used, the document could contain English and Arabic. However, there was no encoding that allowed German and Arabic to be used together.

When Microsoft released Windows, it used a new character set, which modified the ISO 8859-1 character set by adding a few characters for codes that ISO 8859-1 had left undefined. This encoding is known as Cp1252 or ANSI (although it was never standardized by the American National Standards Institute).

In the general melee, a number of encodings sprang up to address the needs of those who used Cyrillic and Asian languages. Particularly for Asian languages, these character sets were still insufficient since they were limited to 256 characters. Also, as would be expected, none of these character sets defined phonetic symbols. To make up for this oversight, phoneticians and software developers invented a number of systems, which generally fall into two categories: those that redefine all or part of the character set and include a font to provide glyphs for that set and those that use standard printable characters to represent phonetics.

One system of redefinitions and fonts was developed by William A. Kretzschmar, Jr., in 1984 for representing data from the Linguistic Atlas of the Middle and South Atlantic States ("Phonetic Output"). Unlike other software it only redefined rarely used characters, not the regular alphabet, so this font could be used in word processing software to represent both phonetics and standard alphabetic characters. Also, instead of having a separate character for each letter-diacritic combination, characters would be combined when printing. For example, "ᵻ" was a combination of "ɪ" and "- ." This system used Borland's Superkey program to allow `CTRL-` and `ALT-` key combinations to be defined to input common sequences of phonetic characters (Kretzschmar and Konopka).

In 1989, Rebecca Dauer describes Better Letter Setter, another program typical of this kind. Better Letter Setter illustrates some of the drawbacks of these systems. First, in this instance at least, the on-screen font did not match the print font. For instance, "Z" would show on the screen where "ʒ" would print (42). Also, to enter many characters, users had to hold down the `ALT` key while typing in a three-digit number on the numeric keypad (43).

A more elaborate system than this, complete with its own word processor, as well as the ability to integrate with other word processors, is described by Jassem and Łobacz. This software provided three fonts, all of which redefined the standard alphabet as well as other codes. One font was for encoding French, Italian, Spanish, and Swedish phonetics; one for English and Polish; and one for German and Russian (18). Some characters were redefined consistently across the fonts, while others were redefined differently in each case. For example, character code 0x70 ("p" in ASCII) was "p" in all three, and 0x35 (ASCII "5") was "ʤ." On the other hand, 0x56 (ASCII "V") was "ɣ," "ʌ," or "y̜"

A different approach is to use regular ASCII and ISO 8859-1 glyphs as phonetic characters. This is useful in some situations, either where the use of phonetics is limited, such as e-mail, or where it can be easily extended, such as in LaTeX. For example, the tipa LaTeX package interprets forty-one standard ASCII characters as phonetics. This small set of characters is supplemented by a considerable number of longer macros that define other characters, as well as accents and combinations of characters and accents. Phonetics are differentiated from regular text with the `\textipa{...}` macro. For example, the LaTeX command "`\tipa{f@"nEtIks}`" is printed "fəˈnɛtɪks." On the other hand, a pronunciation that uses macros to display other characters, such as that for "jab," would look like "`\textipa{\textdyoghlig{\ae}b}`" and would print "ʤæb."

To fix this mess of character sets, a number of companies and organizations formed the Unicode Consortium, which published *The Unicode Standard 1.0* in 1992. Unicode expands upon ASCII, which uses 7 bits, and the ISO encodings, which use 8, by using 16 bits (or 32 bits in Unicode 4.0.0). This allows for 65,536 character codes (or 4,294,967,296 for Unicode 4). Each character in Unicode is defined as a character code, usually written `U+0000` or `U+00000000`, where *0000* or *00000000* is the code as a hexadecimal number. Each of these digits represents four of the bits used by the computer to represent the character in memory. Leading zeros, such as the first two zeros in "U+0061," are simply placeholders. Each character is also assigned a name, usually written in uppercase. For example, an *a* character has the code number of 97 in decimal, so its Unicode number is written “U+0061,” and its Unicode name is “LATIN SMALL LETTER A.”

To facilitate backward compatibility, the first 128 codes in the Unicode character set are the same as the ISO 8859-1 character set. After that, the characters are divided into blocks, with each block containing the characters for one script (although a few alphabets have characters in more than one block, such as French, which has characters in the Basic Latin and the Latin-1 Supplement blocks). IPA extensions are defined in the range from 592–687.

At a basic level, Unicode[1] only defines character sets, that is, correspondences between numeric character codes and the characters themselves. On another level, it also specifies how Unicode data can be encoded in a file. There are a number of considerations in doing this. First, many operating systems still expect text files to be encoded using 8-bit characters. This means that how a Unicode character’s 32 bits are represented in the file needs to be specified exactly. Second, there are already many, many files that use either the ASCII or ANSI character sets. Since the Unicode character set is already a superset of these, being able to treat these legacy files as Unicode would greatly facilitate backward compatibility.

The simplest way to encode Unicode strings in a file is called UTF-32 in the Unicode standard. This simply encodes the 32 bits in four consecutive bytes. For example, in Table 3.2, U+0061, *a*, is encoded as “`00000061`,” and U+FFEE, ○, is encoded as “`0000FFEE`.” Reading a Unicode character in this encoding is trivial, but it has two major drawbacks. First, it is not backward compatible with earlier encodings like ASCII or ANSI. Second, for English texts, it is very inefficient. An ASCII file recoded as UTF-32 would be four times as large without adding any informational content.

[1]This discussion only describes the version of Unicode current at the time this is being written, 4.0.0.

a
U+0061
LATIN SMALL LETTER A

ASCII	61			
UTF-8	61			
UTF-16	61	00		
UTF-32	61	00	00	00

æ
U+00E6
LATIN SMALL LETTER AE

ASCII	N/A			
UTF-8	C3	A6		
UTF-16	E6	00		
UTF-32	E6	00	00	00

ə
U+0259
LATIN SMALL LETTER SCHWA

ASCII	N/A			
UTF-8	C9	99		
UTF-16	59	02		
UTF-32	59	02	00	00

○
U+FFEE
HALFWIDTH WHITE CIRCLE

ASCII	N/A			
UTF-8	EF	BF	AE	
UTF-16	EE	FF		
UTF-32	EE	FF	00	00

Table 3.2: File Encodings of Unicode Characters

The next way to encode Unicode strings is called UTF-16. In it, the characters in the range U+0000–U+FFFF are encoded directly as 2 consecutive bytes. For example, U+0061, *a*, is encoded as “`0061`,” and U+0259, ə, as “`0259`.” This encoding shares both of the drawbacks of UTF-32: no backward compatibility and inefficiency in encoding English texts.

The final way to encode Unicode strings is UTF-8. It encodes the characters in the ASCII range, U+0000–U+007F as a standard 8-bit character, just as it would be in ASCII. Thus, UTF-8 maintains backward compatibility with all ASCII and many ISO 8859-1 files: any ASCII file is, by definition, a UTF-8 file also. For characters above U+007F, flags are set at the bit level that indicate what range the code is in and how that code is distributed over the next two, three, or four bytes. Table 3.3, taken from *The Unicode Standard*, shows the details of how those bits are distributed over the bytes that represent the character code in UTF-8 encoded files. For example, U+0061, *a*, is defined in the ASCII

Scalar Value	**1st Byte**	**2nd Byte**	**3rd Byte**	**4th Byte**
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Table 3.3: UTF-8 Bit Distribution (*Unicode* 77)

range, so it is represented in the file as one byte. U+00E6, *æ*, is above this range, so it is represented by two bytes. U+0259, ə, works similarly. On the other hand, U+FFEE, ○, which is at the upper range of the characters defined by Unicode within 16 bits, has to use three bytes to represent the character. Thus, as well as maintaining backward compatibility, UTF-8 also is efficient for encoding English texts. However, for texts that use many characters from the upper extent of the Unicode Standard, it can be less efficient than UTF-16.

Note that as far as *The Unicode Standard* is concerned, all of these are legitimate representations of Unicode characters. It explicitly says, "It is important not to fall into the trap of trying to distinguish 'UTF-8 *versus* Unicode,' for example. UTF-8, UTF-16, and UTF-32 are *all* equally valid and conformant ways of implementing the encoded characters of the Unicode Standard" (28).

The next thing required to use Unicode is a font that defines glyphs for the current character set. This font does not have to define *all* the characters in the set, but obviously it does need to define those you intend to use. For Unicode, the number of fonts that define glyphs is steadily increasing. Microsoft Office, for example, includes a number of fonts that define glyphs for subsets of Unicode, generally to display the characters for a given language. It also includes "Arial Unicode MS," which defines fonts for a large subset of Unicode. As one would expect, this font is very large, about 22 megabytes in size. However, it does define the IPA extensions block of Unicode, so it is a useful font to fall back upon when no other is available. Other fonts in Office, such as “Lucida Sans Unicode,” also define many IPA characters. Also, some Unicode fonts have been designed just for use with IPA, such as Herman Miller’s Thryomanes font.

The next thing necessary to work in a given character set is a way to input the characters. For standard English text the QWERTY keyboard solves this problem well. Also, solutions have been invented for working with languages based upon the Latin alphabet that also include some accented characters. For example, French keyboards have some keys changed to added accents, as well as an `AltGr` key (“Alternate Graphic”), which allows keys to input yet a third glyph (Harold 182). Alternatively, languages such as Hebrew, which do not use the Latin alphabet, but which have only a fairly small number of characters, can use the standard QWERTY keyboard with an extra key that operates like a `Shift Lock`. When the user presses that key, the keyboard changes state, or shifts into “Hebrew” mode, and registers key presses as Hebrew characters, until the user presses that extra key again to change the state of the keyboard and returns to “English” mode. On the other hand, languages like Japanese or Chinese cannot really use either of these methods because of their large number of characters. Instead, they tend to use a combination of keyboard keys and software that accepts further input. The details of these systems tend to be very dependent on hardware, operating system, and application software (Harold 183–84).

Systems for inputting phonetics have relied upon a number of methods, many of which were described in more detail above and are only summarized here. First, the Linguistic Atlas Programs used Borland’s Superkey application to provide `CTRL`- and `ALT`-key combinations to input either single characters or strings of characters. For example, `ALT-a` would input “æ,” `ALT-e` “ə,” `ALT-z` “ʒ” and `CTRL-1` “ᵻ.” Better Letter Setter used a more awkward system, in which what appeared on the screen would print differently on hard copy. It used the standard keyboard, so `Z` would appear on the screen as “Z,” but “ʒ” would print. Others, Jassem and Łobacz, for example, refined this by providing a screen font. However, their system was complicated by three different fonts, with the same key producing different characters, depending upon the current font.

The final piece of the puzzle for handling IPA transcriptions by computer is the level of operating system and application software support. Many of the original characters sets were dependent upon the operating system. For example, MacRoman was only supported on Apple Macintosh computers. By default, Microsoft Windows used ANSI, although language packs could be installed that allowed it to use a variety of other character sets. And proprietary systems, such as those used for phonetics, relied on software applications for their support.

Operating systems have been slow to include Unicode support, although momentum is beginning to swing in this direction, and recently it has even become quite prevalent. It is the default character set for Microsoft Windows NT/2000/XP, Apple Macintosh OS X, and Sun Solaris. It is also easily used on most other platforms. Many applications, including Microsoft Office, also support Unicode, regardless of which operating system they run on.

For programmers wanting to write applications that use Unicode, the situation is necessarily more complicated. The first requirement for writing a program that supports Unicode is either a programming language that supports it or that has a library available for it. Most programming languages now have Unicode support at some level. Currently, Java, C#, and most scripting languages have Unicode support built-in, and even older languages like C have support for it through the `wchar_t` (wide character) type.

An aspect of program development that is less felicitous for Unicode is graphical user-interface (GUI) libraries. While a few libraries have good Unicode support, for others it is deficient or lacking altogether. Part of the reason for this is the lack of consistent Unicode support in the underlying operating systems. For example, although Windows NT, 2000, and XP all use Unicode natively, Windows 95, 98, and ME can only use it through special libraries. Hence, even Visual Basic 6's user interface designer does not use Unicode controls by default when a program is developed on a version of Windows that supports Unicode. Instead, developers have to enable Unicode controls intentionally, since any application built using Unicode will not work natively on Windows 95, 98, or ME. Third party GUI toolkits for Windows or for other operating systems also have varying Unicode support. Part of the reason for this is that, since they cannot rely on Unicode support in the underlying operating system, they must implement it from scratch. Nevertheless, a number of third-party toolkits have some degree of Unicode support. wxWindows, for example, provides a consistent programming interface on any platform for the underlying native GUI library. If the underlying library handles Unicode, wxWindows can also. On the other hand, Tcl/Tk, for example, has very good Unicode support regardless of the platform. Not only can the elements of the user interface display Unicode, but if the current font does not define the characters it needs to display, the toolkit tries to identify one that does. How well it accomplishes this varies, but this is a feature that no other toolkit provides, as far as I know. Another library that handles Unicode is the Qt library. Under Microsoft Windows, there are still licensing issues with this library, and these make it less attractive than other libraries for open source projects, but its Unicode handling is still excellent. Of course, Java's AWT and Swing libraries also have good Unicode support, having been initially designed for this.

Once the issues of how to use IPA on computers have been considered, however, the data model for dictionary entries raises more problems. This does not necessarily mean considering how the entries are stored, which will be examined later, but instead what overall structure and paradigm is best for representing dictionary entries.

Entity/relational (ER) data models are used to describe collections of entities—anything that can be distinctly identified—and relationships between one or more entities. For example, an entry is an entity, as are pronunciations and word senses. Each entry can have several pronunciations associated with it, but each pronunciation can only be

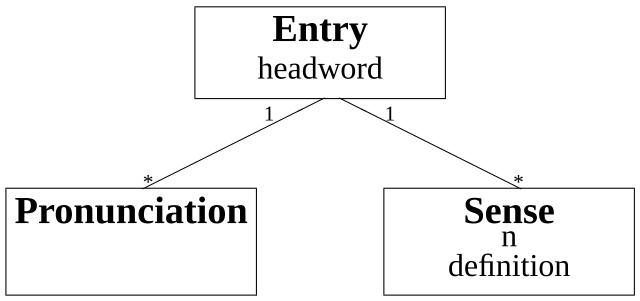


Figure 3.2: Entities and Relations in a Simple Dictionary Entry

associated with one entry. This relationship is described as being *one-to-many*: one entry is associated with many pronunciations. Other kinds of relationships are *one-to-one* (one entry is associated with one headword, for example) and *many-to-many* (each entry may be associated with many subject domains, and each subject domain may be associated with many entries). Figure 3.2 contains a schematic illustrating the entry, pronunciation, and sense entities and the relationships between them. The *1* and * on the lines expressing the relationships indicate that the relationship is one-to-many, and which entity in the relationship is singular (labeled *1*) and which is plural (labeled *). The *entry* entity also has a specified characteristic, called an *attribute*. In this example, that attribute is the entry's headword. Entities can have as many attributes as necessary, such as the *sense* entity, which has an attribute, *n*, containing the number of the definition, and another, *definition*, which contains the text of the definition. Typically, attributes' values are atomic—that is, they cannot be broken into smaller meaningful units—and they have a one-to-one relationship with the entities they describe.

ER models tend to be static and well-defined from the outset. They do not work well if there is a lot of variability in the structure of the information, for example if senses may

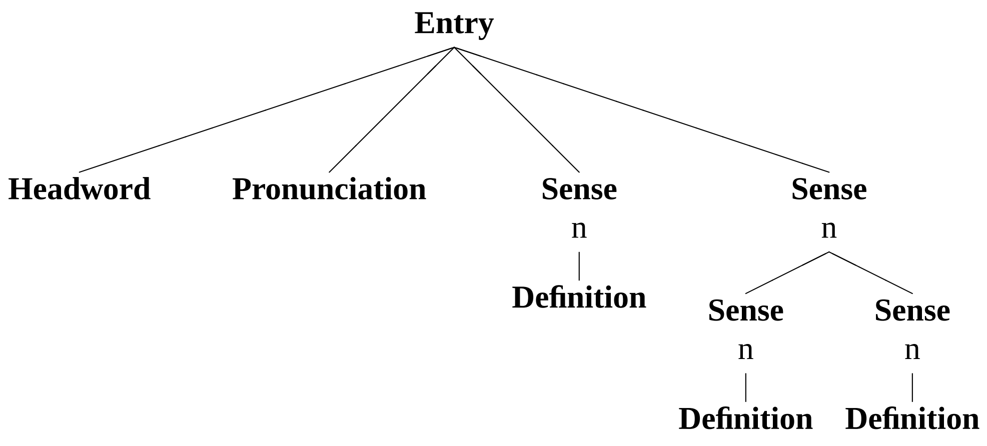


Figure 3.3: A Tree Model of a Simple Dictionary Entry

themselves contain other senses. This seems to be a major obstacle in modelling dictionary entries with an ER model, since entries contain highly flexible structures. Senses can occur either directly inside the entry or contained by other senses; an entry can contain multiple senses tied to different parts of speech. The possibilities are (almost) endless.

However, there is another way to model the data. Instead of conceiving of the parts of entries as being discrete entities with relationships between them, they can be viewed as a set of entities that directly contain other entities in a flexible manner. So, for example, an *entry* can contain one or more *pronunciation* entities and one or more *sense* entities. This gives the entry a tree-like structure. One possible way to express a dictionary entry as a tree model is shown in Figure 3.3. Entities can also have attributes, which again contain atomic data tied directly to the entity. This is called a tree model, because there is one root entity that contains all the other entities, which descend from the root like branches and leaves.

Tree models appear to be a more appropriate way to represent dictionary entries. Although restrictions can be placed upon what an entity can contain, they do not have to be as formal and static as the relationships tend to be in ER models. This way, a sense can easily contain either definitions or more senses.

In practice, ER data models are stored using a relational database, and tree structures using Extensible Markup Language (XML). However, either data model can be stored using other formats or in a hybrid combination of formats. For example, senses may be stored in a database table, and each sense may contain XML markup delimiting its internal structure. Because of this flexibility, it is worthwhile considering a number of storage technologies to determine which will provide the best combination of performance, flexibility, and robustness.

One simple way to store information involves making use of the facilities various programming languages have to serialize data, or to write them to or restore them from a file or string. For instance, Java uses the `Serializable` interface, Python uses the `pickle` module, and other languages have other methods. The main advantage of serializing objects is that it is extremely easy and tends to be fairly fast. Typically, serialization involves little more than calling a function with the object to persist and a file open for writing. For example, Figure 3.4 demonstrates a short program in Python that creates an object and serializes it to a file. While serialization is easy to use, its disadvantages are great. First, it ties the data to one programming language. Second, all the data must be held in memory at one time. For a dictionary with hundreds of thousands of entries, this can be prohibitive. Third, serialization is not safe: if the program or computer crashes before the data is saved, any changes made since the last save are lost. Moreover, if the program or computer crashes while the data is being saved, the data file will be corrupted and all the data, past and present, will be lost.

A text markup language, such as XML (Bray), represents an improvement over object serialization. These markup languages are especially well suited for working with tree

```
# import the serialization library
import pickle

# create the object
obj = object()

# open the file
f = file("output.file.name", "wb")

# serialize the object
pickle.dump(obj, f)

# close the file
f.close()
```

Figure 3.4: Example of Serialization in Python

models, but can be used for handling ER models also, albeit not as naturally. Moreover, dictionary entries have been stored using tree models expressed through markup languages for some time. When the *Oxford English Dictionary* decided to computerize in 1983, they decided to encode the data using IBM's General Markup Language (GML), an ancestor of Standard Generalized Markup Language (SGML) and XML.

Another attempt to encode dictionaries using a markup-based tree model was the Text Encoding Initiative's (TEI's) dictionary subset. In 1995, Ide and Véronis discussed the problems of encoding dictionaries, "among the most complex text types treated in the TEI" (167). The challenges in encoding dictionaries come first from trying to strike a balance in adequately describing the often disparate structures of different dictionaries and second from capturing the visual presentation of the information on the page as well as the information content, which are often radically different.

Following the lead of the *OED* or any other commercial dictionary publisher in this matter is difficult, since they provide little information about the internal workings of their systems. However, the TEI's dictionary subset provides a good example of encoding dictionaries and of the problems encountered in doing so (Sperberg-McQueen and Burnard). First, the structure within and across dictionaries varies so widely that it is difficult to determine a consistent, structured description of the entries or to set guidelines for encoding them. Second, like any printed documents, dictionaries exist on two levels: the typographical and the semantic. Some users may be primarily interested in the first, others in the second, and still others in both. Thus, the challenge comes in trying to include markup that is able to capture both typographical and semantic information, while still striking a balance between having the markup flexible enough to be useful, yet strict enough to be descriptive. In the TEI, the first problem is addressed by having entries contained in one of two tags: the <`entryFree`> tag can contain any elements in any order; the <`entry`> tag provides more structure. The TEI recommends using <`entry`> whenever possible and to fall back on the <`entryFree`> tag only when absolutely necessary. The TEI addresses the second problem by providing a set of guidelines to follow according to whether the encoder is interested in capturing the dictionary's typographic or semantic information. For example, capturing the typographic information in the dictionary can be accomplished using the <`rendition`> tag in the header, and the `rend` attribute available on every tag. Thus, the TEI is capable of encoding either the typographical or semantic models of the data or a combination of them.

Within the TEI's <`entry`> tag, there are multiple levels of structure. Under the <`entry`> tag, the <`hom`> (homograph) tag groups elements related to one part-of-speech homograph in the <`entry`>. The <`sense`> tag groups elements related to one sense of the entry or homograph. Moreover, these elements nest according to a set of guidelines:

> As may be seen, the content model for <`entry`> specifies that entries do not nest, that homographs nest within entries, and that senses nest within entries, homographs, or senses, and may be nested to any depth to reflect the embedding of sub-senses. Any of the top-level constituents (<`def`>, <`usg`>, <`form`>, etc.) can appear at any level (i.e., within entries, homographs, or senses). (Sperberg-McQueen and Burnard)

These top-level elements are <`form`>, <`gramGrp`> (grammatical group), <`def`> (definition), <`trans`> (translation), <`eg`> (example), <`usg`> (usage information), <`xr`> (cross reference), <`etym`> (etymology), <`re`> (reference), and <`note`>. These elements all have their own content models. Finally, many of the tags that allow textual content also allow all of the other tags from the core and base tag sets: for example, <`emph`> and others. In all, the TEI provides a rich and thorough set of tags to express the complex content model represented by dictionaries and dictionary entries. An example of this is provided in Figure 3.5, which gives the definition for *phonetics* given by *MW11*, with its semantic content marked up. This shows nesting <`sense`> tags (for senses "2a" and "2b"), as well as the general encoding for a dictionary entry.

Of course, XML does have both advantages and disadvantages. On the positive side, it has become a widely used standard, which many applications and software libraries can handle intelligently. Thus, the data is not closely tied to any particular application or programming language, such as happens when using programming-language-provided persistence. Second, this standardization also provides many opportunities to share data and interoperate with other users. However, XML does have a number of the same problems as object serialization. Typically, all the data needs to be held in memory in some format at one time. Thus, it is unsafe for the same reasons that object serialization is.

As an alternative to this, the first kind of database technology to consider is relational database systems. Typically, when the word *database* is used, this is the kind of technology

```
<entry>
  <form>
    <orth> phonetics </orth>
    <hyph> pho|net|ics </hyph>
    <pron> f&#x0259;-&#x02c8;ne-tiks </pron>
  </form>
  <gramGrp>
    <pos> n </pos>
    <number> pl but sing in constr </number>
  </gramGrp>
  <etym>
    <date> 1836 </date>
  </etym>
  <sense n="1">
    <def>
      the system of speech sounds of a language
      or group of languages
    </def>
  </sense>
  <sense n="2">
    <sense n="a">
      <def>
        the study and systematic classification
        of the sounds made in spoken utterance
      </def>
    </sense>
    <sense n="b">
      <def>
        the practical application of this science
        to language study
      </def>
    </sense>
  </sense>
</entry>
```

Figure 3.5: *MW11* Entry for *Phonetics* Marked Up Using the TEI

meant. Relational databases implement an ER model. Entities, as well as many-to-many relationships, are expressed using tables. Each entity must have a way of uniquely identifying itself. Relationships are expressed by one entity referring to the identifier of another. For example, Figure 3.6 shows some entries from *Merriam-Webster's Collegiate Dictionary*, 11th edition (*MW11*), stored in a relational database structured according to Figure 3.2. Each table has an attribute, represented by the column *id*, which maintains a unique identifier for each entity (row) in the table. The *Pronunciations* and *Senses* tables each contain the column *entry_id*, which holds an identifier for a row in the *Entries* table. This expresses the one-to-many relationship between entities in the *Entries* table and those in the *Pronunciations* and *Senses* tables, since more than one row in those tables can reference a single row of the *Entries* table.

Relational databases can also incorporate hierarchical, tree-structured data. A simple table can hold the nodes of the tree, using a field to identify the kind of entity. Another table can specify the relationships between nodes. However, compared to more direct ways of encoding such data, or the ease with which relational databases hold ER-structured data, such a representation is awkward. Another way to use relational databases to hold such data is to encode the data as XML and to store that in the table. Other data, for instance identifiers, can be extracted from the XML data and stored in separate fields. To do so does add a level of complication, however. Although the majority of the data can be kept in a more natural storage form, the extracted information kept in separate columns cannot be changed separately. For example, if there is a pronunciation column that takes its value from the pronunciation given in the XML data, changing it without changing the corresponding XML would invalidate the data in the database.

This burden can be ameliorated by using some XML-enabled or -aware relational databases, but this is not necessarily the panacea it might seem. Since XML has become a buzzword, many databases claim to be XML-enabled, but exactly what this means varies. Most of these databases can export and import their data using XML. Some can query

Entries

id	headword
1	lexicography
2	orthoepy
3	phonetics

Pronunciations

id	pron	entry_id
1	ˌlɛksəˈkɑgrəfi	1
2	ˌɔrθəˈwɛpi	2
3	ɔrˌθoʊəpi	2
4	fəˈnɛtɪks	3

Senses

id	n	subn	def	entry_id
1	1		the editing or making of a dictionary	1
2	2		the principles and practices of dictionary making	1
3	1		the customary pronunciation of a language	2
4	2		the study of the pronunciation of a language	2
5	1		the system of speech sounds of a language or group of languages	3
6	2	a	the study and systematic classification of the sounds made in spoken utterance	3
7	2	b	the practical application of this science to language study	3

Figure 3.6: Dictionary Entries Stored in a Relational Database (Entries from *MW11*)

using XPath, an XML-based query language. Some accept XML input for columns. A few store their data in an XML format. Thus, although using an XML database *may* be helpful, it does not necessarily ease the burden that storing XML in a relational database places upon the programmer.

A final option is to use a simpler database system. Libraries such as the Berkeley DB provide a simple string-to-string mapping (*Sleepycat Software*). If sufficient, this structure can be used directly, or if a more complicated system is needed, this can form the basis for a more full-featured database. For example, MySQL uses Berkeley DB for its low-level tables. One way to use a Berkeley DB would be to store the entries as XML and key each entry by its headword. This maintains the natural, hierarchical structure of the entries while providing many of the benefits of relational databases, including more efficient disk use, speed, and incremental saves for the data so less information is lost in the event of a crash.

In the end, a consideration of the technical issues in working with lexicography and phonetics reveals that computers have been used in lexicography for almost half a century. However, computer technology has advanced continuously, and lexicography, being tied to legacy systems, has been slow to take advantage of those advances. But by making use of the latest technologies in character encoding, data models, and other areas, working with dictionary data and pronunciations can be greatly facilitated.

Bearing in mind the lexicographical and technical issues considered in this chapter and the previous one, we can turn our attention to designing a specific application that makes use of these theories and technologies to facilitate managing and editing databases of dictionary pronunciations.

CHAPTER 4

SCHWA

Anyone who can’t pronounce schwa can’t pronounce anything.

— *Metalleus*

Of course, the technologies discussed in the last chapter are useless without an application that applies them to a particular domain in a user-friendly manner. I have brought these technologies together in an application named Schwa, a program to manage and edit large databases of dictionary pronunciations.

In designing this product, I had a number of goals in mind. First, the application should be easy for the end user. This includes a standard installation procedure, intuitive user interface, and good error-handling procedures. This does *not* necessarily imply that the program should have no learning curve, since users—particularly experts using an application pertaining to their field—are often willing to spend some time learning to use a program if the pay-off is sufficient in terms of his or her added ability to accomplish tasks. In other words, if the application provides enough power, users are willing to expend a proportional amount of energy to learn to use that application.

Second, in its use and the work-flow it embodies, the application should encourage good practices in the way that users write and manage dictionary pronunciations. Although it may be possible to write pronunciations that are based upon poor practice, the “natural” way to use the program should encourage good phonological practices, such as those described in Chapter 2.

Third, the application should be based upon good decisions in the technology it incorporates. This means that the latest, bleeding-edge technology is often inappropriate because of the instability of both the standards that define it (if, indeed, there are any) and the libraries that implement it. Often, a better option is to use a more mature and stable technology. The resulting application is more solid in its day-to-day use and often more long-lived in its ability to operate with future technology. Using mature technologies also means that the program's design and the structures that go into creating it should represent good programming practices.

Fourth, the program should be easily extensible by programmers and by more technically adept users. They should be able to build upon the framework it provides to define and perform further tasks on the program's databases and the dictionary entries and pronunciations they contain. Also, further extending the program to import data from and export data to more data sources should be a well defined process and not unduly difficult.

Together with the considerations covered in Chapters 2 and 3, these goals provide guidelines for this application, and the rest of this dissertation describes the results of these guidelines, Schwa. First, this chapter describes the data and databases that Schwa manages. Second, it describes the program from a programmer's perspective: the programming language objects and structures that make up the program code. Third, it describes the program from a user's perspective: the windows that comprise the program's user interface and the processes they make available to the user. Finally, it has a number of examples and suggestions for using the program, both to illustrate daily use and to show how to use Schwa to solve certain problems the writer of lexicographical pronunciations may encounter.

4.1 Data

As Chapter 3 points out, the semantic structure of dictionary entries is both complex and varied. Because of this, the entries are best represented using XML. At the most basic level, XML is simply a way to mark information so a computer can easily process it. Unlike many other ways of making information accessible to a computer, XML is also fairly readable by humans. The way that XML marks up information is similar to how SGML and HTML have done it, and for good reason: XML is descended from SGML. However, in many ways XML is supposed to be simpler than either SGML or HTML.

XML models all data using a hierarchical, tree-like structure. Each document has a root, which is simply an object that contains all other objects in the document. The objects in an XML document are called *elements*, and each element is marked off using *tags*. There are three kinds of tags: an *open tag*, which is always paired with an *end tag*, and an *empty tag*, which is conceptually like having an open tag immediately followed by an end tag. Tags of all kinds begin with a less than sign and end with a greater than sign. Open tags simply have the *tag or element name*: `<tag-name>`. End tags come after all of an element's content, and have a forward slash immediately before the tag's name: `</tag-name>`. Empty tags are like an open tag, except they have the forward slash immediately after the tag's name: `<tag-name/>`.

Elements also have attributes, which are pairs of names and simple values that are attached to the open tag. Attributes are expressed using the name, an equals sign, and the value in quotes: `<tag-name attribute="value">`. Empty tags can also have attributes: `<tag-name attribute="value"/>`.

Elements also have content, called the element's *children*. The children can be a list of complete elements, textual information, or a combination of the two. The textual information cannot contain any less-than sign, greater-than sign, or ampersand. To include one of these characters, an *entity* is used. There are two kinds of entities. The first is called a

Named Entity	**Character**
`&lt;`	`<`
`&gt;`	`>`
`&apos;`	`'`
`&quote;`	`"`
`&amp;`	`&`

Table 4.1: Default XML Named Entities

character reference entity. This begins with a “&#” and ends with a “;”. Between these two is a Unicode character code. This code can be in decimal or, if preceded by “x”, in hexadecimal. For example, the character code for ə is decimal 601, hexadecimal 259. Thus, one way to include a schwa character in an XML document is to use a character reference entity, which would look like “`&#601;`” or “`&#x259;`”. Since the Unicode character code is written U+0259, often the entity will be written “`&#x0259;`”. Another kind of entity is called a *named entity*. This kind uses a mnemonic name to represent one or more characters. Named entities have the form “`&`*`name`*`;`”. By default, XML defines a number of named entities. These are listed in Table 4.1.

These basic building blocks of XML documents—open tags, end tags, empty tags, entities, and content—must be put together correctly to produce a *well-formed* document. There are a number of issues involved in a document’s being well-formed, but simply put, it involves:

1. The open tags, end tags, empty tags, attributes, and entities are formed as described above.

2. The text content contains none of the characters “$<$”, “$>$”, or “&”.

3. The document is entirely contained in one element (the root) with no content outside that element.

4. Elements are *properly nested*. This means that the document does not close an element with an end tag if one of that element's parent elements is still open.

While well formedness provides a basic level of structure for an XML document, it does not say anything about the semantic content of the document or whether that is valid: that is, whether the document uses the correct tags and attributes and whether they are combined correctly. There are several ways to specify a document's semantic content. The oldest way is to create a *document type definition* (DTD) for the document. This specifies what tag names and attributes are allowed in the document, as well as what children each tag can contain. DTDs provide the vocabulary for XML documents. Thus, for example, the TEI dictionary subset provides a set of tags and a vocabulary for describing and marking up dictionary entries.

However, sometimes parts of a document serve different purposes, and more than one vocabulary is necessary to capture the different semantics for the different parts of the document. For example, the TEI could be used to mark up a dictionary entry, while scalable vector graphics (SVG), an XML format for graphics, could be used to describe an illustration for the entry. DTDs do not allow vocabularies to be safely combined. For example, both TEI and SVG could define a <`title`> tag, and an application processing such a document would have no way of knowing which DTD defines the ambiguous tag. To remedy this problem, *namespaces* have been added to XML. Namespaces allow the software processing an XML document to tell which tags belong to which semantic vocabulary, while preventing names from different vocabularies from clashing.

Using namespaces involves two steps. First, a *namespace prefix* must be associated with a *namespace uniform resource identifier* (URI; roughly, a URL). Second, the prefix is added to the tag names for elements that are part of that semantic vocabulary.

Associating a namespace prefix with a namespace URI is done by putting a namespace declaration attribute on an element. Usually, this is either the document's root element or the first element that uses the namespace. For example, the standard namespace prefix for an SVG document is "svg", and the namespace URI is "http://www.w3.org/2000/svg". To include an SVG illustration in a TEI dictionary entry, the declaration would be an attribute on the <`entry`> tag reading "`xmlns:svg='http://www.w3.org/2000/svg'`". Later, on tags from the SVG document set, the prefix "svg" is added to the tag name with a colon (<`svg:title`>). For example, Figure 4.1 shows an <`entry`> element for *hexagon* (*MW11*), with an SVG illustration of a hexagon inside it.[1]

Figure 4.1 also provides a good example of marking up a dictionary entry with XML. It has one root element, <`entry`>. Each entry within the root is properly nested. Also, there are a number of character reference entities in the pronunciation and etymology (e.g., "`&#x0259;`" for ə and "`&#x00e4;`" for *ä*). Most of the document contains open and end tags, although there is one empty element, <`ptr target='-gon'/`>, near the end of the etymology.

Beside being an example of XML in general, Figure 4.1 is also good example of using the Text Encoding Initiative DTD (TEI) to mark up dictionary entries. First, a dictionary entry may be contained by a <`superEntry`> element, which contains one or more homographic entries: entries whose headwords are all spelled alike. In the program Schwa, all entries are contained in a <`superEntry`> element. Second, each entry is contained in an <`entry`> tag. The various forms that the headword can take—orthographic (<`orth`>), hyphenated (<`hyph`>), pronunciation (<`pron`>)—are contained in the <`form`> element. Grammatical information, such as part of speech (<`pos`>), number (<`number`>), and others, are contained by the <`gramGrp`> element. Etymologies are in the <`etym`> element. Since this entry only has one sense, its definition is given in the <`def`> element, which is directly under the <`entry`> element. However, an entry with a more complex definition would group the definition

[1]*MW11* does not have an illustration for *hexagon*. It has been supplied for the purposes of this example.

```
<entry>
  <form>
    <orth> hexagon </orth>
    <hyph> hexa|gon </hyph>
    <pron>
      &#x02c8;hek-s&#x0259;-&#x02cc;g&#x00e4;n
    </pron>
  </form>

  <gramGrp> <pos> n </pos> </gramGrp>

  <etym>
    <lang>Gk</lang>
    <mentioned>hexag&#x014d;non</mentioned>,
    neut. of <mentioned>hexag&#x014d;nos</mentioned>
    <gloss>hexagonal</gloss>,
    fr. <mentioned>hexa-</mentioned> +
    <mentioned>g&#x014d;nia</mentioned>
    <gloss>angle</gloss> &#x2014;
    <xr type='etym'>more at <ptr target='-gon'/></xr>
    <date>1570</date>
  </etym>

  <def> a polygon of six angles and six sides </def>

  <svg:svg xmlns:svg='http://www.w3.org/2000/svg'
           style='width: 3in; height: 3in;'>
    <svg:title> A hexagon </svg:title>
    <svg:polygon
        points='50,0 100,25 100,75 50,100 0,75 0,25'
        fill='red' stroke='black' stroke-width='1px'/>
  </svg:svg>
</entry>
```

Figure 4.1: An Entry for *Hexagon* with an SVG Illustration

in a hierarchy of <`sense`> elements, such as is done in the definition of *phonetics* in Figure 3.5.

One useful element, not illustrated here, is the <`usg`> element, which can occur in any other element. The <`usg`> element has the attribute `type`, which indicates what kind of usage information the element contains. There are a number of possible values for the `type` attribute:

geo	geographic area
time	temporal, historical era (“archaic,” “old,” etc.)
dom	domain
reg	register
style	style (figurative, literal, etc.)
plev	preference level (“chiefly,” “usually,” etc.)
acc	acceptability
lang	language for foreign words, spellings, pronunciations, etc.
gram	grammatical usage (Sperberg-McQueen and Burnard)

An example of the <`usg`> element is shown in Figure 4.2, which lists the entry for *entry* from the *Oxford Dictionary of Pronunciation* (*ODP*). This example shows how multiple, nested <`form`> elements can be used to group different pronunciation variants (in this case, different geographical pronunciations) while one parent <`form`> groups the orthographic form of the headword with all the pronunciations. Figure 4.2 also illustrates the `extent` attribute on the <`orth`> or the <`pron`> elements. This attribute indicates whether it contains an entire word or only part of it. In this case, it is used to indicate that a pronunciation is only for the plural suffix for *entry*.

The element <`entry`> can also contain a number of other elements, the most important of which from Schwa’s point of view being the <`note`> element. This tag

```
<entry>
  <form>
    <orth> entry </orth>
    <form>
      <usg type="geo"> Br </usg>
      <pron> &#x02c8;&#x025b;ntr|i </pron>
      <pron extent="suff"> -&#x026a;z </pron>
    </form>
    <form>
      <usg type="geo"> Am </usg>
      <pron> &#x02c8;&#x025b;ntri </pron>
      <pron extent="suff"> -z </pron>
    </form>
  </form>
</entry>
```

Figure 4.2: *ODP* Entry for *Entry* Marked Up Using the TEI

takes a `type` attribute, indicating whether the element contains information about usage, grammar, context, or some other matter. Another important attribute is `resp`, which identifies the person adding the note. Schwa uses the <`note`> element to add miscellaneous information to the entry, such as the details of when an entry was imported into the main database, and it adds a `resp` attribute with the value “schwa” to these elements. These <`note`> elements are also used to add information to the entry about changes that an editor has made, and Schwa can automatically insert a template for these notes, including an appropriate value for the `resp` attribute.

In Schwa, the dictionary entries are maintained as TEI-encoded XML. This allows a number of advantages. First, the data model is standard and is thus able to make use of the thought and decisions that went into designing the TEI and the TEI dictionary subset. Second, keeping the entries in XML allows Schwa to use tools written for working with

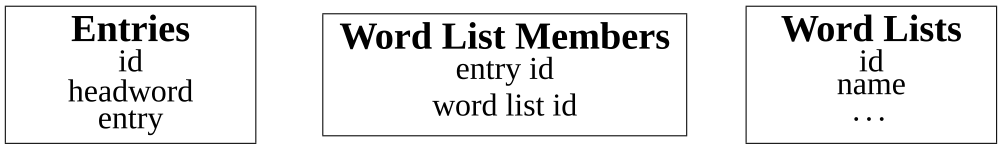


Figure 4.3: ER Model for the Schwa Database

XML. Finally, it allows the data to be kept in a flexible format that respects the data's own intrinsic complexity.

To organize the entries in the database, Schwa uses *word lists*. At the most basic level, a word list is simply a subset of the headwords in the entire database, which can be defined and used in a variety of ways. Word lists are created whenever data is imported into the database, and a second word list is usually also created at that time to list the entries that have been imported that conflict with existing entries. The user can also create word lists by searching the headwords in the database with regular expressions or by searching the entries' XML structure using XPath (Clark). Word lists are also used to view and edit subsets of the database and to export data. Potentially, word lists can be used for any other operation that requires a subset of the database.

The database itself could take a number of forms. One possibility would be a relational database system, in which one table would list the entries, another the word lists, and a third the relationship between the entries and the word lists. Thus, described using an entity/relational model, the entities represented by the database would be *entries* and *word lists*, and there would be a many-to-many relationship between these two, i.e., each entry might be included in many word lists and each word list would have many entries. There may also be other entities, for example, to store the pronunciations from the entries so they can be easily accessible to the database engine. The entities and relationships in the database are represented graphically in Figure 4.3.

When the word *database* is used, typically a relational database is being referred to. However, another option to consider is a more primitive database system, such as a Berkeley database (*Sleepycat Software*). This has tables that simply associate one string value with another, and it does not support any kind of query language, such as SQL. If the added features of relational databases, such as SQL, are not needed, often a Berkeley database is a good option upon which to develop a customized database solution. A Berkeley database for Schwa would have one table for the entries, one for the word lists, and one for each word list, which would contain all the entries in the word list.

Since either of these options would present a good database solution for Schwa, and the technical discussion in Chapter 3 did not suggest that one was clearly better than the other, deciding between them requires implementing both, benchmarking the implementations, and comparing the results. For the relational database solution, two database engines were used. One was the JET database engine, used in Microsoft Access, and the other was MySQL, a large, commercial-grade database server. Although I do not expect users to install and use MySQL, I implemented this for the sake of comparison. For the simpler database solution, I used the Berkeley database system. (The details of the object model for all these implementations is described in Section 4.2.) To benchmark these storage options, a program uses each database to import and export a series of data files ranging in size from 2,000 to 10,000 entries. The time it takes to accomplish importing and exporting is measured in seconds. This procedure, operating on various sizes of data sets, tests the ability of each database to operate both on small databases and on the larger ones that most users will be working with.

The results of running the benchmarks are given in Table 4.2. In comparing the databases, The Berkeley database outperforms all the others, even for smaller data sets, and it also scales better as the data sets become larger. Given the results of the benchmarking tests, I used the Berkeley database system in the Schwa program.

Database	**Number of Entries**				
	2,000	**4,000**	**6,000**	**8,000**	**10,000**
Berkeley	33.08	66.67	101.17	136.76	172.18
JET	80.03	156.11	230.46	304.11	379.03
MySQL	53.76	107.72	161.71	212.96	268.45

The tests were run by importing and exporting the given number of entries into and out of each kind of database. The tests were performed on a 1GHz Athlon PC with 128MB of RAM, running Windows XP Pro, Service Pack 1. The results are given in seconds.

Table 4.2: Results of Benchmarking Storage Options

4.2 CODE

Once the underlying data format has been decided upon, the next step is to decide upon the programming language to use and the structure of the code. In deciding upon the programming language for this application, a number of considerations and criteria are involved. First, the language needs to provide good XML support. This is not the problem it once was, since most languages have solid standard or third-party libraries for working with XML. Second, it needs to be able to work with Unicode. This also is not as much of an issue as before. However, the ease with which languages can encode and decode Unicode and work with data expressed in it still varies. Third, the programming language needs to provide a stable platform for building the application. This also applies to any libraries the application uses. Fourth, it needs to provide a fairly rapid, easy-to-use development process. Fifth, code written in the language needs to be easily maintainable after the program is written, both in terms of debugging and adding functionality.

The number of possible programming languages to consider is almost endless. However, for the purposes of this discussion, I will begin by limiting consideration to those languages I am familiar with and would consider doing such a project in. Thus, for this discussion, I will consider C++, Java, and Python. One caveat concerning this evaluation of languages is that the choice of programming language is at least partly subjective. While one person may value conciseness and be willing to overlook (or even cherish) syntactic clumsiness, another may be willing to put up with a certain amount of verbosity in exchange for a more clean syntax, and still another may be willing to overlook syntactic inconsistencies as long as the semantics are very regular. *De gustibus non disputandum*.

The first language I want to consider is C++. It can process XML using a variety of libraries. The Apache project's Xerces parser provides parsing facilities for XML in C and C++. Also, its Xalan project provides for XSLT and XPath. Likewise, its Unicode facilities are good. On the level of C, the `wchar_t` (wide character) type supports working with Unicode strings, and the C++ standard library has the `wstring` object, which allows for manipulating Unicode strings in an object-oriented manner. Libraries such as iconv provide for encoding and decoding Unicode data (*libiconv*). The problem with C++ is stability and maintenance. Theoretically, C++ can be very stable. However, writing stable applications in C++ requires time to work out pointer issues and to implement and debug them. These factors also make C++ less maintainable than others. In general, while C++ would be satisfactory, there are better solutions in the other languages.

Of course, no modern discussion of languages would be complete without mentioning Java (*Source for Java*). Its internal Unicode handling is excellent: all strings are handled as Unicode. Also, its XML facilities are very good. Java is generally stable and is easy to develop in, and programs written in it are maintainable. However, its memory requirements have given it, and particularly its GUI libraries, the reputation of being less responsive than their C and C++ counterparts.

Finally, the last language, Python, is one of a collection of scripting languages. These are very high-level, dynamic, interpreted languages that have become popular in a variety of domains, including web programming and system utilities. Python's Unicode handling is excellent. It also has the added benefit that programmer-written codecs can be incorporated and called using the same mechanisms as built-in codecs. That is, the programmer can define a function for converting between one of the phonetic character sets that redefine the upper range of the ANSI character set, and it can be used in the program as naturally as the UTF-8 or Latin-1 encodings. Also, there are a number of good libraries for handling XML, and since Python has been under development for as long as Java has, it is about as stable. One of the touted benefits of this class of languages is their rapid development time, and this applies to Python also. Also, its emphasis on readability makes it very maintainable. Keeping these considerations in mind, Python appears to be the best option. I know the language well and am productive in it. Its Unicode and XML handling and, most of all, its rapid development time and maintainability all recommend it.

The Python website describes this language as being “an *interpreted, interactive, object-oriented* programming language,” which has a very clear syntax (“What Is Python?”). Python is interpreted, not compiled, and it has strong, but dynamic, typing (i.e., it strongly differentiates between different value types, but only values are typed, not variables). Because of these two factors, writing code in Python is typically quite fast. While it is possible to use just functions to organize code, the basic building blocks of Python programs are objects. Objects combine data with actions. In Python, the data attached to objects is referred to as *attributes*, and the actions are *methods*. Also, attributes and methods can be combined as *properties*. Syntactically, properties look like attributes; however, when a property is set or accessed, a method call is triggered. This allows data to be dynamically calculated or protected so that a property can only be read from, not written to. Objects can subclass other objects, in which case they inherit the attributes and methods of the object they subclass, and can selectively redefine them.

On a higher level, code is organized into *modules*, which collect data, functions, and classes into one *namespace*. In Python (as opposed to XML) a namespace is simply a named grouping of entities. For example, the `os` module in the standard library provides access to the operating system. Inside it, there is the constant `name`, which contains the name of the operating system. It also has a function `chdir`, which changes the current directory. Both of these—`name` and `chdir`—exist in the namespace of `os`. They are accessed using dot notation: “`os.name`” and “`os.chdir`”. Modules can also be organized into packages, which are simply modules that contain other modules. For example, `os` is actually a package that itself contains a module named `path`, which provides functions to manipulate OS path names. The module `path` contains a function named `abspath`, which creates an absolute path name from a relative path. Again, it is accessed using dot notation: “`os.path.abspath`”. This introduction to Python should provide enough information to follow the description of how the code in Schwa is organized.

Having decided on the language, the next consideration is the graphical user interface library, or toolkit. There are a number of GUI toolkits available and in widespread use for Python. The de facto standard is Tcl/Tk (*Tcl SourceForge Project*). Another popular one is the wxWindows library. There are also a number of other toolkits, but these are not as popular, and none of the others can handle Unicode.

The first, Tcl/Tk, is accessed through Python using the Tkinter module. This toolkit has been in active development for a long time, so it is very stable. This is also a drawback, however, because Tkinter does not have as rich a set of controls as other libraries. Where Tkinter shines, however, is in its Unicode handling. Beyond simply displaying a Unicode string, if the current font does not have a glyph defined for a character, Tkinter tries to find a font to use that does define it. This works better on some platforms than others, and finding a font for a character can take a long time if many fonts are installed, but this is still a nice feature that none of the other toolkits provide. The awkward part of Tkinter is passing Unicode data in and out of it, which is sometimes temperamental: sometimes it will accept a Python `unicode` object; at other times, though, it wants a UTF-8 encoded string.

The second toolkit, wxWindows, and its Python module, wxPython, provide a cross-platform layer on those of the platform's native controls (*wxWindows*). It has not been in active development as long as Tkinter, but it is still quite stable. Moreover, its set of controls is very full. On the other hand, its Unicode handling works best on platforms that support it natively. Windows 98 and ME, however, can use wxWindows in conjunction with Microsoft's Unicode emulation layer. Thus, given its adequate Unicode handling and its superior set of controls, wxPython appears to be the best option for creating the user interface.

Much of the code in Schwa makes use of a Python library I have been accumulating for some time, the `ericr` package. While some parts of this package are still quite rough, the data structures and functions that Schwa depends upon are solid.

Perhaps the most important module in this package is `ericr.app`. This contains the `Application` class, which provides a framework for creating applications. New applications subclass the `Application` object, primarily to redefine the `main` method. By using this class, applications automatically get command-line option parsing, information logging, and an exception handling framework. Schwa makes use of this class both in the main Schwa application and in some utility programs.

The `ericr` package also defines some useful data structures and design patterns. Data structures are general-purpose classes that are good in a number of situations. Design patterns are more abstract solutions to problems which occur repeatedly and can therefore be identified and analyzed. Often, design patterns involve many classes interacting together. The first design pattern implemented in the `ericr` package is the Observer pattern (Gamma, et al., 293). This pattern is used whenever one or more classes needs to be notified of events by other classes. The implementation of the Observer pattern provided in `ericr.datastruct.observer.ObserverList` is a standard Python list that is callable. The `ObserverList` instance is populated with functions or methods. Then, the class sending the notifications calls the `ObserverList` as if it were a function. The

`ObserverList`, in turn, calls each item in it with the same arguments. In Schwa, this class is used, among other things, whenever one class needs to keep track of the progress of another, for example, in the `schwalib.task` framework, described below.

Beside general-purpose objects, the `ericr` package also provides a number of utility functions and objects for working with XML and text. The `ericr.ml.utils` module provides a number of functions for working with XML data. The `normalize` function changes all sequences of whitespace in a string to a single space. `escape` converts the "<", ">", and "&" characters in a string to character entities, so it can be printed as XML content, and `quoteattr` escapes the characters in a string, so it can be used for the value of an attribute. The `ericr.text` package provides modules that operate on textual data. The `ericr.text.decompose.decompose` function *decomposes* Unicode characters: it takes composite characters, such as *á*, and turns them into a string of two characters (*a* and ´). The `ericr.text.interp.Interpolator` class performs simple value substitution in a string. `Interpolator` classes inherit from `dict`, the standard Python dictionary or hash table class, so they act as a dictionary on one level, but calling the `interpolate` method on these objects substitutes the values in the dictionary into slots indicated by the dictionary key names. The syntax for these substitution markers is the same as for Perl's variable interpolation feature: a "$" prefix. For example, if "x" is associated with the value 42 in the dictionary interpolator, performing interpolation on the string "the answer = $x" would produce the string "the answer = 42". While this is functionally similar to Python's own string formatting operator (`%`), which is in fact more powerful, the syntax that the `Interpolator` class uses is both familiar and simpler.

While the `ericr` package provides some fundamental building blocks and utilities for the Schwa application, most of the code is in the `schwalib` package. This package groups the code for the application, its data objects, and its user interface into one package that is easy to maintain and distribute.

The core of the package is the `schwalib.db` module. This contains the classes for working with databases and data objects. The main class is `Database`, which creates, opens, and closes the database and creates data objects. There are a number of methods and properties that make up the public interface for `Database` objects:

`Database(dbhome)` This initializes and returns a new `Database` object. `dbhome` is the directory that contains (or will contain) the database files.

`.entries` This property returns a `DbDict` object (see page 73) that accesses the entries in the database.

`.wordlists` This property returns a `DbDict` object that accesses the word lists in the database.

`.iorth` This property returns a `Index` object (see page 74) that indexes the entries by orthographic form.

`.ipron` This property returns a `Index` object that indexes the entries by pronunciation.

`.Entry(xml, key)` This method constructs a new `Entry` object (see page 74). `xml` is the XML data representing the entry. `key` is an optional key to store the entry under in the database. If not provided, the value of `key` is inferred from the XML data.

`.WordList(name, source, note, created, mdate)` This method constructs a new `WordList` object (see page 75). The arguments are used to create the word list. The most important argument is `name`, which specifies the name the word list is stored under in the database.

The entries and word lists are accessed through `schwalib.db.DbDict` objects. The interface for these objects is essentially the same as that for standard Python `dict` objects. Objects are stored in the database by associating them with a key in the `DbDict` object, and they are retrieved using the same key. Additionally, there is an `ObserverList` object that allows other objects to observe additions and deletions. Currently, this is used to maintain up-to-date indices on the main database. The two methods defined for `DbDict` that are not taken from `dict` are `close()`, which closes the underlying database table, and `getFrom(key, n, set)`, which retrieves the next *n* keys, beginning from *key*. If the optional argument *set* is specified, the keys must also be members of that set of entry headwords. This method is used to retrieve keys for the list of entries in Schwa's main editing window.

Schwa maintains two indices on the main entry database, one on the orthographic forms and one on the pronunciations in the entries. These indices are implemented by the `schwalib.db.Index` class. The primary interface for this class is the same as that for Python `dict` objects. Like `DbDict`, this class also defines a `close()` method, which closes the underlying database table. Objects are retrieved from indices using the value they are indexed by, and a list of all the objects that are stored in that index is returned. For example, retrieving the value "-z" from the pronunciation index will return a list of all the entries that have a suffix pronounced [-z]. `Index` objects also act as observers of `DbDict` objects, automatically updating their contents based upon changes made to the database they are observing.

The main data for the Schwa program is represented using the `schwalib.db.Entry` class. This class's primary data is held in the `doc` attribute, which holds the entry's XML data stored as libxml2 data structures (Veillard). Libxml2 is the XML library for the GNOME project (*GNOME*), and these data structures are similar to the World Wide Web Consortium's DOM standard (Le Hors, et al.). The `entry` property provides access to the XML data as a string. The `key` property accesses and sets the key under which the entry is to be stored in the database. Finally, the `Entry` object also overloads the in-place addition operator (`+=`). This takes another `Entry` object and merges it into the current entry. This is used when importing data, so that the new data is merged into the database without overwriting any existing entries.

The last class in the data model is `schwalib.db.WordList`. Word lists are a subset of the entire database, as described earlier. `WordList` objects publish three operations: addition, deletion, and querying. Addition is handled by the `add(key)` method, which takes a *key* and simply adds it to the `WordList`'s subset. Deletion is handled by `remove(key)`, which removes the key or throws and exception if the key is not in the subset, and by `discard(key)`, which removes the key if it is in the subset or silently returns if it is not. Querying the `WordList` subset for membership is handled by overloading the `in` operator: *`key`* `in word_list`.

`WordList` objects can be populated using a variety of different methods. Importing data populates a word list, by adding entries as they are imported. `WordList`s can also be created from the entries already in the database by searching the entries' orthographic forms, by filtering the data using an XPath expression, and by including all the headwords listed in a file.

The last two methods of defining a `WordList`—by XPath expression and by a headword file—are implemented in the `schwalib.wordlists` module. This defines two functions. The first, `XPathTask(db, wl, xpath_expr)` takes a `Database` as its first argument and a `WordList` to add entries to as its second argument. The third argument is an XPath expression to use in testing the entries in the database. The second function, `ListTask(db, wl, filename)` takes the same first two arguments as the first function. The third argument is a file name identifying a file that lists the headwords. Both functions are factory functions that create and return an instance of `schwalib.task.-TaskThread`, which adds entries to the `WordList` instance when run (see page 78 for more information on `TaskThread` objects).

In defining `WordList` objects from XPath expressions, users can use a number of XML namespaces and extension functions that Schwa makes available (see Appendix C for a short description of XPath and for an explanation of how to use these extensions functions). Using these extensions complicates setting up the XPath processor, so all access to the XPath processor is kept in the `schwalib.utils.xpath` module, which exposes two functions in its public interface. The first, `valid(expr)` takes an XPath expression and tests it for validity. If it is invalid, it returns an `Exception` object representing the problem. The second function, `evaluate(expr, doc)` takes an XPath expression and a libxml2 document and returns a list of the results of the expression. These functions provide a convenient entry point for using XPath, without having to set up the XML namespaces and extensions functions that Schwa provides. This also simplifies maintaining Schwa, because XPath extensions can be added to this module, rather than at every point in the code that uses XPath.

One of these extension functions is implemented in the `schwalib.utils.-regexp` module, which implements the `regexp:test(string, pattern, flags)` function from the EXSLT collection of XPath extensions (*EXSLT*). This function tests strings to see if they match a regular expression, which is useful in populating a `WordList` from an XPath expression.

Another set of XPath extension functions is provided in the `schwalib.prons` module, which Schwa makes available using the *pron* namespace prefix. These functions are described in more detail in Appendix C.

Schwa also provides a module to facilitate working with pronunciations from a programmer's perspective: `schwalib.pom`. Although not currently used in Schwa, it does provide a useful tool for future extensions. This module implements a pronunciation object model (POM), which deals with pronunciations as a multi-level tree structure.

The levels are pronunciation, word, segment,[2] and symbol. These objects publish these attributes and methods:

`.data` For most objects—`Pronunciation`, `Word`, and `Segment`—this is a list of the children of the current node. For `Symbol` objects, this is the Unicode character of the symbol the object represents.

`.diacritics` This is a list of Unicode characters representing symbols modifying the structure. For different levels, this list has different semantics: `Pronunciation` and `Word` objects, it is ignored; for `Segment` objects, it is a list of suprasegmentals; for `Symbol` objects, it is a list of diacritics.

`unicode(`*`node`*`)` Creating a Python `unicode` object from a POM object returns a Unicode string containing the phonetic data represented by the POM and its children.

For example, the POM tree for *New Orleans* is represented in Figure 4.4. This example illustrates several things about the object model. First, the suprasegmental primary stress is pulled out into the `diacritics` attribute of the `Segment` object for [ɔr], as the secondary stress is for [n(j)u]. While the stress marker does not exist at the level of the `Symbol`, other diacritics, for example nasalization, would exist there, but not at the level of the `Segment`. Second, the optional [j] in *new* is represented by maintaining the parenthesis around the symbol at all levels. Other variation, however, is best represented using another pronunciation and another POM tree. For example, the alternate American pronunciation for *New Orleans* listed in the *ODP*, [ˌn(j)u ərˈlinz], would be best represented using another `Pronunciation` object tree. Thus, POM objects are designed to integrate

[2]Throughout this discussion, the term *segment* is intentionally poorly defined. It may correspond to syllabication, or—especially in the context of dictionary pronunciations—it may correspond to the hyphenation given for the headword.

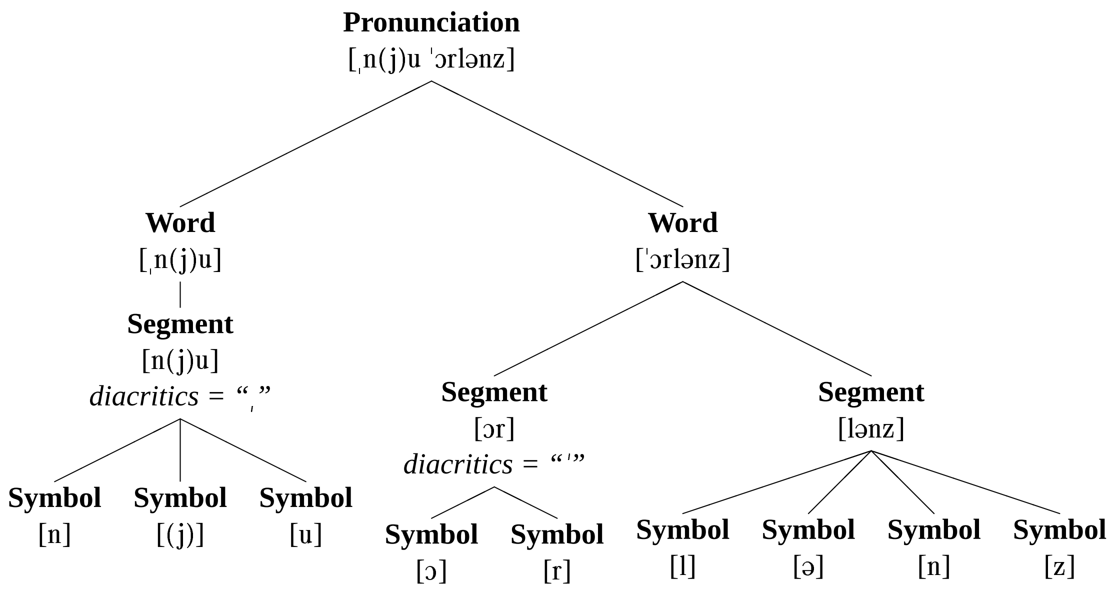


Figure 4.4: Pronunciation Object Model for *New Orleans*

into Schwa's XML entries by representing all the content of a single <`pron`> element, not multiple elements.

While POM objects can be converted to Unicode strings directly, creating POM objects from Unicode is a bit more complicated. To facilitate this, the `schwalib.-pom` module includes a utility function, `fromUnicode(string)` that takes a Python `unicode` object and returns a `Pronunciation` object that represents the phonetic data in the input string.

One possible extension would be to implement a final level of abstraction below *Symbol*, called *Feature*. This could hold a feature name and a value, e.g., *voiced* and *true*, and could allow for searching and manipulation by features. However, even at its current stage, the `schwalib.pom` module is a useful tool for programmers working with phonetic data.

Another useful framework that the Schwa library package provides is a set of classes for executing long-running tasks in a separate thread. Simply put, threads are like multiple programs, executing simultaneously and sharing memory, data structures, and resources like files and database connections. Although easy to explain, working with threads is difficult. Making sure that multiple threads concurrently modifying the same data structures do not leave those structures in an invalid state involves a lot of thought, planning, and a few special-purpose data structures. The central object in this framework is `schwalib.-task.TaskThread`, and it inherits from the `threading.Thread` class in the standard Python library. This class has a number of features beyond those of standard Python threads: first, it generates progress messages, and second, it can be paused or stopped before the task is finished.

To create a long-running task, the programmer defines a class that inherits from `TaskThread` and overrides the `perform()` method. Inside the `perform()` method, there are also a number of conventions the programmer should follow to make sure that the class's requirements are met. First, either in the class's initializer or at the beginning of `perform()`, she should set the `total` attribute to an integer indicating the size of the task. While executing the task, she should do a number of things:

1. Check the value of the `stop` property. If it is true, the task should exit gracefully.
2. Call the `pause()` method. If the task is being asked to pause (to wait for user input, for example), this call will block until the task has been given permission to resume.
3. Update the `count` property. This will generate a progress report and notify any interested classes.

4. Call one of `warning(exception)`, `error(exception)`, or `fatal(exception)` in the event of an error. This will pause the task, generate an error report, and notify any interested classes.

`TaskThread` objects have an `observers` property, which is an instance of `ObserverList`, so classes that are interested in progress or error reports need to define a method that accepts two arguments: first, the task instance; second, the object providing the status or error information, which will be an instance of `schwalib.-task.ProgressInfo` or `schwalib.task.ErrorInfo`. Then, the class needs to append that method to the task's `observers` property.

For example, Figure 4.5 presents a class that counts to a total, while notifying another thread, which prints the notification message. In this case, the observer is just a function, `observer`, that prints out a message. If the notification is for an error, the function prompts the user for whether or not the task should continue. Finally, the program creates a `CounterTask` instance, registers the `observer` function, and starts the task.

One of the primary purposes of the `TaskThread` framework is importing and exporting data. A filter to import and export data using a certain format must be accessible from a class that uses the interface defined by `schwalib.transfer.-core.IDataTransfer`. This class defines two factory functions. The first, `make-Exporter(db, obj, dest)` takes a `Database`, a `WordList` to export, and a file name to export to. The second, `makeImporter(db, wl, source)` takes a `Database`, a `WordList` to import into, and a file name. Both functions return an instance of `TaskThread`. When the task to started, the data transfer will be performed.

Since the data in Schwa is kept in TEI-encoded XML, this represents a good format to provide an importer and exporter for. The entry point for this transfer format is in `schwalib.transfer.xmltrans.XMLDataTransfer`. In exporting, the entries and word lists are placed in <`TEI.2`> elements, which are placed in a <`teiCorpus.-`

```
class CounterTask(TaskThread):
    def __init__(self, countTo, errorOn=-1):
        TaskThread.__init__(self)
        self.total = countTo
        self.errorOn = errorOn
    def perform(self):
        while ((not self.stop) and
               (self.count < self.total)):
            self.pause()
            if self.count == self.errorOn:
                self.error(Exception(str(self.count)))
            self.count += 1

def observer(task, info):
    if hasattr(info, 'exception'):
        print 'ERROR', str(info.exception)
        print info.message
        print
        print '[C]ontinue or [S]top (S)? ',
        input = sys.stdin.readline()
        if input: input = input[0].lower()
        else: input = 's'
        if input == 's': task.stop = True
        task.paused = False
    else:
        print 'PROGRESS', info.percent

task = CounterTask(10000, -1)
task.observers.append(observer)
task.start()
```

Figure 4.5: An Example of a Long-Running Task

2> element. If only a word list is exported, only the <TEI.2> element for that word list is exported. In importing, the elements are simply merged directly into the database.

Beside TEI-compliant XML, Schwa also can transfer data to and from files in a form of GML used by Oxford University Press for the *OED*. The entry point for this format is `schwalib.oxford.OxfordDataTransfer`. Briefly, GML is an ancestor of XML, and it is very similar to XML, with these differences:

1. The elements in the document are not required to occur within a single parent element.
2. Atomic attribute values do not have to be quoted.
3. General entity references begin with an ampersand and end with a period (not a semi-colon), for example, "`&aacu.`".
4. Ampersand characters can occur outside of the context of an entity reference, for example, "`this & that`".

For example, Figure 4.6 contains an entry in GML. The specific tag set that is shown here and that Schwa can import is a subset of what Oxford University Press uses in dictionary production.

Because of the differences between GML and current markup languages, Schwa contains a GML parser that transforms each top-level element into a valid DOM document. This document is then transformed into a TEI dictionary entry and entered into the database.

Another aspect of the Oxford import filter is the `schwalib.oxford` codec. This encodes ASCII strings that use standard characters to represent phonetics into Unicode IPA strings. These codes use a set of correspondences similar to that of the tipa LaTeX module. This codec can be used exactly like any other Python codecs, so calling "`unicode('%peIp@rm@"SeI', 'schwalib.oxford')`" returns the Unicode string "ˌpeɪpərməˈʃeɪ".

```
<e>
  <hw>papier-m&acirc.ch&eacu.</hw>
  <pr>
    <la>Brit.</la> <ph>%papjeI"maSeI</ph>,
    <la>U.S.</la> <ph>%peIp@rm@"SeI</ph>
  </pr>
</e>
```

Figure 4.6: *Papier-mâché*: An Example GML Dictionary Entry

Together, the data transfer object and the codec import and export data to and from the Oxford GML format. For an example of this, compare the GML entry in Figure 4.6 with the TEI entry in Figure 4.7. Notice that information that Schwa adds for its own use, in this case the `key` attribute, is kept in the `schwa` XML namespace.

The classes discussed so far all work together to implement the data model, the database, and classes that operate on the data. They are not useful, however, without a user interface to make them available to people who want to edit and maintain phonetic data. Classes that implement the user interface are in the `schwalib.gui` package. The entry point for the application is the class `schwalib.gui.app.SchwaApplication`, which subclasses `ericr.app.Application`. Functionally, this class primarily creates the wxPython GUI application and hands off control to it.

The main window for the application is implemented in `schwalib.gui.frame.-SchwaFrame`. Functionally, this is the most important class in the user interface. Beside creating the main window, it also manages the application's options and event handlers for the menu and toolbar. The three windows that control user interaction—the editing window, the search window, and the view window—are implemented in the

```
<?xml version="1.0" encoding="UTF-8"?>
<superEntry
  xmlns:schwa="http://www.arches.uga.edu/˜erochest/schwa"
  schwa:key="papiermache">

  <entry>
    <form>
      <orth>papier-m&#x00e2;ch&#x00e9;</orth>
      <form>
        <usg type="geo">Brit.</usg>
        <pron>
          &#x02cc;papje&#x026a;
          &#x02c8;ma&#x0283;e&#026a;
        </pron>,
        <usg type="geo">Am.</usg>
        <pron>
          &#x02cc;pe&#x026a;p&#x0259;r
          m&#x0259;&#x02c8;&#x0283;e&#026a;
        </pron>
      </form>
    </form>
  </entry>
</superEntry>
```

Figure 4.7: *Papier-mâché*: Imported into Schwa

`schwalib.gui.child` module. Aside from creating their user interfaces and handling events for controls directly on themselves, they have minimal functionality.

Further features are provided by the `schwalib.gui.keymaps.KeyMaps` class. The object manages a set of mappings between keyboard combinations and strings of characters. It also implements a GUI event handler, so whenever a keyboard combination is pressed, it inserts the corresponding text into whatever control has the keyboard focus. Of course, the `KeyMaps` event handler must first be associated with the control for this to work. Most of the interface for the `KeyMaps` class is similar to that of the standard Python dictionary. However, it also has a method, `load(filename)`, which loads the keyboard mappings from a file, and another method, `save(filename)`, which saves the keyboard mappings to a file.

Another feature is implemented by the `schwalib.gui.tools.ToolsManager` class. Tools are a way that users can customize Schwa and extend its functionality. The `ToolsManager` creates a GUI menu from the contents of the `%UserProfile%\Application Data\schwalib\Tools` directory and acts as an event handler for those menu items. Whenever one is selected, it executes the Python script from the `Tools` directory associated with that menu item. These scripts are executed with one global variable predefined: `frame`, which is the current instance of `schwalib.gui.frame.SchwaFrame`. A sample tool that opens the `Tools` directory is bundled with Schwa (Figure 4.8). This provides an easy way for users to locate the `Tools` directory and to add their own scripts there.

Another important GUI feature is handling the `TaskThread` objects used to transfer data into and out of Schwa. This is done with `schwalib.gui.dialogs.task.TaskDialog` class, which runs a task and observes it. As it receives progress messages, it updates a progress bar on a dialog box. This dialog also has a “Cancel” button on it, which stops the task. Thus, the added thread control that `TaskThread` provides meshes nicely with the functionality required of the GUI.

```
import os

os.startfile(frame.tools.directory)
```

Figure 4.8: The Schwa Tool, "Open Tools Directory.py"

These packages and modules contain the source code for Schwa. However, as much as Python facilitates programming, it does not allow programmers to distribute their code with the easy-to-use installers that users of Windows have come to expect. To provide this kind of installation utility, first the code for Schwa needs to be packaged with the parts of the Python system it relies on, then compiled into an installation program.

The first part of this is accomplished by a simple C program, which is modeled on the Python interactive interpreter. This program simply starts the Python system. After starting the system, it adds three compressed ZIP files to Python's search path for modules: one ZIP file for the standard Python library, one for the `ericr` package, and one for the `schwalib` package. Finally, it runs a file named "schwa.py" found in the same directory as itself. With this system, a user does not need to have Python or any third-party modules that Schwa uses installed on her system.

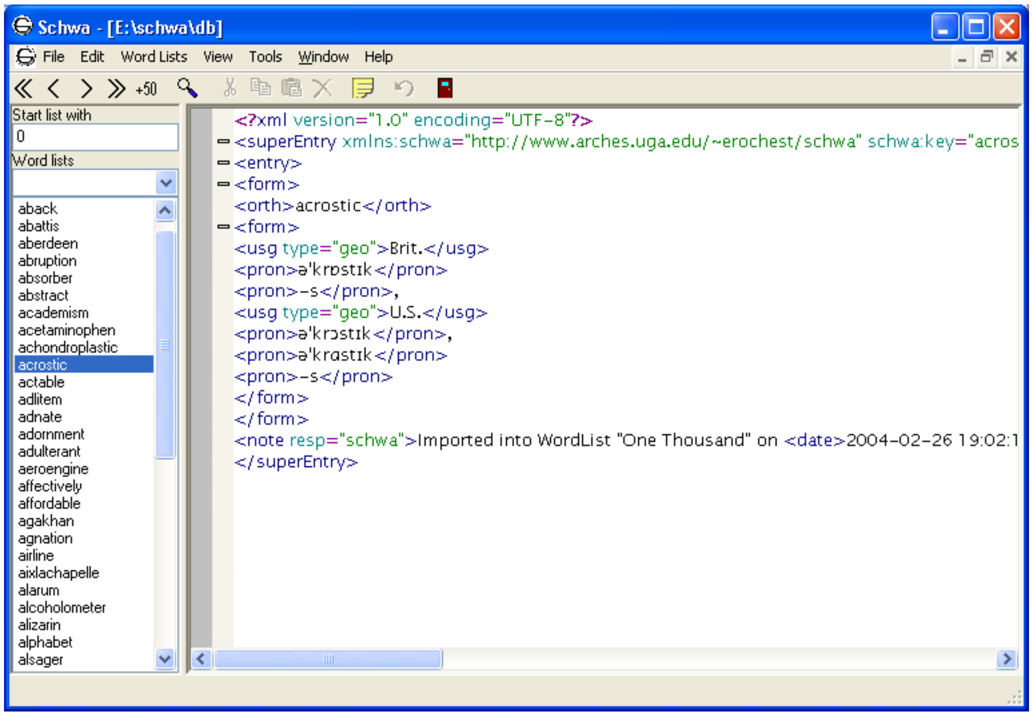
Actually to place everything on the user's system appropriately, Schwa uses the Inno Setup Compiler, a free installation program compiler (Russell). This places Schwa on the system, creates the application data directory and tools directory and shortcuts for the program.

4.3 The Program

The user only sees the visible effects of this code. To use Schwa effectively, the user needs to know how to navigate the windows and dialogs effectively. This section describes Schwa's user interface and how to use it to accomplish the task of managing and editing lexicographical pronunciations.

The primary window in Schwa and the one in which users will spend the most time is the editing window (Figure 4.9). This is a standard Windows application, with a menu and toolbar across the top of the window and a status bar across the bottom. Down the left side of the editing window is a navigational panel, and a simple XML editor fills the rest of the space. To navigate through the entries in the database, the user first must enter a headword to begin browsing with in the "Start list with" text box. Then, the user can optionally select a word list from the drop-down box labeled "Word lists." If a word list is selected, only entries in that word list will be displayed in the list that fills most of the navigational panel. Once a word is entered in the "Start list with" box, when the user presses `ENTER`, the list of words is populated with at most fifty words. To select the first word in the list of words for editing, the user simply clicks on that item in the list. To navigate through the list of words, there are a number of options, which are listed in Table 4.3. Whenever the user navigates out of an entry, any changes made to it are saved, so the user does not need to hit any special key to keep her changes.

When editing an entry, the text editor works like most other text editors. The standard Windows clipboard functions—cut, copy, and paste—are available either through a pop-up window, menu items, toolbar buttons, or keyboard shortcuts (unless they have been reassigned using key maps). Moreover, the user can easily enter phonetic characters using keyboard shortcuts. These shortcuts are defined by the user on the "Key Maps" tab of the Options dialog. For example, in the default set of key mappings, `ALT-e` is assigned to the schwa character, so when the user presses that shortcut, ə is entered. The default key



```
<?xml version="1.0" encoding="UTF-8"?>
<superEntry xmlns:schwa="http://www.arches.uga.edu/~erochest/schwa" schwa:key="acros
<entry>
<form>
<orth>acrostic</orth>
<form>
<usg type="geo">Brit.</usg>
<pron>ə'krɒstɪk</pron>
<pron>-s</pron>,
<usg type="geo">U.S.</usg>
<pron>ə'krɔstɪk</pron>,
<pron>ə'krɑstɪk</pron>
<pron>-s</pron>
</form>
</form>
<note resp="schwa">Imported into WordList "One Thousand" on <date>2004-02-26 19:02:1
</superEntry>
```

Figure 4.9: Editing Window

mappings are those developed by William A. Kretzschmar, Jr., and Rafal Konopka for entering phonetic data for the Linguistic Atlas projects, and they are listed in Appendix A.

When editing pronunciations, often the user will need to view the pronunciation of similar words to check for consistency. This is done using the search window (Figure 4.10). This window has a bar across the top that allows the user to enter search terms, which is either a simple string or a regular expression search pattern. The search term is matched against the entries' orthographic forms.

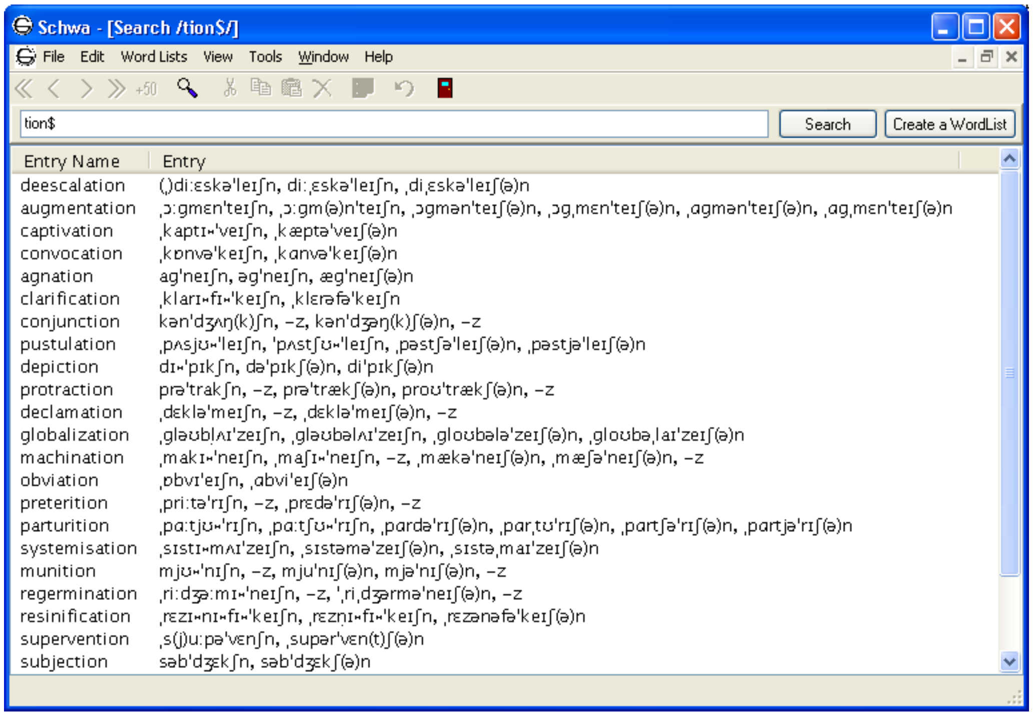
Regular expressions are a powerful language for expressing text searches. They allow you to search for patterns in text that plain string searches do not. For example, you can specify that you want all the words that end in *y*. There are many dialects of regular expres-

Movement	Menu	Toolbar	Shortcut Key
Move to the next fifty entries in the database	View → Next 50	+50	F9
Move to the first entry in the list of entries	View → First	«	F5
Move to the previous entry in the list of entries	View → Previous	‹	F6
Move to the next entry in the list of entries	View → Next	›	F7
Move to the last entry in the list of entries	View → Last	»	F8

Table 4.3: Navigating the List of Words in the Editing Window

sions. The one that Schwa uses is probably the most popular: the regular expression syntax of Perl (and Python). Appendix B provides a short introduction to regular expressions.

After starting the search, Schwa displays the results in the main part of the window as a list of headwords and pronunciations. Often, searching for a common string can produce an overwhelming number of results. To manage them, it is easy to scan only the first few dozen entries. Although convenient, this does not provide a good overview of the results. To handle large result sets, the user can have Schwa only display a random sample. This facilitates looking through the result sets and thereby encourages consistency across the database, which results in better pronunciations. The results are presented in two columns: one lists the headword and the other lists all the pronunciations in the entry. After getting the results, the user can double-click on one to open it in an entry view window (Figure 4.11). This entry cannot be edited, but it allows the user to see the entire entry. Also, the

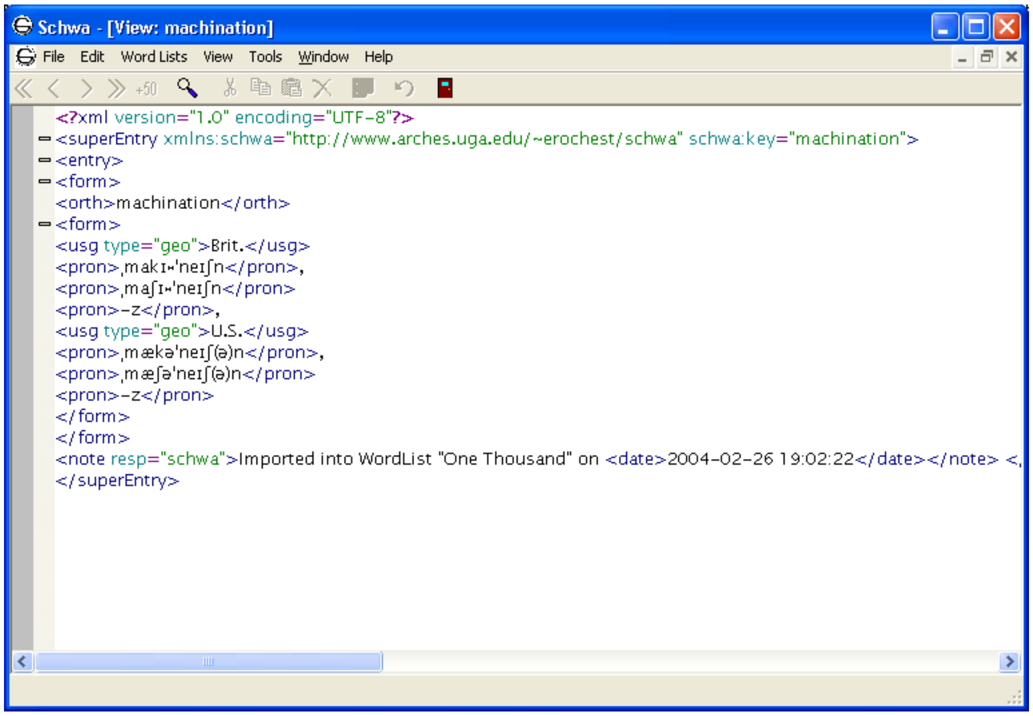


Entry Name	Entry
deescalation	()di:ɛskə'leɪʃn, di:ˌɛskə'leɪʃn, ˌdiˌɛskə'leɪʃ(ə)n
augmentation	ˌɔ:gmɛn'teɪʃn, ˌɔ:gm(ə)n'teɪʃn, ˌɔgmən'teɪʃ(ə)n, ˌɔgˌmɛn'teɪʃ(ə)n, ˌɑgmən'teɪʃ(ə)n, ˌɑgˌmɛn'teɪʃ(ə)n
captivation	ˌkaptɪ-'veɪʃn, ˌkæptə'veɪʃ(ə)n
convocation	ˌkɒnvə'keɪʃn, ˌkɑnvə'keɪʃ(ə)n
agnation	ag'neɪʃn, əg'neɪʃn, æg'neɪʃ(ə)n
clarification	ˌklarɪ-fɪ-'keɪʃn, ˌklɛrəfə'keɪʃn
conjunction	kən'dʒʌŋ(k)ʃn, –z, kən'dʒəŋ(k)ʃ(ə)n, –z
pustulation	ˌpʌsjʊ-'leɪʃn, 'pʌstʃʊ-'leɪʃn, ˌpəstʃə'leɪʃ(ə)n, ˌpəstjə'leɪʃ(ə)n
depiction	dɪ-'pɪkʃn, də'pɪkʃ(ə)n, di'pɪkʃ(ə)n
protraction	prə'trakʃn, –z, prə'trækʃ(ə)n, proʊ'trækʃ(ə)n, –z
declamation	ˌdɛklə'meɪʃn, –z, ˌdɛklə'meɪʃ(ə)n, –z
globalization	ˌgləʊblʌɪ'zeɪʃn, ˌgləʊbəlʌɪ'zeɪʃn, ˌgloʊbələ'zeɪʃ(ə)n, ˌgloʊbəˌlaɪ'zeɪʃ(ə)n
machination	ˌmakɪ-'neɪʃn, ˌmaʃɪ-'neɪʃn, –z, ˌmækə'neɪʃ(ə)n, ˌmæʃə'neɪʃ(ə)n, –z
obviation	ˌɒbvɪ'eɪʃn, ˌɑbvi'eɪʃ(ə)n
preterition	ˌpri:tə'rɪʃn, –z, ˌprɛdə'rɪʃ(ə)n, –z
parturition	ˌpɑ:tjʊ-'rɪʃn, ˌpɑ:tʃʊ-'rɪʃn, ˌpɑrdə'rɪʃ(ə)n, ˌpɑrˌtʊ'rɪʃ(ə)n, ˌpɑrtʃə'rɪʃ(ə)n, ˌpɑrtjə'rɪʃ(ə)n
systemisation	ˌsɪstɪ-mʌɪ'zeɪʃn, ˌsɪstəmə'zeɪʃ(ə)n, ˌsɪstəˌmaɪ'zeɪʃ(ə)n
munition	mjʊ-'nɪʃn, –z, mju'nɪʃ(ə)n, mjə'nɪʃ(ə)n, –z
regermination	ˌri:dʒə:mɪ-'neɪʃn, –z, 'ˌriˌdʒərmə'neɪʃ(ə)n, –z
resinification	ˌrɛzɪ-nɪ-fɪ-'keɪʃn, ˌrɛzn̩ɪ-fɪ-'keɪʃn, ˌrɛzənəfə'keɪʃ(ə)n
supervention	ˌs(j)u:pə'vɛnʃn, ˌsupər'vɛn(t)ʃ(ə)n
subjection	səb'dʒɛkʃn, səb'dʒɛkʃ(ə)n

Figure 4.10: Search Window

search results can be saved for later viewing and editing by creating a word list. This is done by pressing the “Create a WordList” button at the top right of the search window.

Some of the behavior of the search window and the rest of Schwa can be controlled using the options dialog. This is available by selecting the “Edit → Options…” menu item. This dialog contains a number of tabbed pages, the first of which, titled “Editor,” handles general options for Schwa (Figure 4.12). First, this tab contains an option to allow “folding” in the editor. When enabled, the editor places plus or minus signs in the left margin. These icons correspond to the current entry’s structure, and clicking on them toggles whether parts of the entry are displayed or hidden. For example, the beginning of a multi-line <`form`> element would initially have a minus icon displayed. Clicking that icon would cause the contents of the <`form`> element to be hidden and the icon to turn

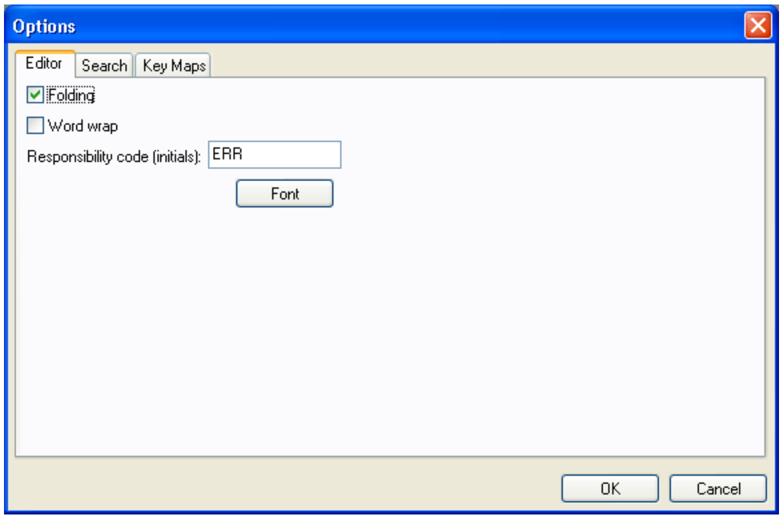

Schwa - [View: machination]
File Edit Word Lists View Tools Window Help
+50

```
<?xml version="1.0" encoding="UTF-8"?>
<superEntry xmlns:schwa="http://www.arches.uga.edu/~erochest/schwa" schwa:key="machination">
<entry>
<form>
<orth>machination</orth>
<form>
<usg type="geo">Brit.</usg>
<pron>ˌmakɪ̈ˈneɪʃn</pron>,
<pron>ˌmaʃɪ̈ˈneɪʃn</pron>
<pron>-z</pron>,
<usg type="geo">U.S.</usg>
<pron>ˌmækəˈneɪʃ(ə)n</pron>,
<pron>ˌmæʃəˈneɪʃ(ə)n</pron>
<pron>-z</pron>
</form>
</form>
<note resp="schwa">Imported into WordList "One Thousand" on <date>2004-02-26 19:02:22</date></note> <,
</superEntry>
```

Figure 4.11: Entry View Window

into a plus sign. The editor tab also contains an option to enable word wrapping in the editor. On the “Insert Resp. Note” command, the editor will also insert a <`note`> element into entries, with the responsibility assigned to the value given in the “Responsibility code” text box. Finally, there is a button to choose the default font in the editor.

The second tab on the options dialog controls how the search window operates (Figure 4.13). The first option controls whether the search uses a plain string search or a regular expression. If a plain string search is specified, searching for “`.*tion$`” would look for that string *exactly*: the period, asterisk, letters, and dollar sign. If a regular expression search is specified, the string “`.*tion$`” would look for “tion” at the end of the headword. The second option specifies whether the search window should sample the results. Because some common string may return an overwhelming number of results, sampling

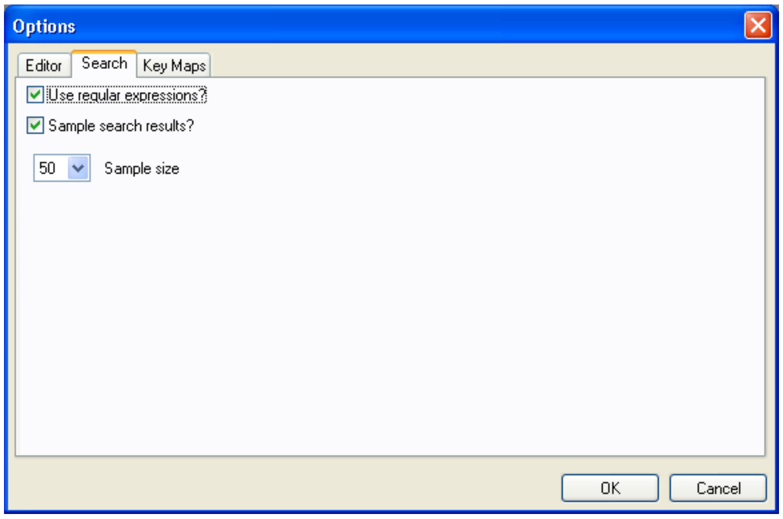


☑ Folding
☐ Word wrap
Responsibility code (initials): ERR

Figure 4.12: General Options

the results ensures that a random sample of the results is displayed, so the user can get a better, more representative set of the search results. If the option to show a random sample is enabled, the “Sample size” drop-down control allows the user to choose the size of the sample.

The third and final tab on the options dialog manages the keyboard mappings available to enter phonetic characters in the editor (Figure 4.14). The box on the left side of the window lists the currently defined set of key mappings. At the bottom-left, the current file containing the key mapping definitions is shown. The file used can be changed with the “Load” button, and the current set of mappings can be saved with the “Save” button. The mappings are not saved by default, so if any changes have been made to the mappings, they need to be saved explicitly. New mappings can be defined by clicking in the text box above the “Key” column and pressing the keyboard shortcut to define. Then, when the user navigates the built-in Unicode browser on the right side of the window and double-clicks the character or characters that the shortcut keys should map to, these characters will



☑ Use regular expressions?

☑ Sample search results?

50 Sample size

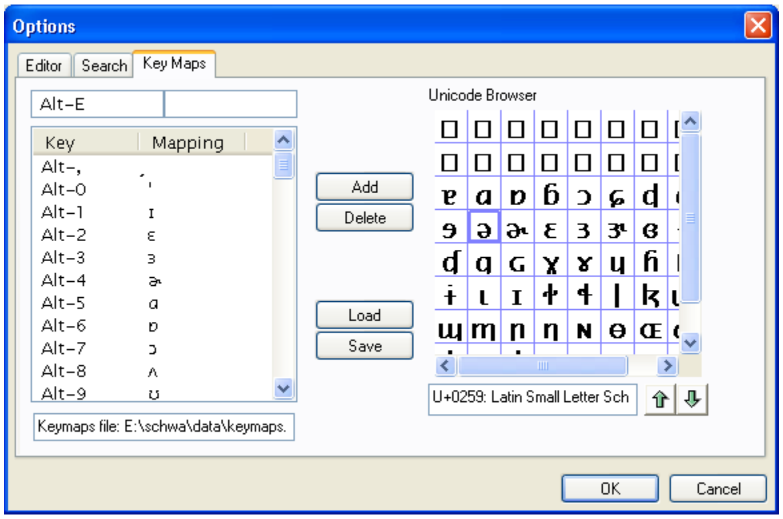
Figure 4.13: Search Options

appear in the text box above the “Mapping” column. Pressing the “Add” button adds the contents of those two text boxes into the list of defined key mappings. This combination of the text boxes and the Unicode browser provide a user-friendly way to define and manage keyboard shortcut mappings.

These windows provide the basic visual blocks of the Schwa application and the main means of allowing the user to interact with the database and entries. The next section examines these GUI elements in action.

4.4 Schwa in Use

This section contains information about using Schwa. It covers basic information like managing databases to more complex tasks like creating a new tool or data transfer filter.



Alt–E

Key	Mapping
Alt–,	ˏ
Alt–0	ˈ
Alt–1	ɪ
Alt–2	ɛ
Alt–3	ɜ
Alt–4	ɚ
Alt–5	ɑ
Alt–6	ɒ
Alt–7	ɔ
Alt–8	ʌ
Alt–9	ʊ

Keymaps file: E:\schwa\data\keymaps.

U+0259: Latin Small Letter Sch

Figure 4.14: Key Maps Options

4.4.1 Working with Databases

Some of the most basic tasks in working with Schwa involve managing the databases themselves. These examples show how to perform basic operations on the databases: creating, opening, and closing.

Creating and Opening Databases

Creating and opening databases involves the same steps. When the user selects the "File → Open. . . " menu item, Schwa displays a standard "Browse for Folder" dialog box. From here, the user can select an existing directory or create a new directory and select it. Since Schwa's databases are comprised of multiple files, they are identified by the directory that contains those files, and since the databases have a core set of basic files that are named identically across databases, each database must be in its own directory.

Once the directory has been selected, Schwa creates the basic database structure if necessary and opens the database on a blank editing window.

Closing Databases

Once a database has been opened, the user can close it by selecting the “File → Close” menu item. Of course, exiting Schwa with the “File → Exit” menu item or the “Exit” toolbar button also closes the current database.

4.4.2 Transferring Data

Once the database has been created, typically users will want to import word lists and pronunciations from an existing source. Although Schwa can create entries from scratch, repeatedly doing so would be onerous, to say the least. Instead, the user will receive the entries and pronunciations in a format that Schwa understands, verify that the data format is correct, and import the data. After editing, the user will export it back into the original format and send the changes back to the original source or to another source. With the appropriate export filters, someone writing dictionary pronunciations can export entries that originally were from a variety of sources, thus leveraging work done for one project or dictionary in creating a number of other dictionaries.

If a transfer filter does not exist for the data format, the user may need to create a new filter. Although not as easy other tasks in Schwa, if the user has a comfortable grasp of Python—or can find someone who has such a grasp—she can write a new transfer filter and integrate it into Schwa.

Verifying Data

Before importing data, verifying it is always a good idea. Although the database will not be corrupted if importing data fails, the entries that were added before the failure will remain in the database and will be duplicated after the user corrects the data and imports it successfully. Therefore, verifying data is always a good idea.

To perform this verification, the user selects the “File → Verify Import Data. . . ” menu item, which starts the verify wizard. First, the user selects the appropriate data format from the drop-down list. Second, she selects a file name to import the data from. Finally, she presses the “Finish” button. The wizard displays a progress dialog and begins verifying the data. If there is a problem, Schwa displays a message showing what error information it has and exits the verification process. If there are no errors, Schwa displays a dialog box stating this.

Importing Data

After the data has been successfully verified, it is safe to import. The import process is very similar to the verification process. First, the user selects the “File → Import. . . ” menu item. This starts the import wizard (Figure 4.15). When prompted, the user selects the output format from a drop-down box, a file name to import the data from, and a word list or enters a name for a new word list to import the new data into, and finally, she presses the “Finish” button. The wizard displays a progress dialog and begins importing the data. If there is a problem, Schwa displays information about the error and stops importing the data.

If there are conflicts between the existing data and the new data—that is, if there is already an entry for one of the headwords the wizard is trying to import—Schwa displays a dialog box offering to create another word list that contains the conflicts. This makes it easy to walk through the conflicting entries and to integrate the new data. Finally, Schwa displays a dialog box providing some statistics about the data that has been imported.

Exporting Data

Exporting data is essentially the opposite of importing it: instead of adding words to the database from a file; words are added to a file from the database. To export data, the user

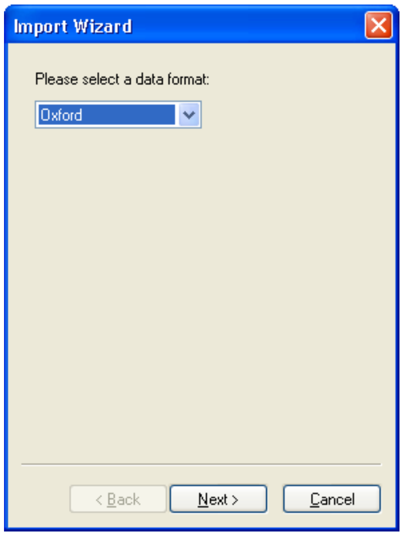


Figure 4.15: Import Wizard

selects the “File → Export. . . ” menu item. When prompted, the user selects the word list to export, a data format, and a file name to export to. Finally, she presses the “Finish” button. While exporting the data, Schwa displays a progress dialog, and when finished, it displays a status dialog giving statistics about the exported data.

Writing a New Data Transfer Filter

Although more difficult than using an existing format, writing a new data transfer filter is relatively easy. The most difficult part is writing the Python code actually to perform the data transfers. This section illustrates how to write a data transfer filter for Schwa. The format itself will be relatively simple: the data will be in a text file, with one entry per line. Each line contains a headword and one or more pronunciations, separated from each other by a forward slash (`/`). The pronunciations are represented using the Oxford University Press encoding. For example, Figure 4.16 shows several entries containing data taken from

```
bristly/"brIsli/"brIs@li
folding/"foUldIN
mackerel/"m{k(@)r@l
rationalize/"r{Sn@%laIz/"r{S@n@"laIz
```

Figure 4.16: Sample Data for a Simple Data Format

Merriam-Webster's Collegiate Dictionary, 11th edition (however, the pronunciations are represented in IPA). Figure 4.17 shows these entries after they have been imported.

The first step is to create a module that implements the interface specified in `schwalib.transfer.core.IDataTransfer`. This code will be in a module named `simpledata` (Appendix D). This class provides Schwa a single access point to create the import and export classes for the simple data format.

The class to import the data is called `SimpleImporter`. This class must be a subclass of `TaskThread` and conform to all the conventions of that class. `Simple-Importer` first reads all the data from the source and splits it into lines. It sets its `total` attribute to equal the number of lines. Then, in the `perform()` method, it processes each line, incrementing its `count` property, checking its `stop` property, and calling `pause()` as appropriate.

The class to export the data is called `SimpleExporter`. This class is also a subclass of `TaskThread`. `SimpleExporter` first sets its `total` attribute to the number of entries in the word list. Then, in the `perform()` method, it process each entry by encoding the headword and pronunciations appropriately and writing them to the destination file.

That is all the code necessary to create the import filter. A complete listing is given in Appendix D. The last step is to integrate the filter into Schwa by telling it where the new

```
<entry>
  <form>
    <orth> bristly </orth>
    <pron> &#x02c8;br&#x026a;sli </pron>
    <pron> &#x02c8;br&#x026a;s&#x0259;li </pron>
  </form>
</entry>
<entry>
  <form>
    <orth> folding </orth>
    <pron> &#x02c8;fo&#x028a;ld&#x026a;&#x014b; </pron>
  </form>
</entry>
<entry>
  <form>
    <orth> mackerel </orth>
    <pron> &#x02c8;m&#x00e6;k(&#x0259;)r&#x0259;l </pron>
  </form>
</entry>
<entry>
  <form>
    <orth> rationalize </orth>
    <pron>
      &#x02c8;r&#x00e6;&#x0283;n&#x0259;&#x02cc;la&#x026a;z
    </pron>
    <pron>
      &#x02c8;r&#x00e6;&#x0283;&#x0259;n&#x0259;
      &#x02cc;la&#x026a;z
    </pron>
  </form>
</entry>
```

Figure 4.17: Sample Data for a Simple Data Format, Imported

```
[Formats]
Count = 1
Format0 = SimpleFormat

[SimpleFormat]
Name = Simple
Class = simpledata.SimpleDataTransfer
```

Figure 4.18: Options to Integrate Simple Data Format into Schwa

data format is located. This is done by adding a few lines to Schwa's options file, which is located at `%UserProfile%\Application Data\schwalib\Schwa.ini`. On Windows 2000 and XP, `%UserProfile%` will usually be something like `C:\-Documents and Settings\`*`User Name`*. The sections and options to add are listed in Figure 4.18. The section "Formats" specifies how many formats have been added and their option's section names. The section "SimpleFormat" specifies the new format's name and the class to use in creating the transfer objects.

In the end, writing new data filters and integrating them into Schwa is not a mindless process, but it is not wholly onerous, either. Schwa and `schwalib` provide the frameworks necessary to do most of the work, and the programmer can focus on writing the code to transfer the data.

4.4.3 Searching

One of the more powerful features of Schwa is its ability to search the database's entries by their orthographic forms. Schwa provides a couple of different ways of performing the searches and displaying the results. These features are particularly important because they help to ensure that all the pronunciations in the database are consistent, which is important for good lexicographical practice.

A Plain Search

Plain searches compare a simple string of characters against the orthographic forms in the database. This comparison is case sensitive, so “A” would not match “a”. Although not as powerful as using regular expressions, this search is faster, especially over large databases.

To perform a plain search, first the user needs to ensure that the “Use regular expressions” option has been turned off. To do this, she selects “Edit → Options. . . ” and the “Search” tab. On that tab, she should make sure there is no check beside “Use regular expressions?”

Once the user is certain that regular expression searches have been disabled and a database is open, she can open the search window one of two ways: first, by selecting the “View → New Search” menu item; second, by pressing the “Search” toolbar button. The search window that appears will have a text box stretching across the top of it. The user enters the string to search for and presses the “Search” button.

After searching the database, Schwa displays the results in the two-column list that fills most of the search window. The user can change the width of the columns by clicking and dragging the right border of each column.

Using Regular Expression Searches

Regular expressions are a powerful tool in searching text; however, they can be slower than plan searches. Regular expressions are described in Appendix B. To use them in the search window, first the user must ensure that they are enabled by selecting the “Edit → Options. . . ” menu item and the “Search” tab. On that tab, the user should make sure there is a check beside “Use regular expressions?”

Once the user is certain that regular expression searches have been enabled and a database is open, she can open the search window one of two ways: first, by selecting the “View → New Search” menu item; second, by pressing the “Search” toolbar button. The search window that appears will have a text box stretching across the top of it. The user enters the string to search for in there and presses the “Search” button. For example, searching for “tion$” will find any orthographic forms that end with the letters “tion”: of *convention* and *conventionalize*, only the first will be displayed in the results list.

After searching the database, Schwa displays the results in a two-column list that fills most of the search window. The user can change the width of the columns by clicking and dragging the right border of each column.

SAMPLING SEARCH RESULTS

Many searches for common terms can return an overwhelming number of results. For example, the plain search for “tion” returned over 3,000 results when run on a database of almost 100,000 entries. The easy way for the user to manage that amount of information is to scan the first entries returned to see how the pronunciations in them are handled. Unfortunately, this does not necessarily provide a representative view of the data. To display a representative snapshot of the search results, Schwa can show only a random sample of the results.

Enabling and controlling random sampling is done from the options dialog. First, the user opens this dialog by selecting the “Edit → Options...” menu item and then the “Search” tab. The first option available is enabling random sampling. This is done by making sure there is a check mark next to “Sample search results?” If sampling is enabled, the user can also change the sample size using the “Sample size” drop-down box. The default sample size is 50, but there are a number of other options available.

If random sampling is enabled, the user can see the total number of results a search returned by watching the status bar. When the search is complete, the status bar will read something like, “Found 3188 results. Displaying 50.”

Creating a Word List from a Search

After a search has been completed, it can be saved to a word list. This is done by clicking the “Create a WordList” button and entering the name of the new word list. By default, this name is based upon the search terms. When saved as a word list *all* the results are saved, regardless of whether sampling is enabled.

4.4.4 Word Lists

Word lists are a powerful, flexible way of managing, grouping, and filtering data. They allow data to be grouped for easy editing and exporting. While defining word lists from searches has been covered already, there are other operations in managing and defining word lists.

Viewing the Entries in a Word List

If the user wishes to view the entries in a word list, she should enter an initial headword in the “Start list with” text box and select a word list from the “Word lists” drop-down box. The list of words down the left of the window will include only words from the word list.

Deleting Word Lists

Sometimes Schwa creates word lists that are temporary. For example, importing data can create a word list containing the conflicts between existing and imported data. After reconciling those conflicts, the user may wish to remove that word list. To do so, she should select the “Word Lists → Delete…” menu item. A small dialog box appears, listing the

```
//pron[regexp:test(text(), concat(pron:unichr(618), "z$"))]
```

Figure 4.19: Sample XPath Expression to Create a Word List

current set of word lists. By selecting one and pressing the “Delete” button, it will be deleted, although the entries in it will remain in the database. When the user is finished deleting word lists, she should click “OK”.

Defining a Word List from an XPath Expression

One way to define word lists is using XPath expressions (see Appendix C for a short description of XPath). This allows the user to create word lists based upon complex queries and upon the entire entry, not just the orthographic forms.

To create a word list from an XPath expression, the user should select the “Word Lists → From XPath...” menu item. Then, she should enter a name for the word list and an XPath expression. For example, the XPath expression in Figure 4.19 finds all the pronunciations that end in [ɪz]. This word list is a good example of using Schwa’s XPath extension functions. `regexp:test` tries to match the contents of a <`pron`> element against a pattern, and `pron:unichr` helps to create the pattern by inserting a Unicode character 618 (U+026a, ɪ).

While defining the word list, Schwa displays a progress dialog. Sometimes, particularly in large databases, creating a word list based upon an XPath expression can take a long time. However, the result is a group of entries that result may not be easy to gather any other way.

Defining a Word List from a List

The final way to create a word list is from a file. This file has a simple format: each line contains a headword, and all the headwords in the file are added to the word list. Defining a word list from a list in a file is similar to defining a word list from an XPath expression. The user selects the "Word Lists → From List. . . " menu item and enters the name of the word list to create and enters or browses to find the name of the file containing the list of words.

4.4.5 Editing

Although working with word lists is important for organizing the database, most of the user's time will be spent editing the entries.

Navigating the Database

The first step in editing an entry is selecting an entry to edit. First, the user must initialize the list of words on the left of the editing window. To do this, she either enters a word to start the list with in the "Start list with" text box or have the list automatically fill itself with the next 50 headwords. The user has the list of words fill itself automatically either by clicking the "Next 50" toolbar button, by selecting the "View → Next 50" menu item, or by pressing `F10`. If the user has the list of words fill automatically and the list is currently empty, the list starts itself with a reasonable default value of "0" (zero).

Once the list of words has been initialized or updated, the user can select an entry by clicking on it in the list. Also, there are a number of commands for navigating through the list by moving to the first or last entry or to the previous or next entry. These commands are listed in Table 4.3.

Creating and Deleting Entries

Although typically users will rely upon imported data and not add entries to the database individually, they can manually manage what entries are in the database. Entries are added using the "Edit → New Entry…" menu item. This displays a dialog asking for the key to store the entry under. Typically, this is the headword. Entries are removed from the database by selecting the entry in the list of words and selecting the "Edit → Delete Entry…" menu item.

Basic Editing Commands

Schwa responds to the standard Windows text editing commands: cut, copy, paste, delete, undo, redo, and select all. These can be accessed using the toolbar or the "Edit" menu, for cut, copy, paste, delete, and undo. Right clicking in the editor displays a context menu with all of these commands. Also, Schwa recognizes the standard keyboard shortcuts for these commands, unless they have been reassigned using Schwa's keyboard mappings.

Responsibility <note> Tags

Often in working with lexicographical databases, the editors need to leave a record of who made what changes, when, and why, especially if more than one person is maintaining the entries. In Schwa, this can be accomplished using the responsibility note feature. This inserts a <`note`> tag with an attribute giving the responsibility to the current user. Inside the <`note`> is a <`date`> element with the current date and time and room for the user to leave a comment (see Figure 4.20).

To use this feature, the user first needs to set her initials. This is done by opening the options dialog ("Edit → Options…"). The "Editor" tab has a text box labeled, "Responsibility code (initials)." The user should enter her initials there and click "OK."

```
<note resp="ERR">
  <date>2004-03-01 10:30:24</date>
  This is a comment.
</note>
```

Figure 4.20: An Example Responsibility <`note`> Element

Now the user can insert a <`note`> element by positioning the cursor where she wants the element and selecting the menu item “Edit → Insert Resp. Note”, clicking the “Resp. Note” toolbar button, or pressing `F10`.

4.4.6 Working with Pronunciations

Since the primary purpose of Schwa is to manage and edit pronunciations and phonetic data, the program provides a number of tools to facilitate those tasks. These include keyboard shortcut mappings in the editor and extended regular expressions in the search window. Working with pronunciations also involves making decisions about how to use XML to represent them most effectively, and this section also addresses some of the issues in representing dictionary pronunciations using XML.

Keyboard Shortcut Mappings

One of the more useful features of editing pronunciations using Schwa is its keyboard shortcut mappings. Schwa comes with a default set of mappings based upon those described by William A. Kretzschmar, Jr., and Rafal Konopka. These default mappings are listed in Appendix A, but they can also be modified using the “Key Maps” tab on the options dialog. Any changes made to the set of mappings are not saved to disk by default, so the user should be careful to save them if any changes have been made.

Long Form	**Short Form**	**Value**
`$consonants`	`$C`	consonant characters
`$vowels`	`$V`	vowel characters
`$suprasegmentals`	`$S`	suprasegmentals characters
`$tones`	`$T`	tone characters
`$diacritics`	`$D`	diacritic characters
`$other`	`$O`	other IPA characters

Table 4.4: Values for String Interpolation

To use the mappings that are currently defined, the user simply enters one of the keyboard shortcuts while in the main editor window. For example, if the default mappings are defined and the user presses `ALT-e`, a schwa character (ə) is entered into the editor.

Extended Regular Expressions

Another way that Schwa facilitates working with phonetics is by extending the regular expressions used in the search window. These regular expression extensions allow the user to insert classes of characters into a standard regular expression. In interpolation, the names are prefixed by a dollar sign (“`$`”). The names follow immediately (“`$vowels`”) or may be enclosed in curly braces (“`${vowel}`”), which is necessary when the name is not separated from other letters (e.g., “`${vowels}other characters`”). A dollar sign can be included in an interpolated string if it is not followed by a name. If it is followed by one, it can still be included if it is escaped with a backslash (“`\$vowels`”). The names that are available for interpolation are given in Table 4.4.

Regular expressions allow character classes to be incorporated into patterns. For instance, “`\s`” matches any whitespace character; “`\w`” matches any word character (as defined by the programming language C). The programmer can also define other, *ad hoc*

```
[aeiouy\xe6\xf8\u0153\u0250-\u0252\u0254
\u0258-\u025b\u025e\u0264\u0268\u026a\u026f
\u0275\u0276\u0289\u028a\u028c\u028f]
```

Figure 4.21: A Regular Expression to Match One IPA Vowel Character

```
[$vowel]
```

Figure 4.22: An Extended Regular Expression to Match One IPA Vowel Character

regular expression classes. The class “`[a-zA-Z]`” matches any set of English letters. Since the various classes of phonetic characters are not grouped together in Unicode, sets of phonetic characters have to be constructed manually. These search sets are verbose and difficult to construct, debug, and maintain. The values Schwa defines for interpolation can make constructing these regular expressions much easier. Figure 4.21 shows a regular expression class designed to search for IPA vowels symbols. Figure 4.22 shows an extended regular expression that performs the same search.

Representing Multiple Pronunciations and Variants

Beside editing and searching for pronunciations when using Schwa, the user must also represent them in XML. This involves making decisions about how to encode multiple and variant pronunciations in a single entry. There many ways to encode variant pronunciations using the TEI, and this section simply suggests two ways.

Probably the most important element in any encoding is *consistency*: if half the database handles variant pronunciations differently than the other half, computer tools will have trouble working with the data. For example, if the user writes an XPath expression to search the pronunciations that are encoded one way, it may not work properly when searching pronunciations that are encoded differently.

```
<form>
  <orth> ... </orth>
  <pron> ... </pron>
  <form type="variant">
    <pron> ... </pron>
  </form>
</form>
```

Figure 4.23: Encoding a Variant Pronunciation Using the `type` Attribute

In the TEI dictionary subset, the primary tag for working with variants is the <`form`> tag. This not only contains all the spoken and written forms of a word, but it can also group them by usage. One way to indicate a variant form is to use the `type` attribute with the value “variant” on the form tag. Figure 4.23 shows a framework for such an encoding scheme.

Another, more informative way to encode variant pronunciations is to use the <`usg`> element in conjunction with nested <`form`> elements. The <`usg`> element has a `type` attribute that specifies what kind of variation the usage represents, and the content of the element specifies the details of the usage. For example, “<`usg type="time"`> `archaic` </`usg`>” indicates that the usage is archaic; “<`usg type="register"`> `slang` </`usg`>” indicates that the usage is slang. The TEI suggests some values for the `type` attribute and the content of the <`usg`> element, which are listed on page 63. Of course, other attribute values and content are always possible.

The data imported using the Oxford format uses the <`usg`> element extensively. For example, Figure 4.24 shows the *Oxford Dictionary of Pronunciation*’s entry for *variant*. The British and American pronunciations are each grouped together, and the <`form`>

```
<entry>
  <form>
    <orth> variant </orth>
    <form>
      <usg type="geo"> Br </usg>
      <pron>
        &#x02c8;v&#x025b;&#x02d0;r&#x026a;&#x0259;nt
      </pron>,
      <pron extent="suff"> -s </pron>
    </form>
    <form>
      <usg type="geo"> Am </usg>
      <pron>
        &#x02c8;v&#x025b;ri&#x0259;nt
      </pron>,
      <pron extent="suff"> -s </pron>
    </form>
  </form>
</entry>
```

Figure 4.24: *ODP* Entry for *Variant*

element for each has a <`usg`> element identifying its usage as being geographically determined and with its content specifying where.

4.4.7 Other Tasks

Aside from managing data, editing, and working with pronunciations, Schwa also provides other features, which contribute to the application's overall purpose of managing lexicographical pronunciation data; however, they do not directly apply to any particular aspect of the program. Instead, these features are general and can be used in a number of areas of Schwa.

WRITING A TOOL

First, Schwa gives the user the ability to write small Python scripts that are automatically associated with a menu item. These scripts are stored in `%UserProfile%\Application Data\schwalib\Tools`. The simplest way for the user to find this directory is to use the default tool provided with Schwa, "Open Tools Directory," which opens Windows Explorer to this directory.

The menu structure of the "Tools" menu mirrors the directory structure of the `Tools` directory. Any Python files—that is, files with a ".py" extension—are automatically added to the menu when Schwa starts up. The menu item reflects the script's file name, without the extension.

Figure 4.25 provides an example of a short, useful tool. This tool takes the current entry and adds a `<note>` element. Unlike the "Resp. Note" feature, this tool inserts the complete text for the note, including the responsibility attribute, the date, and content saying, "This entry was last modified on. . . ."

This script is placed in the `Tools` directory and named "Insert Last-Modified Note.py". First, the script needs to access the edit window. The tools scripts have one globally defined variable: `frame`, which is the current instance of `schwalib.gui.frame.SchwaFrame`. The current edit window is referenced in the frame's `editChild` attribute. The responsibility initials are available from the `frame.resp_initials`. Next, the note is constructed, including an end tag for the `<superEntry>` element. This is included so the note can be inserted into the entry by replacing the existing end tag with the note element plus an end tag. Finally, the data is taken from the editor, the substitution made so that the `<note>` element is inserted, and the editor's text is updated.

```
from ericr.ml.utils import escape, quoteattr
from schwalib.utils.dt import DateTime

# get the editor
editor = frame.editChild.editor
# get the information
resp = frame.resp_initials
date = DateTime.today()

# construct the note
note = (
    u'<note resp=%s>'
    u'This entry was last modified on '
    u'<date>%s</date>'
    u'</note>\n'
    u'</superEntry>'
    ) % (quoteattr(resp), escape(str(date)))

# insert the text and enter it into the editor
data = editor.GetText()
new_data = data.replace(u'</superEntry>', note)
editor.SetText( new_data )
```

Figure 4.25: An Example Tool: “Insert Last-Modified Note.py”

Before actually being used, this tool should be improved by checking to see if an edit window is actually open and by making the error handling more robust, among other things.

GETTING HELP

Of course, any application needs information on how to use it. Schwa ships with a standard Windows help file that is available by pressing the `F1` key or selecting the “Help → Contents. . . ” menu item.

4.5 In the End

Schwa brings together a number of technologies into an application that is easy to use, yet able to manage large amounts of lexicographical pronunciations. Using it, someone writing dictionary pronunciations can maintain a large database for a variety of publishers. Moreover, Schwa is fairly easy to extend to handle more data transfer formats and to add small, but useful, functions.

CHAPTER 5

CONCLUSION

It occurred to me then that computers increase a lexicographer's awareness of what he knows that he does not know that he knows.

— *Jane Robinson (IBM Research)*

Having completed this application, it is worthwhile to ask how it helps users accomplish the tasks it has been designed for and what effect it has on those tasks themselves. All tools change the tasks they help accomplish, whether at the fundamental level of simply making the task possible or the more subtle level of making some—but not other—aspects of the task easier, and thereby changing the way the task is approached.

Of course, there are a number of tasks that Schwa does not facilitate. It is not a general-purpose application for computational lexicography. It does not include the corpus tools necessary for writing definitions, the verification tools for editing, or the layout and printing tools for publishing. It also does not make the task of determining pronunciations easier. It does not try to look up pronunciations in data, extract them from sound files, or provide a portal to linguistic research and resources.

That said, there are also a number of aspects of writing dictionary pronunciations that Schwa does make easier. It helps in working with IPA by providing an easy way to enter IPA characters. This has a number of advantages. First, Unicode IPA is commonly used, and it has better phonetic resolution than most other transcription systems. Because

Unicode is a standard for text encoding, transferring data between Schwa and other applications—whether through files or other system resources such as the clipboard—is significantly easier. Also, because IPA is a standard phonetic transcription, using existing linguistic research and resources that also use Unicode and IPA is possible. Moreover, since IPA can represent pronunciations at a lower resolution than the respelling systems that dictionaries use, it can more easily serve as an archive form for pronunciations, which would contain the full phonetic information and from which dictionary pronunciations would be derived.

A second area in which Schwa is helpful is in managing the infamously large amounts of information that dictionaries contain. There are several ways in which Schwa aids in this. First, the ability to import and export entries and pronunciations from and to diverse sources means that a user can incorporate pronunciations created for one dictionary or project into another one. This can significantly cut down on the amount of work the writer must do. Also, the user interface provides a number of tools for working with the data, such as word lists, which provide a means to break the data up into manageable amounts.

Third, Schwa's ability to import and export XML means that a wide variety of standard XML tools available can be used to process the data, if necessary. This can be used to prepare the information for publishing, to make sweeping changes to the data, or to perform a number of other tasks.

Finally, and more abstractly, Schwa reminds the user of the new reality of dictionaries: that they are in the early stages of making the transition into an electronic format. Nothing that makes that trip is entirely what it was, and what dictionaries will look like on the other side is anyone's guess. As John Simpson says, the dictionary is moving from "text to dictionary, back to text, and on to bibliography database, picture, sound, graphical analysis, or whatever" (12). Schwa, through its electronic nature and ability to deal with the dictionaries as database and therefore as immanently malleable, emphasizes the journey that lexicography is still in the beginning stages of: a piece of reference and bibliographic technology that has not significantly changed for several hundred years, but is beginning to undergo a radical metamorphosis.

However, Schwa's life cycle is just beginning. In the process of getting it this far, I have realized there are a number of areas in which it could be improved and a number of directions in which it can grow. One aspect of Schwa that needs improvement concerns the data management features. Although its current facilities for this are good, they could certainly be improved. Specifically, more tools, such as more search and verification mechanisms, for working with pronunciations are needed. Also, currently the ways that word lists can be defined are limited. New ways to define them based upon other criteria, for example, using a drag-and-drop user interface, would be helpful.

Another aspect of Schwa that needs improvement relates to features for extensibility and customization. Primarily, this will involve a better architecture for tools, automation, and customization. One possibility is to use a Command pattern (Gamma, et al., 233). In this, objects that perform an operation are defined. These can be either built-in, for the standard Schwa operations, or defined in extension modules. Either kind could be bound to menus and tool bars, by the program or by the user. This also allows the key mapping mechanism to be extended. As they are implemented now, key maps can only insert text. By having key maps trigger command objects, however, they could be much more versatile. Thus, extensions could define new commands and either bind them to key maps themselves or allow the user to do so. Then, after being initially created, commands could be stored in a file using Python's serialization facilities. Menus, toolbars, and key mappings would then be recreated by reading the command file back in.

Another option for improving Schwa's extensibility would be the creation of a customized scripting language. This could provide access to the command objects to implement basic functionality. It could also include features such as pronunciation-extended regular expressions and built-in XPath support for working with entries. This would create a programming language designed to facilitate working with the data in Schwa databases.

One important consideration in all these changes is that none of them entail making backward-incompatible changes to the databases. Thus, this new version of Schwa, which would in many ways be radically different than the one implemented here, could access the databases created by the current version of Schwa. This is an important feature for maintaining consistency for the user base.

In the final analysis, although Schwa still has room to grow, it is nevertheless usable and useful. It represents a practical tool for computational lexicography, and lexicographical pronunciations in particular. As such, it stands as part of a journey, beginning with Cawdrey’s *A Table Alphabeticall*, extending through Johnson, Webster, and the *OED*, and continuing through the *OED2* and `http://www.oed.com` and into the uncharted future of electronic lexicography.

Bibliography

Algeo, John. “The Emperor’s New Clothes: The Second Edition of the Society’s Dictionary.” *Transactions of the Philological Society* 88.2 (1990): 131–50.

American Heritage College Dictionary. Ed. Joseph P. Pickett. 4th ed. New York: Houghton Mifflin, 2002.

Artin, Edward. “Dictionary Treatment of Pronunciations: General.” *Lexicography in English*. Eds. Raven I. McDavid and Audrey R. Duckert. Annals of the New York Academy of Sciences. 211. New York: NY Academy of Sciences, 1973. 125–28.

Benson, M. “Collocations and General Purpose Dictionaries.” *International Journal of Lexicography* 3.1 (1990): 23–35.

Bladon, Anthony, et al. “Session 4: American English and the International Phonetic Alphabet: The International Phonetic Association Reacts.” *Conference Papers on American English and the International Phonetic Alphabet*. Ed. Arthur Bronstein. Publication of the American Dialect Society. 80. Tuscaloosa: U of AL P, 1998. 123–26.

Bray, Tim, et al., eds. *Extensible Markup Language (XML) 1.0*. 2nd ed. 6 Oct. 2000. 11 Nov. 2003. <`http://www.w3.org/TR/REC-xml`>.

Bray, Tim, Dave Hollander, and Andrew Layman. *Namespaces in XML*. 14 Jan. 1999. 24 Feb. 2004. <`http://www.w3.org/TR/1999/REC-xml-names-19990114`>.

Bronstein, Arthur J. “The Development of Pronunciation in English Language Dictionaries.” *Studies in the Pronunciation of English: A Commemorative Volume in*

Honour of A. C. Gimson. Ed. Susan Ramsaran. New York: Routledge, 1990. 137–52.

Clark, James, and Steve DeRose. *XML Path Language (XPath) 1.0*. 16 Nov. 1999. 24 Feb. 2004. <`http://www.w3.org/TR/1999/REC-xpath-19991116`>.

Congleton, J. E. “Pronunciation in Johnson’s *Dictionary*.” *Papers on Lexicography in Honor of Warren N. Cordell*. Eds. J. E. Congleton, Edward Gates and Donald Hobar. Terre Haute, IN: Dictionary Society of North American, 1979. 59–81.

Dauer, Rebecca M. “IPA Characters on an IBM Compatible Personal Computer: Better Letter Setter Reviewed.” *Journal of the International Phonetic Association* 18.1 (1988): 41–43.

EXSLT: Regular Expressions. 30 June 2001. 10 Dec. 2002. <`http://www.exslt.org/regexp/index.html`>.

Fillmore, Charles J., and B. T. S. Atkins. “Starting Where the Dictionaries Stop: The Challenge of Corpus Lexicography.” *Computational Approaches to the Lexicon*. Eds. B. T. S. Atkins and A. Zampoli. Oxford: Clarendon, 1994. 349–93.

Gimson, A. C. “Phonology and the Lexicographer.” *Lexicography in English*. Eds. Raven I. McDavid and Audrey R. Duckert. Annals of the New York Academy of Sciences. 211. New York: NY Academy of Sciences, 1973. 115–21.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

GNOME: The Free Linux Desktop Project. 2003. 3 Mar. 2004. <`http://www.gnome.org`>.

Harold, Elliotte Rusty. *XML Bible*. 2nd ed. New York: Hungry Minds, 2001.

Hulbert, James R. *Dictionaries: British and American*. Rev. Ed. London: Andre Deutsch, 1968.

Hultin, N. C., and H. M. Logan. “The New Oxford English Dictionary Project at Waterloo.” *Dictionaries: Journal of the Dictionary Society of North America* 6 (1984): 128, 183–98.

Ide, Nancy, Nancy, and Jean Véronis. “Encoding Dictionaries.” *Computers and the Humanities* 29 (1995): 167–79.

International Phonetic Association. *Handbook of the International Phonetic Association: A Guide to the Use of the International Phonetic Alphabet*. New York: Cambridge UP, 1999.

Jassem, Wiktor, and Piotra Łobacz. “IPA Phonemic Transcription Using an IBM PC and Compatibles.” *Journal of the International Phonetic Association* 19.1 (1989): 16–23.

Johnson, Samuel, ed. *A Dictionary of the English Language*. 1755. 2 vols. New York: AMS, 1967.

Jost, David, Beth Rowen, and Susan Schwartz. “The Use of On-Line Databases in Neology.” *Dictionaries: Journal of the Dictionary Society of North America* 16 (1995): 16–18.

Kennedy, Graeme. *An Introduction to Corpus Linguistics*. New York: Addison Wesley Longman, 1998.

Kenyon, J. S. “Pronunciation.” *Webster’s New International Dictionary of the English Language*. 2nd ed. Ed. William Allan Neilson. Springfield, MA: Merriam, 1934. xxii–lxxviii.

Krapp, George Philip. *The English Language in America*. Vol. 1. New York: MLA, 1925.

Kretzschmar, William A., Jr. “Phonetic Output and Display.” *Journal of English Linguistics* 22.1 (1989): 47–53.

Kretzschmar, William A., Jr., and Rafal Konopka. “Management of Linguistic Databases.” *Journal of English Linguistics* 24.1 (1996): 61–70.

Kuhn, Sherman M. “Making the *Middle English Dictionary*.” *Dictionaries: Journal of the Dictionary Society of North American*. 4 (1982): 14–41.

Landau, Sidney, I. *Dictionaries: The Art and Craft of Lexicography*. 2nd ed. New York: Cambridge UP, 2001.

—. “Session 3: Should We Change the Ways We Represent Pronunciation in American English Dictionaries?” *Conference Papers on American English and the International Phonetic Alphabet*. Ed. Arthur Bronstein. Publication of the American Dialect Society. 80. Tuscaloosa: U of AL P, 1998. 117–20.

Le Hors, Arnaud, et al. *Document Object Model (DOM) Level 2 Core Specification 1.0*. 13 Nov. 2000. 11 Nov. 2003. <`http://www.w3.org/TR/DOM-Level-2-Core/`>.

libiconv. 1998. GNU Project. 11 Nov. 2003. <`http://www.gnu.org/software/libiconv/`>.

Logan, Harry M. “Electronic Lexicography.” *Computers and the Humanities* 25 (1991): 351–61.

Martelli, Alex, and David Ascher. *Python Cookbook*. Sebastopol, CA: O’Reilly, 2002.

McDavid, Raven I., Jr. “The Social Role of the Dictionary.” *Papers on Lexicography in Honor of Warren N. Cordell*. Eds. J. E. Congleton, Edward Gates and Donald Hobar. Terre Haute, IN: Dictionary Society of North American, 1979. 17–28.

Merriam-Webster’s Collegiate Dictionary. 11th ed. Ed. Frederick C. Mish. Springfield, MA: Merriam-Webster, 1995.

Metalleus Archive. 2003. 11 Nov. 2003. <`http://www.umich.edu/˜archive/linguistics/texts/papers/metalleus`>.

Miller, Herman. *The Language Page*. 2003. 11 Nov. 2003. <`http://www.io.com/˜hmiller/lang/`>.

Murray, K. M. Elisabeth. *Caught in the Web of Words: James A. H. Murray and the* Oxford English Dictionary. New Haven, CT: Yale UP, 1977.

MySQL. 2003. MySQL AB. 11 Nov. 2003. <`http://www.mysql.com/`>.

Neufeldt, Victoria. “Session 3: Should We Change the Ways We Represent Pronunciation in American English Dictionaries?” *Conference Papers on American English and the International Phonetic Alphabet*. Ed. Arthur Bronstein. Publication of the American Dialect Society. 80. Tuscaloosa: U of AL P, 1998. 109–14.

OED On-line. 2000. Eds. John Simpson and Edmund Weiner. Oxford University Press. 11 Nov. 2003. <`http://www.oed.com/`>.

Peake, Mervyn. *Titus Groan*. *The Gormenghast Trilogy*. Woodstock, NY: Overlook, 1988. 7–396.

Pearsons, Enid. “Session 3: Should We Change the Ways We Represent Pronunciation in American English Dictionaries?” *Conference Papers on American English and the International Phonetic Alphabet*. Ed. Arthur Bronstein. Publication of the American Dialect Society. 80. Tuscaloosa: U of AL P, 1998. 114–17.

Python Programming Language. 2003. Python Software Foundation. 11 Nov. 2003. <`http://www.python.org/`>.

Quirk, Randolph. “The Social Impact of Dictionaries in the UK.” *Lexicography in English*. Eds. Raven I. McDavid and Audrey R. Duckert. Annals of the New York Academy of Sciences. 211. New York: NY Academy of Sciences, 1973. 76–88.

Read, Allen Walker. “The Social Impact of Dictionaries in the United States.” *Lexicography in English*. Eds. Raven I. McDavid and Audrey R. Duckert. Annals of the New York Academy of Sciences. 211. New York: NY Academy of Sciences, 1973. 69–75.

—. “The Theoretical Basis for Determining Pronunciations in Dictionaries.” *Dictionaries: Journal of the Dictionary Society of North America* 4 (1982): 87–96.

Robinson, Jane. “Discussion.” *Lexicography in English*. Eds. Raven I. McDavid and Audrey R. Duckert. Annals of the New York Academy of Sciences. 211. New York: NY Academy of Sciences, 1973. 298–301.

Russell, Jordan. *Inno Setup*. 25 Feb. 2004. <`http://www.jrsoftware.org/isinfo.php`>.

Sedelow, Sally Yeates. “Computational Lexicography.” *Computers and the Humanities* 19 (1985): 97–101.

Sheldon, Esther K. “Walker’s Influence on the Pronunciation of English.” *PMLA* 62 (1947): 130–46.

Simpson, John. “The Revolution in English Lexicography.” *Dictionaries: Journal of the Dictionary Society of American* 23 (2002): 1–15.

Simpson, John, and Edmund Weiner. “An On-line *OED*.” *English Today: The International Review of the English Language* 16.3 (2002): 12–19.

Sledd, James. “Dictionary Treatment of Pronunciation: Regional.” *Lexicography in English*. Eds. Raven I. McDavid and Audrey R. Duckert. Annals of the New York Academy of Sciences. 211. New York: NY Academy of Sciences, 1973. 134–38.

Sleepycat Software: Berkeley DB. 2003. Sleepycat Software. 11 Nov. 2003. <`http://www.sleepycat.com/`>.

Source for Java Technology. 2003. Sun Microsystems. 11 Nov. 2003. <`http://java.sun.com/`>.

Sperberg-McQueen, C. M., and Lou Burnard. *Guidelines for Electronic Text Encoding and Interchange.* May 1999. <`http://www.tei-c.org/Guidelines/index.htm`>.

Stubbs, Michael. *Words and Phrases: Corpus Studies of Lexical Semantics*. Oxford: Blackwell, 2001.

Tcl SourceForge Project. 2003. 11 Nov. 2003. <`http://tcl.sourceforge.net/`>.

Unicode Consortium. *Unicode Standard, Version 4.0.0*. Reading, MA: Addison-Wesley, 2003. 11 Nov. 2003. <`http://www.unicode.org/versions/Unicode4.0.0/`>.

Upton, Clive, William A. Kretzschmar, Jr., and Rafal Konopka. *Oxford Dictionary of Pronunciations*. Eds. Susan Wilkin and Judith Scott. New York: Oxford UP, 2001.

Urdang, Laurence. “A Lexicographer’s Adventures in Computing.” *Dictionaries: Journal of the Dictionary Society of America* 6 (1984): 150–65.

Veillard, Daniel. *The XML C Parser and Toolkit of Gnome: libxml*. 25 Feb. 2004. <`http://xmlsoft.org/`>.

Webster’s New International Dictionary of the English Language. 2nd ed. Ed. William Allan Neilson. Springfield, MA: Merriam, 1934.

Webster’s New World College Dictionary. 4th ed. Ed. Michael Agnes. New York: Macmillan, 1999.

Weiner, Edmund. “The New *Oxford English Dictionary*.” *Journal of English Linguistics* 18.1 (April 1985): 1–13.

Wells, John. “A New Pronunciation Preference Survey.” 1999. 13 Feb. 2004. <`http://www.phon.ucl.ac.uk/home/wells/poll98.htm`>.

“What Is Python?” *Python Programming Language*. 2003. Python Software Foundation. 24 Feb. 2004. <`http://www.python.org/doc/Summary.html`>.

Woolf, B. “The Null Object Pattern.” *Pattern Languages of Programming*. Sept. 1996. <`http://citeseer.ist.psu.edu/woolf96null.html`>.

wxWindows Home. 2003. Julian Smart, Anthemion Software. 11 Nov. 2003. <`http://www.wxwindows.org/`>.

Zawinksi, Jamie. “marginal hacks.” 2002. 11 Nov. 2003. <`http://www.jwz.org/hacks/marginal.html`>.

Appendix A

Schwa Key Macros

These are the default macros provided with the Schwa application. They are based on the macros described by William A. Kretzschmar, Jr., in “Phonetic Output and Display” and still used in the Linguistic Atlas of the United States and Canada for phonetic input.

`Control-0`	ˌ	`Alt-8`	ʌ
`Alt-0`	ˈ	`Control-9`	ʊ-
`Control-1`	ɨ	`Alt-9`	ʊ
`Alt-1`	ɪ	`Control-Down`	˅˂
`Control-2`	{ɛ}	`Alt-Down`	˅
`Alt-2`	ɛ	`Control-Left`	˄˃
`Control-3`	ə̆	`Alt-Left`	˂
`Alt-3`	ɜ	`Control-Right`	˅˃
`Control-4`	{ɚ}	`Alt-Right`	˃
`Alt-4`	ɚ	`Control-Up`	˄˂
`Control-5`	ɐ	`Alt-Up`	˄
`Alt-5`	ɑ	`Control-a`	æ̃
`Control-6`	ɑ̯	`Alt-a`	æ
`Alt-6`	ɒ	`Control-b`	ß̯
`Control-7`	{ɔ}	`Alt-b`	ß
`Alt-7`	ɔ	`Control-c`	˜
`Control-8`	ə̃	`Alt-c`	c̘

Control-comma	ˌ	Alt-o	̥
Alt-comma	̗	Control-q	ɹ
Control-d	{d}	Alt-q	ɾ
Alt-d	ð	Control-r	{ɾɺ̴̮}
Control-e	{ə}	Alt-r	ɾɺ̴̮
Alt-e	ə	Control-s	z̥
Control-f	i-	Alt-s	ʃ
Alt-f	̠	Control-semicolon	.
Control-g	Γ	Alt-semicolon	·
Alt-g	ʔ	Control-t	t̺
Control-h	{h}	Alt-t	t̬
Alt-h	ɦ	Control-u	ɟ
Control-i	{ɪ-^<}	Alt-u	ɺ
Alt-i	ɪ-^<	Control-v	̬
Control-j	{j}	Alt-v	ʊ
Alt-j	ʲ	Control-w	{w}
Control-k	ʞ	Alt-w	̑
Alt-k	k̟	Control-x	͡
Control-l	l̗	Alt-x	͜
Alt-l	ɬ	Control-y	ʎ
Control-m	ɯ	Alt-y	θ
Alt-m	ɱ	Control-z	{z}
Control-n	ɲ	Alt-z	ʒ
Alt-n	ŋ		
Control-o	o-		

Appendix B

Regular Expressions

Regular expressions are a powerful language for expressing text searches. They allow the user to search for things that plain string searches do not, for example, all the words that end in *y*.

There are many *dialects* of regular expressions. The one that Schwa uses is probably the most popular: the regular expression syntax of Perl. Python also uses it. There are many resources available for learning regular expressions. *Mastering Regular Expressions*, by Jeffrey Friedl, published by O'Reilly is a comprehensive guide to regular expressions. Also, the Python website has a HOWTO guide that is more detailed than this appendix buy less weighty than *Mastering Regular Expressions*.

The information presented here builds from the most simple constructs in regular expressions up to the more complex ones. Each section briefly describes the construct and provides an example of it.

Matching

The simplest regular expressions match a string of one character. For example, the expression "`a`" matches the character *a* wherever it may appear. Most characters can be represented this way. Special characters that have meaning in the regular expression can be matched by escaping the character with a backslash (\) before it (e.g., "`\.`" would match a period).

However, matching only one character is next to useless. Concatenating two single-character expressions match that string: “`ab`” matches the string *ab*; “`the`” matches the string *the*.

Regular expression can also match alternatives of characters by separating the alternatives with a pipe character (`|`). “`a|b`” matches either *a* or *b*. Likewise, “`x|y|z`” matches either *x*, *y*, or *z*.

The expressions so far have matched specific characters. Wild cards match any character. The only wild card in regular expressions is the period character (`.`). Thus the regular expression “`.`” matches *a*, *z*, or any other character, and “`a.e`” matches *ace*, *age*, or *ate*.

Regular expressions also include sets of characters by enclosing them in square brackets (`[]`). A set matches any character in the set. Thus, “`[aeiou]`” matches the standard set of English vowels.

Sets can also be defined by what they cannot contain. This is indicated by placing a caret (`^`) as the first character in the set. So “`[^aeiou]`” would match any character that is *not* a vowel. (While this set includes consonant characters, it also includes whitespace, punctuation, and a good deal more.)

There are also a number of predefined sets:

`\d`	Any digit (`0`–`9`).
`\D`	Any non-digit character; anything not in `\d`.
`\s`	Any whitespace character.
`\S`	Any non-whitespace character.
`\w`	Any alphanumeric character or an underscore.
`\W`	Any character that is neither alphanumeric nor an underscore.

Repetition

There are several modifiers that indicate that the preceding match occurs more than once or not at all:

- **`?`** The match occurs zero or one times: “`a?`”.
- **`+`** The match occurs one or more times: “`a+`”.
- **`*`** The match occurs zero or more times. A pattern like “`a.*b`” is often used to indicate that *a* and *b* are either adjacent or are separated by any number of undefined characters.

Anchors

There are also a number of *anchors*: expressions that match places in the string being searched, but not actual characters.

- **`^`** At the beginning of an expression, this matches the beginning of a string.
- **`$`** At the end of an expression, this matches the end of a string.
- **`\b`** This matches the edge of a word. That is, the character to one side of it is a word character (`\w`) while the character on the other side is a non-word character (`\W`).

Groups

Finally, regular expressions can be grouped using parentheses. This allows repetition modifiers (`?`, `+`, and `*`) to modifier a larger unit in the expression.

For example, “`a(bc)+`” matches *a* followed by one or more occurrences of *bc*: *abc*, *abcbc*, *abcbcbcbcbc*, but not *abbcc* or *abbc*.

APPENDIX C

XPATH

XPath is a language to query XML data structures and to return sets of elements that fulfill the conditions of the query. It allows the user to query on the names, attributes, and contents of nodes, as well as the overall structure of the XML document. XPath is used in Schwa to define new word lists. In defining word lists, if the query matches any node in the entry, that entry is included in the word list. This appendix provides a short introduction to XPath. A more complete introduction is available in Elliotte Rusty Harold's *XML Bible*, in the chapter on XSL Transformations, beginning on page 513, or the XPath Specification (Clark and DeRose).

However, this appendix gives a complete description of the extensions to the core XPath specification that Schwa also provides. These extensions facilitate working with regular expressions and pronunciations, and they are described below.

STANDARD QUERIES

Any element within an XML document can be found using an XPath location path. This path can be either relative to another element within the document or absolute from the root element. All XPath queries are started from the context of a node, and in Schwa this context node is always the root node. To find a child element of the context node, the query is simply the name of the child element: "`entry`" would find all the <`entry`> elements of the current node. When part of a query is matches, the context node becomes the matched node. Thus, to find all the <`form`> elements that are children of <`entry`>

elements, the query would be "entry/form". The slash (/) between the elements indicates a new subquery. A query that begins with a slash indicates that the initial context node should be the root element: "/entry/form" matches all <form> elements that are children of the root <entry> element, if the root element is, in fact, a <entry> element.

Unfortunately, neither of these queries would match anything in Schwa, since the root element for all entries is <superEntry>. To modify these queries to work in Schwa, the query must use the // subquery, which matches all the descendants of the current node or the current node itself. For example, "form//pron" matches all <pron> elements that are the descendants of a <form> element, whether there are any intervening elements or not. At the beginning of a query, // matches the root node or any descendants: "//entry/form" matches all the <form> elements that is a child of *any* <entry> element, wherever it occurs in the XML document.

To match an attribute, prefix the name of the attribute with an at sign (@). For example, "//usg/@type" matches that type attribute of any <usg> element that defines a type attribute.

Wild cards can also be used with either element or attribute names. The only wild card defined is the asterisk (*), which matches any name: "//form/*" matches all the children of a <form> element, and "//usg/@*" matches all the attributes defined on all the <usg> elements.

Even with wild cards, matching an attribute directly is less useful than testing for the content of an attribute or for other details of the current node. This is done using square brackets ([]). A boolean (true-or-false) expression goes inside the square brackets. For example, "//usg[@type='geo']" matches any <usg> elements that have a type attribute with the value of "geo". There are also a number of functions available for tests, such as text(), which returns the text value of the current node (the value of all the text in the current node and its children). An XPath reference would list the many functions available.

Extension Functions

Beside the standard functions, Schwa also makes available a number of extension functions. One of the most useful is the regular expression `test(string, pattern, flags)` defined by the EXSLT XPath extensions (*EXSLT*). This function takes a regular expression, such as is described in Appendix B. Because extension functions reside in a different namespace than the standard functions, they must use the namespace prefix assigned to them. In the case of the `test` function, this prefix is “`regexp:`”. This function takes three arguments:

string The string to match the regular expression against.

pattern The regular expression pattern.

flags This argument is optional. If provided, it is a string listing flags that change how the matching is done. Currently, the only flag recognized is “`i`”, which performs a case-insensitive match.

For example, “`//orth[regexp:test(text(), '.*tion$')]`” would match any <`orth`> elements whose text value ends in the string “tion”.

Schwa also provides a number of XPath extension functions to facilitate working with phonetic data under the “`pron:`” namespace prefix. These functions return strings of Unicode characters corresponding to sets of phonetic characters:

`pron:consonants()` This function returns all the Unicode IPA consonant characters.

`pron:diacritics()` This function returns all the Unicode IPA diacritic characters.

`pron:other()` This function returns all the Unicode IPA characters not returned by the other functions.

`pron:suprasegmentals()` This function returns all the Unicode IPA suprasegmental characters.

`pron:tones()` This function returns all the Unicode IPA tone characters.

`pron:vowels()` This function returns all the Unicode IPA vowel characters.

Another function, `pron:pron-sets(string)` performs interpolation, as described in the discussion on page 72 on the `Interpolator` class. The values that this function makes available for interpolation are the IPA phonetic character classes. Each class is available using two interpolation markers:

`$consonants, $C` These markers interpolate the IPA consonants.

`$diacritics, $D` These markers interpolate the IPA diacritics.

`$other, $O` These markers interpolate the IPA characters not covered by the other markers.

`$suprasegmentals, $S` These markers interpolate the IPA suprasegmentals.

`$tones, $T` These markers interpolate the IPA tone characters.

`$vowels, $V` These markers interpolate the IPA vowels.

A final XPath function, `pron:unichr(n)` takes a decimal character code number and returns a string containing the corresponding Unicode character. Figure C.1 shows some examples of using these XPath functions. Combined with the `regexp:test` extension function, these can be used to create powerful regular expressions that search phonetic data.

```
pron:consonants()
pron:vowels()
pron:tones()
pron:pron-sets("Consonant characters: $consonants")
pron:pron-sets("Vowel characters: $V")
concat("Here is a schwa: ", pron:unichr(601))
```

Figure C.1: Using XPath Extension Functions from `pron:`

Appendix D

Simple Data Format Filter

This is the code for the Python module `simpledata`, which acts as a data transfer filter for the simple data format described in “Writing a New DataTransfer Filter” on page 97.

```
from schwalib.transfer.core import IDataTransfer
from schwalib.task import TaskThread
from ericr.ml.utils import escape

class SimpleDataTransfer(IDataTransfer):
    def makeExporter(self, db, wl, dest):
        return SimpleExporter(db, wl, dest)

    def makeImporter(self, db, wl, src):
        return SimpleImporter(db, wl, src)

class SimpleImporter(TaskThread):
    def __init__(self, db, wl, src):
        self.db = db
        self.wl = wl
        self.src = src
        # read in the data as lines
        self.lines = list(src)
        # set the total to the number of lines
        self.total = len(self.lines)
        super(SimpleImporter, self).__init__()

    def perform(self):
        for line in self.lines:
            # stick to TaskThread’s conventions
            if self.stop:
                break
            self.pause()
            # process the line of data
            line = line.strip()
            data = line.split(’/’)
            headword = unicode(data.pop(0), ’latin-1’)
            data = [
                unicode(p, ’schwalib.oxford’)
                for p in data
                ]

            # put the data into XML
            buffer = [
                u’<?xml version="1.0" encoding="UTF-8"?>’,
                u’<superEntry><entry><form><orth>’,
                escape(headword),
                u’</orth>’
                ]
            buffer += [
                u’<pron>%s</pron>’ % escape(p)
                for p in data
                ]
            buffer.append(’</form></entry></superEntry>’)
            xml = u’’.join(buffer)
            xml = xml.encode(’utf-8’)

            # create the entry and put it into the db
            entry = self.db.Entry(entry=xml)
            key = entry.key
            self.db.entries[key] = entry
            self.wl.add(entry.key)

            # update the count
            self.count += 1

        self.db.wordlists[self.wl.name] = self.wl

class SimpleExporter(TaskThread):
    def __init__(self, db, wl, dest):
        self.db = db
        self.wl = wl
        self.dest = dest
        self.total = len(self.wl)
        super(SimpleExporter, self).__init__()
```

```
def perform(self):
    for key in self.wl:
        # stick to TaskThread’s conventions
        if self.stop:
            break
        self.pause()

        # get the entry
        entry = self.db.entries[key]

        # process the entry into a buffer
        buffer = []
        headwords = entry.doc.xpathEval2(’//orth’)
        headword = headwords[0].content
        headword = unicode(headword, ’utf-8’)
        buffer.append( headword.encode(’latin-1’) )
        for pron in entry.doc.xpathEval2(’//pron’):
            data = unicode(pron.content, ’utf-8’)
            data = data.encode(’schwalib.oxford’)
            buffer.append(data)

        # output the data to the file
        print >>self.dest, ’/’.join(buffer)
```