AMAR: A Computational Model of Autosegmental Phonology

by

Daniel M. Albro

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1994

Author ....
Department of Electrical Engineering and Computer Science
October 18, 1993

Certified by ....
Robert C. Berwick
Professor of Computer Science and Engineering and Computational Linguistics
Thesis Supervisor

Accepted by ....
Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

AMAR: A Computational Model of Autosegmental Phonology

by

Daniel M. Albro

Submitted to the Department of Electrical Engineering and Computer Science
on October 18, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis describes a computational system with which phonologists may describe a natural language in terms of autosegmental phonology, currently the most advanced theory pertaining to the sound systems of human languages. This system allows linguists to easily test autosegmental hypotheses against a large corpus of data. The system was designed primarily with tonal systems in mind, but also provides support for tree or feature matrix representation of phonemes (as in *The Sound Pattern of English*), as well as syllable structures and other aspects of phonological theory. Underspecification is allowed, and trees may be specified before, during, and after rule application. The association convention is automatically applied, and other principles such as the conjunctivity condition are supported. The method of representation was designed such that rules are designated in as close a fashion as possible to the existing conventions of autosegmental theory while adhering to a textual constraint for maximum portability.

Thesis Supervisor: Robert C. Berwick
Title: Professor of Computer Science and Engineering and Computational Linguistics

Acknowledgments

I would like to thank the following people for their contributions to this thesis:

Morris Halle and James Harris, for improving my grasp of the linguistic basis for the system, for many enlightening discussions of phonology, and for taking the time to see AMAR run;

Robert Berwick, for encouragement, tolerance of my tendency to constantly increase the scope of the project, the suggestions that led to AMAR's current form, the sources from which most of the examples in this paper came, and all manner of help;

Dave Baggett and Carl de Marcken, for advice, encouragement, proofreading, comic relief, and the ideas underlying the grand new theory of Astrophonology...

In addition, I would like to thank Dave Baggett for giving me the initial suggestion for this project;

Doug Jones, for much helpful advice and encouragement;

Kristi Jenson, for letting me use her radio;

Isabel Wang, one of my greatest friends, for encouragement, humor, and a place to stay while writing the thesis;

and finally, Juanita Vargas, for keeping my spirits up, for sitting beside me for hours, for flying up from Texas to help me out, and for everything else. I dedicate this thesis and the AMAR system to her.

Contents

1 Introduction **11**

2 Background **15**
2.1 KIMMO 15
2.2 The Delta Programming Language 21

3 Design and Overview **27**
3.1 Linguistic Overview 27
3.2 User Interface 28
3.2.1 Phoneme Specification 28
3.2.2 Tone Specification 32
3.2.3 Free Associates 34
3.2.4 Definitions 34
3.2.5 Rule Specification 34
3.2.6 Specifying Inputs 35
3.3 Program Design 38
3.3.1 Objects 38
3.3.2 Language Specification Parser 40
3.3.3 Input and Output 41
3.3.4 Matching 42
3.3.5 Application 43

4 Examples **47**
4.1 Simple Example 47
4.2 Bambara 49
4.3 Spanish 51
4.3.1 Matrix Model 51
4.3.2 Tree Model 53
4.3.3 Hypothetical Model 59
4.4 Arabic 64

5 Discussion **75**
5.1 Improvements and Extensions 75
5.2 Possible Uses for the System 77

A Full Grammar for Specification File **79**

B Examples Incompletely Specified in the Text **83**
B.1 Mende 83
B.2 Tagalog 84
B.3 Turkish 87

C Selected Code **91**

C.1 Objects 91
C.2 Application 111
C.3 Matching 137
C.4 Input/Output 145

List of Figures

1-1 Autosegmental Representation of a Tone 11
1-2 Mandarin Tone Shortening Rule 12

2-1 Turkish High Vowel Deletion Rule 16
2-2 Hyor Vowel Deletion Rule 16
2-3 Representation of High and Rising Tones in Autosegmental Notation 17
2-4 Mende High Tone Assimilation Rule 17
2-5 Mende Rising Tone Shortening Rule 17
2-6 Infixation Rule of Tagalog 18
2-7 Reduplication Rule of Tagalog 18
2-8 Nasal Coälescence Rule of Tagalog 18
2-9 Autosegmental Representation of “mùsó ! já:bí” 22
2-10 Partial Three-Dimensional Segment Structure 23
2-11 Autosegmental Representation of Some Rules of Bambara 23

3-1 Feature Matrix Representation of “a” 29
3-2 Typical Syllable Structure 31
3-3 Rule Types Allowed by AMAR, with Conventional Equivalents 36
3-4 Inheritance Structure of Segments within AMAR, with Corresponding Predefined Identifiers 39

4-1 Internal representation of *mùsò+dôn* 50
4-2 Internal Representation of “su DeDo” 54
4-3 Internal Representation of Output from “su DeDo” 54
4-4 Internal Representation of “un DeDo” 55
4-5 Internal Representation of Output from “un DeDo” 55
4-6 Tree Structure 59
4-7 Partial Representation of “un Beso” 60
4-8 “un Beso” after Application of Nasal Assimilation 61
4-9 “un Beso” after Application of Continuancy Rule One 62

5-1 Digo End Run Rule 76
5-2 Restatement of Digo End Run 76

List of Tables

3.1 Predefined Identifiers 33
3.2 Usable Effects under AMAR 37
3.3 Special Input/Output Characters 37

4.1 Conjugation of *ktb* in Classical Arabic 64

Chapter 1

Introduction

Although some tools exist to assist phonologists using older representational frameworks such as that presented in Chomsky and Halle's *The Sound Pattern of English* (Chomsky and Halle 1968, hereafter SPE), until recently there have been few options for linguists wishing to employ computational methods for the more recent autosegmental phonology. This thesis attempts to fill the gap by presenting a system—the Automated Model of Autosegmental Rules (AMAR)—embodying autosegmental representation and mechanics, together with an interface designed to shorten learning time for linguists already familiar with the autosegmental notation first proposed by Pulleyblank (1986). The goal for AMAR is to provide an abstraction barrier such that a linguist may describe natural languages in autosegmental notation—thus allowing him or her to model almost any language—and the system will take care of all computational details. As such, AMAR rules are expressed in a notation that is as close as possible to Pulleyblank's while remaining capable of being written within a text-only system.

Autosegmental theory differs from linear theories such as the SPE theory in that phonemes are not assumed to be atomic, but are hypothesized rather to be composed of autonomous *segments* such as tones or features, interconnected by association lines and situated on independent tiers within a chart. For example, in one version of the theory, one might partially represent a tonal phoneme as in figure 1-1, with **V** on the skeletal tier representing a vowel, and **T** on the tonal tier representing a tone connecting to it.

Autosegmental phonology has its roots in attempts to explain the mechanics of tonal languages (Goldsmith 1976a, Goldsmith 1976b, Goldsmith 1976c). Thus, the system was designed particularly to allow the formulation and testing of tonal rules. For example, one might wish to model the tone shortening rule of Mandarin Chinese. Mandarin has four tones: a high, level tone ("ā"); a rising tone ("á"); a low tone that falls slightly, then rises ("ǎ") fairly high; and a falling tone ("à"). Of these tones, all are of the same length except for the low tone, which is long. Thus, when a low tone precedes another low tone, as in the phrase *Wǒ hěn kùn* ("I am very sleepy"), Chinese avoids an awkward concatenation of two long tones by shortening the first one. This process is, however, rather difficult to describe in terms of linear phonology, since what apparently happens is that the middle part of the tone (the part that "falls") drops out, thus leaving a rising tone. In autosegmental theory, the Mandarin low tone is modeled as three atomic tones (3, 5, and 1) from a scale of five tones (from the highest, 1, to the lowest, 5) attached to a single vowel. When two such vowels adjoin, the 5 tone is simply disconnected from the first vowel, leaving a rising tone. In autosegmental notation, this rule would appear as in Figure 1-2. To model this aspect of Mandarin using the system to be

$$\begin{array}{c}\mathbf{V}\\ |\\ \mathbf{T}\end{array}$$

Figure 1-1: Autosegmental Representation of a Tone

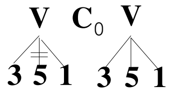


Figure 1-2: Mandarin Tone Shortening Rule

discussed here, one would first provide a file specifying the phonemes and tones, plus the shortening rule, and then specify (either in a file or at run-time) an input to be transformed. Thus, one might provide the input *Wǒ hěn kùn*, as above, and receive the output *Wó hěn kùn* after the system has transformed the input into a tier structure, applied the rule, and transformed it back into a human-readable string.

As stated above, the system was originally designed simply to handle tonal systems such as the one discussed for Chinese. However, it was subsequently determined that by simple additions further aspects of autosegmental theory could be supported, and consequently the system supports feature trees, syllable structures, word and morpheme boundaries, and feature matrices (as in Chomsky and Halle (1968)). Supported rule operations include: deletion of segments; insertion of segments; segment connection; breaking of connections; metathesis, in which a segment is moved somewhere else on its tier (*e.g.*, "atr" might become "art"); spreading, in which a segment connected to another segment on a different tier becomes connected to the latter segment's neighbors[1]; and segment replacement. Both word-internal and sandhi[2] rules are supported, and the system allows (if feature trees or matrices are used) phonemes to be underspecified, with trees or matrices being built up before, during, and after rule application. Some language-specific parameters relating to tones may be specified (Goldsmith 1990, page 18–19), and the system automatically observes the association convention (Goldsmith 1990, pages 11–19), conditions against line crossing (Goldsmith 1990, page 47), and conjunctions against connecting segments which do not freely associate (Goldsmith 1990, pages 45–46) (The user specifies freely associating segments.)

In order to utilize the system, the user must first specify the language to be modeled. One first provides a name for the language, and then a list of phonemes. AMAR is not limited to any given system of orthographic characters[3], so the user could, for example, specify the phonemes of a language using Multi-Lingual Emacs (MULE) or some other such editor which allows non-ASCII characters. Next, the user decides upon the specification method to follow, choosing from CV, CV/Matrix, X/Matrix, CV/Tree, and X/Tree. In this section only the simplest method, CV, will be discussed. This method allows only three tiers—skeletal (the "skeleton," containing C's and V's to which other segments are attached), tonal (containing tones), and phonemic (containing the simplest possible representation of phonemes—their names)—thus basically permitting only rules pertaining to tones, without reference to phonemic features.

In CV, the user would, after having listed the phonemes, specify which are consonants and which are vowels. Tones would next be specified, starting with whether they should automatically be connected to a place on the skeletal tier before rule application. Other tonal specifications include language specific parameters such as the number of tones in the language, the maximum number of tones per vowel and vice versa, the names of the tones, a specification of freely associating segments (e.g., in most tonal languages, tones freely associate with vowels), and representations for characters with tones (for example, in Chinese à might represent the vowel "a" associated with the tones 2 and 4.)

The Mandarin example above would be specified by first describing the language, starting with the name:

```
Language Mandarin:
```

[1] See section 3.3.5 for a more complete description of this process.

[2] rules which concern the interface between words—for example, a rule that moves the final tone of a word into the following word

[3] although anything not composed of a standard alphabetical character followed by either digits or standard alphabetical characters will need to be enclosed in quotation marks

Mandarin has the following phonemes (expressed basically as in Pinyin, the orthographic system used in mainland China):

```
Phonemes:  a, b, c, ch, d, e, f, g, h, i, j, k, l, m, n, o, p, q,
   r, s, sh, t, u, "ü", w, x, y, z, zh.
```

The specification method is CV:

```
SpecMethod:  CV.
```

Of the phonemes, some are consonants, and some are vowels:

```
Vowels:  a, e, i, o, u, "ü".
Consonants:  b, p, m, f, g, k, l, n, r, s, zh, ch, sh, j, q, x, d,
   t, w, y.
```

We wish tones to be initially connected:

```
ConnectTones
```

Mandarin has five tone levels:

```
ToneLevels:  5.
```

In Mandarin there is a maximum of three tones per vowel and no maximum number of vowels per tone.

```
MaxTonesperVowel:  3.
MaxVowelsperTone:  INFINITE.[4]
```

We must specify how to represent tones in the input:

```
ToneReps:
 ā:  a / 1 1,
 á:  a / 3 1,
 ǎ:  a / 3 5 1,
 à:  a / 2 4,
```

and so forth. Finally, we must specify which segments associate with others. In Chinese, as in most (probably all) tone languages, tones associate freely with vowels, and skeletal segments associate with phonemes:

```
Associates:  {segment{T}, segment{V}}, {segment{X}, segment{P}}[5].
```

After the language has been specified, the user must then specify a set of rules, in the order in which they will be applied. For example, the tone shortening rule referred to above would be expressed as follows:

```
Rule "Long Tone Shortening":
NoWordBounds
NoMorphBounds
Tiers:
  skeletal: V       C0 V,
  tonal:    3 (5) 1   3 5 1.
Connections:
  3[1] -- V[1],
  5[1] -- V[1],
```

[4] or leave this field blank
[5] this structure will be explained later

```
    1[1] -- V[1],
    3[2] -- V[2],
    5[2] -- V[2],
    1[2] -- V[2].
  Effects:
    5[1] -> 0.
```

Note that individual segments may be referred to in any of the following methods, as long as no ambiguity is introduced: the name of the segment alone (if there is only one such segment in the chart), the name of the segment followed by a number in brackets indicating which occurrence of the segment is to be selected (counting starting at one, from left top to right bottom), or the name of the segment followed by, in brackets, a number indicating which occurrence it is on a given tier and the name of that tier. For example, in the previous rule, `C0` could have been referred to as `C0` (since it is unique), `C0[1]` (since it is the first instance of `C0` in the chart), or `C0[1, skeletal]` (since it is the first instance of `C0` on the skeletal tier.)

Inputs may be provided directly from the keyboard (standard input), or from a file. Outputs will be sent to standard output, where they may be redirected to a file, if desired. For example, if in the example above the language was specified in the file **chinese**, the inputs were specified in the file **chinese.ipt**, and the user wishes to redirect output to the file **chinese.opt**, then the command

```
amar chinese chinese.ipt > chinese.opt
```

would be used. This would allow, for example, the comparison of program outputs with some file of expected outputs. Since inputs may be taken from standard input and outputs are directed to standard output, the system could be used in a language processing system—for example, input could be redirected to come from a dictionary of some kind, and the system would be set up to produce an output form needed by a parser, to which the output could be redirected.

Chapter 2

Background

There exist two widely-used systems for computational phonology: KIMMO—a morphologically-oriented system based in an SPE-influenced two-level model of phonology, and the Delta Programming Language—a phonetically-oriented system based loosely on metrical and autosegmental phonology; in addition there may be other systems unknown to the author.

2.1 KIMMO

The KIMMO system is based on a two-level model of phonology devised by the Finnish computational linguist Kimmo Koskenniemi and described by him in a series of publications starting in 1983 (Koskenniemi (1983a, 1983b, 1984, 1985), Karlsson and Koskenniemi (1985)). In this model, all phonological rules operate in parallel, with no intermediate representations—that is, all rules directly relate the surface form to the input form. KIMMO represents phonological and morphological rules as finite state transducers; the rules, operating in parallel, move over an input string in one direction only and generate an output form for each character (including "null" characters for which there is an output form and no input form.) Implementations of KIMMO exist in LISP (KIMMO, Lauri Karttunen 1983), Xerox INTERLISP/D (DKIMMO/TWOL, Dalrymple *et al.* 1987)[1], Prolog (Pro-KIMMO, Sean Boisen 1988), and C (PC-KIMMO, Antworth *et al.* 1990.)

Due to its finite transducer representation of rules and its two-level model, KIMMO runs extremely quickly for most applications. Barton, Berwick, and Ristad (1987) shows that KIMMO uses an exponential algorithm, rather than an algorithm linearly proportional to the length of the input, as Koskenniemi had originally claimed, but the speed is nevertheless higher than that of competing phonological systems. Because of its morphological bent, KIMMO handles morphological rules quite well. For example, there might be a rule adding the ending *-oj* to signify the plural in Esperanto. This rule could be specified by:

PLURAL:oj $\Leftrightarrow$ +:0 __

Thus, KIMMO could receive a lexical representation such as *libr+PLURAL* and return a surface form *libroj*, or it could just as easily take *libroj* and return *libr+PLURAL*. In a system like AMAR, one would have to represent "PLURAL" as some sort of phoneme or tone (*cf* the Arabic example in section 4.4), and with one rule system in AMAR, one can only go in one direction, so AMAR would only support something like *libr+PLURAL* to *libroj*.

KIMMO is indeed fast and adequate for many rule systems. However, it has weaknesses in the areas of notation, rule ordering, nonlinear representations, and nonconcatenative morphology (Antworth 1990, Anderson 1988, pages 11–12 and 530–534, respectively). Many implementations require the user, for each rule, to translate by hand from two-level notation to a table of state transitions. In addition, the two-level notation itself is rather foreign to traditional (non-computational)

[1] This implementation accepts two-level rules written in linguistic notation and automatically compiles them into finite state transducers

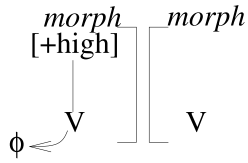


Figure 2-1: Turkish High Vowel Deletion Rule

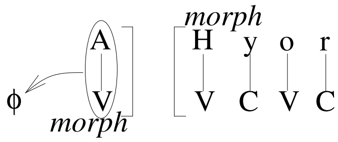


Figure 2-2: Hyor Vowel Deletion Rule

linguists, who tend to be most familiar with replacement rules, *i.e.*, rules of the form "*a* becomes *b* in the environment *c*." Aside from notation, the two-level *model* (which, as it roughly corresponds, under certain conditions, to a Turing machine, should be able to handle any computation) greatly complicates development of complex rule systems. For example, in the Turkish language there is a rule (Oflazer 1993, see figure 2-1) that deletes a high vowel if it is immediately preceded by a morpheme ending in a vowel. However, this rule does not apply if the high vowel is part of the morpheme "Hyor" (where "H" refers to a high vowel unspecified for roundness and backness.) Instead, the preceding vowel is deleted. In KIMMO, this behavior is represented by finite transducers corresponding to the three rules:

H:0 ⇒ V (':') +:0 __

which translates to "H only but not always corresponds to NULL when preceded by a vowel before a morpheme boundary, which corresponds to a surface NULL,"

H:0 / ⇐ V:0 +:0 __ y o r

which translates to "H never corresponds to NULL when preceded by a vowel and morpheme boundary, both to be deleted, and followed by 'yor'" and

A:0 ⇔ __ +:0 H:@ y o r

which translates to "A (a low, non-round vowel unspecified for backness) always and only corresponds to NULL when followed by a morpheme boundary and Hyor, regardless of the surface representation of H." Of these rules, the second acts simply to tell the first rule not to apply whenever the third rule applies. In an ordered rule system, such as that modeled by AMAR, only two rules would be needed, since the application of a rule deleting the vowel preceding "Hyor" would modify the environment before the "H," and preclude it from being deleted. In contrast, AMAR might represent such a rule:

```
Rule "Hyor Deletion":
Tiers:
      root: A  "]m" "m["  H y o r,
  skeletal: V  "]m" "m["  V C V C.
Connections:
  A -- V[1],
  H -- V[2],
  y -- C[1],
  o -- V[3],
  r -- C[2].
Effects:
  A -> 0,
  V[1] -> 0.
```

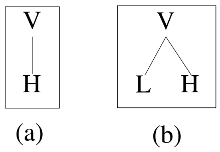


Figure 2-3: Representation of High and Rising Tones in Autosegmental Notation

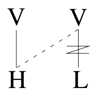


Figure 2-4: Mende High Tone Assimilation Rule

(Figure 2-2 shows the conventional autosegmental representation of this rule.) This rule would then be followed by the rule of high vowel deletion (shown in conventional notation in figure 2-1):

```
Rule "High Vowel Deletion":
Tiers:
      high:      "]m" "m[" +high,
  skeletal:  V   "]m" "m["   V.
Connections:
  V[2] -- +high.
Effects:
  V[2] -> 0.
```

In addition to the complications brought about by the lack of rule ordering, KIMMO's unilinear representation (all phonological elements must be represented as a single linear string of symbols) disallows any sort of rule relying on a multi-tiered representation (*e.g.*, any rule of autosegmental or non-linear phonology.) To handle systems best described by such rules, KIMMO must rely on brute-force methods by simply listing all alternatives. For example, in the West African language Mende (Antworth 1990, Halle and Clements 1983), a lexical low-tone corresponds to a surface high tone after a lexical high or rising tone. In an autosegmental representation, a high tone would be represented as in figure 2-3a, whereas a rising tone would be represented as in figure 2-3b. As seen from the right, therefore, a high tone is exactly the same thing as a rising tone. That is, autosegmental theory would posit that any rule that matches both a high and a rising tone actually is matching a vowel whose rightmost tone is high. Thus, an autosegmental system could represent the rule simply as in figure 2-4, whereas KIMMO has to clumsily mention both high and rising tones:

\`:´ ⇔ [´|ˇ]: V +:0 (C) __

If there were any other type of tone structure in Mende ending with a high tone, for example a falling and rising tone such as the low tone of Mandarin, the autosegmental rule would correctly match that as well, whereas KIMMO would have to mention every possible tone contour ending with a high tone. In addition to the previous correspondence, the data for Mende also show that a lexical rising tone before a surface high tone becomes a surface low tone. This requires two more rules in KIMMO:

ˇ:\` ⇐ __ V +:0 (C) :´
ˇ:\` ⇒ __ V +:0 (C) :´

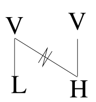


Figure 2-5: Mende Rising Tone Shortening Rule

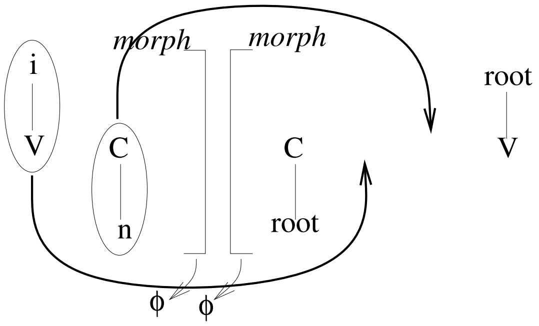


Figure 2-6: Infixation Rule of Tagalog

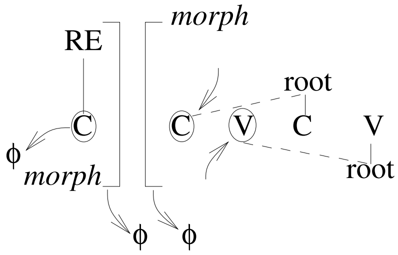


Figure 2-7: Reduplication Rule of Tagalog

In AMAR, this requires a single rule (shown in conventional notation in figure 2-5):

```
Rule "Rising to Low":
Tiers:
  skeletal: V V,
     tonal: L H.
Connections:
  V[1] -- L,
  V[1] -- H,
  V[2] -- H.
Effects:
  V[1] -Z- H.
```

Finally, KIMMO experiences some difficulties with nonconcatenative morphology. That is, KIMMO has no facilities for moving or copying segments, so a language such as Tagalog, where morphemes may require the first syllable of a word to be copied, or where morphemes need to be placed *inside* of a word, requires awkward maneuvers with large finite automata. In other words, KIMMO is not fast for languages such as Tagalog, and its representation simply does not adequately describe the non-

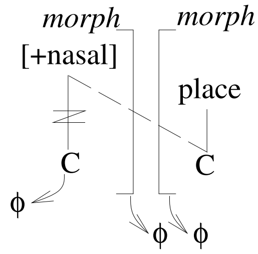


Figure 2-8: Nasal Coälescence Rule of Tagalog

concatenative morphology of that language. Instead, it must rely on clumsy, brute-force methods to try to simulate nonconcatenative morphologies in a concatenative way. For example, three processes found in Tagalog are "in" infixation, CV reduplication, and nasal coälescence (Antworth 1990). If a word in Tagalog begins with a consonant, the infix "in" is placed between the first consonant and vowel—thus the surface form *CinV* would correspond to a lexical form *in+CV*. In KIMMO, this would be represented as:

X:0 ⇒ __ +:0 C 0:i 0:n V

where the infix has to be represented lexically as "X+". In other words, KIMMO pretends that the infix is somehow mysteriously already present in the proper position, and its presence is merely signaled by some artificial infix. In AMAR the rule would be more complicated, but correspond to the more natural "in+", and would move the infix instead of deleting and reïnserting it[2]:

```
Rule Infixation:
Tiers:
     vroot: i      "]m" "m["       vroot,
  skeletal: V  C   "]m" "m["   C     V,
     croot:    n   "]m" "m[" croot.
Connections:
  V[1] -- i,
  C[1] -- n,
  C[2] -- croot,
  V[2] -- vroot.
Effects:
  i -> "m[" _ vroot,
  n -> croot _,
  C[1] -> C[2] _,
  V[1] -> C[2] _,
  "]m"[1,skeletal] -> 0,
  "m["[1,skeletal] -> 0.
```

Next, CV reduplication copies the first CV of a root. In both KIMMO and AMAR CV reduplicative infixes are represented lexically by *RE+*. However, KIMMO treats "RE" as two actual letters, and simply has a rule for every single letter in the language. Thus, the statement "copy the first consonant and vowel of the root" becomes:

> If the first letter of the root is "p," replace "R" with "p." If the first letter is "t," replace "R" with "t," and so forth for every consonant in the language. Next, if the second letter of the root is "a," replace "E" with "a." If the second letter of the root is "e," replace "E" with "e," and so forth for every *vowel* in the language.

The rules would be stated as follows (where "..." should be replaced by a list of all consonants or vowels in Tagalog):

R:{p,t,k,...} ⇒ __ E:V +:0 {p,t,k,...}
E:{a,i,u,...} ⇒ R:C __ +:0 C {a,i,u,...}

The process in AMAR proceeds regardless of whatever consonants and vowels there are in Tagalog. It in fact does not even need to copy the vowels and consonants at all, but simply creates a new C slot and a new V slot in the skeletal tier and connects them to the original representations of the consonants and vowels in a manner consistent with what most linguists believe actually occurs[3]:

```
Rule Reduplication:
```

[2] Illustrated in conventional notation in figure 2-6.
[3] Illustrated in conventional notation in figure 2-7.

```
Tiers:
     vroot:                 vroot,
  skeletal: C    "]m" "m["  C   V,
     croot: RE   "]m" "m[ croot.
Connections:
  C[1] -- RE,
  C[2] -- croot,
  V -- vroot.
Effects:
  croot ::-> C / _ C[2],
  vroot ::-> V / _ C[2],
  C[1] -> 0,
  "]m" -> 0,
  "m[" -> 0.
```

The final Tagalog process under discussion here is nasal coälescence, in which a consonant following "N" (which represents a nasal unspecified for point of articulation) picks up the N's nasality while keeping its own point of articulation. The "N" then disappears (or, it could be said that the "N" coälesces with the following consonant.) For example, *maN+pili* would yield "mamili," *maN+tahi* would yield "manahi," and *maN+kuha* "maŋuha." KIMMO represents this by declaring subsets for labial obstruant stops ("P"), coronal obstruant stops ("T"), velar obstruant stops ("K"), and nasals ("NAS") and declaring deletion correspondences:

	p	b	t	d	k	g	@
	0	0	0	0	0	0	@
1:	1	1	1	1	1	1	1

then using the rules (nasal deletion and nasal assimilation, respectively):

N:0 $\Rightarrow$ ___ (+:0) (R:C E:V +:0) :NAS
{P,T,K}:{m,n,ŋ} $\Leftrightarrow$ N: (+:0) (R:C E:V +:0) ___

AMAR represents this rule as[4]:

```
Rule Coalescence:
Tiers:
     nasal: +nasal  "]m" "m[",
     place:         "]m" "m["  place,
     croot:  croot  "]m" "m["  croot,
  skeletal:   (C)   "]m" "m["    C.
Connections:
  C[1] -- croot[1],
  croot[1] -- +nasal,
  C[2] -- croot[2],
  croot[2] -- place.
Effects:
  "]m"[1,skeletal] -> 0,
  "m["[1,skeletal] -> 0,
  croot[2] :: +nasal,
  croot[1] -Z- +nasal,
  C[1] -> 0,
  croot[1] -> 0.
```

Up till now, AMAR has not shown too great of an advantage over KIMMO in terms of number of rules: simply two fewer. However, in Tagalog more than one of these processes may apply in a single

[4]Illustrated in conventional notation in figure 2-8.

word. When they do, CV reduplication precedes "in" infixation and nasal coälescence (the last two do not apply in the same words.) AMAR takes care of this automatically, simply by ordering the rules. In KIMMO, however, all the rules have to be rewritten such that they take each other into account and the ordering is correct. Thus, for example, the *full* version of the consonant half of CV reduplication becomes:

R:{p,m,t,n,k,ŋ,...} ⇔ __ (0:i 0:n) E:V +:0 {p,p,t,t,k,k,...}:{p,m,t,n,k,ŋ,...}

The actual rule used by KIMMO[5], which the user would have to enter in many implementations, would be the following:

	R	R	R	R	R	R	0	0	E	+	p	p	t	t	k	k	@
	p	m	t	n	k	ŋ	i	n	V	0	p	m	t	n	k	ŋ	@
1:	2	5	8	11	14	17	1	1	1	1	1	1	1	1	1	1	1
2.	0	0	0	0	0	0	2	2	3	0	0	0	0	0	0	0	0
3.	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0
4.	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
5.	0	0	0	0	0	0	5	5	6	0	0	0	0	0	0	0	0
6.	0	0	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0
7.	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
8.	0	0	0	0	0	0	8	8	9	0	0	0	0	0	0	0	0
9.	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0
10.	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
11.	0	0	0	0	0	0	11	11	12	0	0	0	0	0	0	0	0
12.	0	0	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0
13.	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
14.	0	0	0	0	0	0	14	14	15	0	0	0	0	0	0	0	0
15.	0	0	0	0	0	0	0	0	0	16	0	0	0	0	0	0	0
16.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
17.	0	0	0	0	0	0	17	17	18	0	0	0	0	0	0	0	0
18.	0	0	0	0	0	0	0	0	0	19	0	0	0	0	0	0	0
19.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

2.2 The Delta Programming Language

The Delta programming language (Hertz 1990) is a stream-based implementation of nonlinear phonology and phonetics, descended from the linear (SPE-based) rule development system SRS (Hertz 1982.) The Delta system takes a metrical approach to nonlinear phonology, as opposed to AMAR's primarily autosegmental approach. That is, the data used by the system takes the form of "Deltas," or groups of synchronized streams corresponding loosely to AMAR's charts made up of tiers. Thus, the Bambara phrase "mùsó ꜜ já:bí" might be represented as:

```
word:     |        noun         |         verb        |
morph:    |       root          |         root        |
phoneme:  | m | u | s | o |     | j |    a    | b | i |
CV:       | C | V | C | V |     | C | V | V | C | V |
nucleus:  |   |nuc|   |nuc|     |   |  nuc  |   |nuc|
syllable: |  syl  |  syl  |     |  syl      |  syl  |
tone:     |   L   |   H   | L |             H       |
          1   2   3   4   5   6   7   8   9   10  11
```

Nothing is explicitly connected to anything else, but rather simply ordered with respect to one another by means of synchronization lines. In AMAR, as in autosegmental phonology, this might

[5] Condensed with the false assumption that Tagalog only contains the underlying consonants "p," "t," and "k."

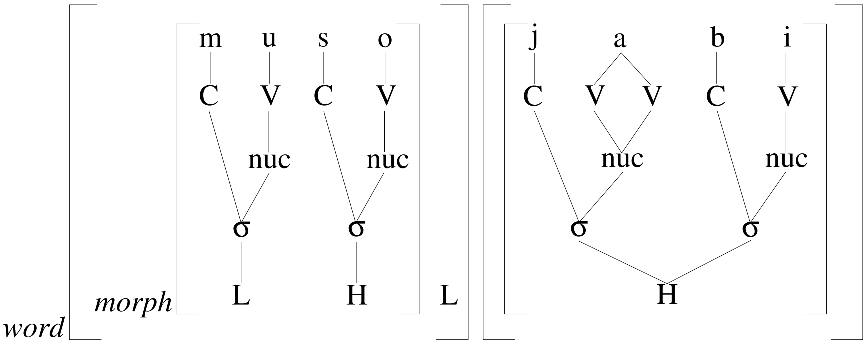


Figure 2-9: Autosegmental Representation of "mùsó ꜝ já:bí"

be represented as in figure 2-9[6]. In order to modify data, the user of Delta writes a program in the Delta programming language (Hertz 1990), which might typically read in some portion of input, transform it into a Delta of appropriate values and then modify synchronization lines and stream tokens (the equivalent of segments in AMAR) in order to simulate the application of rules.

The Delta programming language has great advantages in power and flexibility. The user may program almost any desired behavior (within the limits of synchronized streams), using the general purpose computer language C (Kernighan and Ritchie 1988) whenever the Delta programming language lacks some feature. Moreover, any stream (or "tier," in autosegmental terms) may contain numbers or tokens with a name and any number of *features* (in Delta, objects that may be either binary- or multi-valued, thus corresponding loosely to AMAR features and class nodes). This property allows the user to specify phonetic information such as duration and frequency in addition to the phonological information allowed by the current version of AMAR.

Along with Delta's advantages, and in some cases because of them, there are a few disadvantages that will be felt more or less depending upon the intended application. Because Delta does not assume as much as AMAR about the theoretical underpinnings of a user's work, the user has more flexibility. However, it may well be said that much of the reason for developing autosegmental phonology was to *reduce* the flexibility allowed by previous theories. That is, too much flexibility allows the user to easily make mistakes and describe languages incorrectly or even impossibly—the attempt of autosegmental theory is to describe possible human languages and *only* possible human languages. In addition, Delta possesses this flexibility because it is a programming language. Thus, the user must learn a computer language and a way of thinking significantly different from that of the typical linguist. The Delta user must write all the code for matching and application of rules, dealing with line crossing, application of the association convention, enforcement of conformance with language parameters, the behavior of freely associating segments, etc., whereas AMAR handles all of these automatically. Thus, Delta enforces a procedural view of phonology, where the linguist must specify in detail what is to happen at each step for any given language, as opposed to the more descriptive model of AMAR, in which the user specifies only how a given language is different from other languages, and the program attempts to take care of the mechanics common to all languages[7]. Aside from the Delta System's procedural nature, the major difference between it and AMAR is that it is limited to a somewhat two-dimensional structure. Segments may "connect" or be synchronized with segments above or below them, but a structure such as that in figure 2-10, where each segment associates with three other segments, would be impossible. This limitation is quite significant, as this sort of structure is fairly common in theories of feature geometry, for example. Due to this two-dimensional nature, Delta cannot handle trees such as those postulated by Mohanan (1983) (Mohanon 1983), Clements (1985b) (Clements 1985), and Sagey (1986) (Sagey 1986), thus

[6] In AMAR the user would not directly manipulate a textual representation of a phrase as would a user of Delta.

[7] However, the user of AMAR must still specify the features and tree structure for each language, although these are considered universal. This requirement follows because linguists have not come up with any more or less agreed upon universal structure.

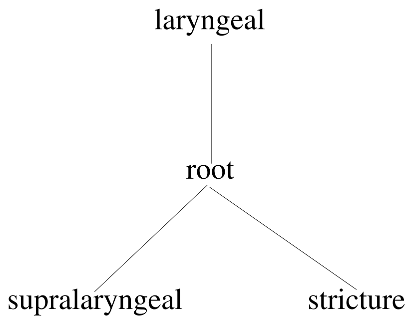


Figure 2-10: Partial Three-Dimensional Segment Structure

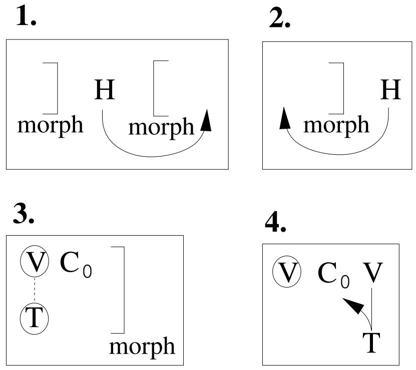


Figure 2-11: Autosegmental Representation of Some Rules of Bambara

excluding much of current autosegmental theory.

The different ways in which linguistic problems are modeled in Delta and AMAR can be shown in the African tonal language Bambara. In Bambara there are two tone levels, and floating tones are used as morphemes, *e.g.* to indicate what in English would be expressed by a definite article. Thus, an indefinite noun is distinguished from a definite noun by the addition of a floating low tone. When a morpheme has additionally a floating *high* tone, there is an interaction between the two tones. Normally, a floating high tone moves forward into the next morpheme. If it is blocked by a low tone, however, it moves into the preceding morpheme. The rightmost tone of each morpheme is then connected to the rightmost vowel, and tones pair up with vowels from right to left, with the leftmost vowel spreading to any extra tones if there are more tones than vowels or the leftmost tone spreading to extra vowels if there are more vowels. Autosegmentally, these processes might be represented by the four rules shown in figure 2-11. In Delta, the effects would be carried out by the following code (in addition to the code required to specify the language and set things up, read in the inputs, etc):

```
:: Forall rule for floating High tone assignment:
:: Forall floating H tones (^bh = "before H", ^ah = "after H")...

forall ([%tone _^bh H !^ah] & [%morph _^bh ^ah]) ->
  do
    if
      :: If the floating H occurs before a floating L,
      :: move the H tone into the end of the preceding
      :: morph.  Otherwise, insert the H tone at the
      :: beginning of the following morph.  Moving the H
      :: tone is accomplished by inserting a new H tone
```

```
      :: and deleting the floating one.
      ([%tone _^ah L !^al] & [%morph _^ah ^al]) ->
              insert [%tone H] ...^bh;
      else -> insert [%tone H] ^ah...;
    fi;

    :: Delete original floating H & following sync mark:

    delete %tone ^bh...^ah;
    delete %tone ^ah;
  od;

:: Forall rule for sync mark merging:
:: For each morph (^bm = "begin morph", ^am = "after
:: morph")...

forall [%morph _^bm <> !^am] ->
  do
    :: Set ^bs (begin syllable) and ^bt (begin tone)
    :: to ^am (after morph):
    ^bs = ^am;
    ^bt = ^am;

    repeat ->
      do
        :: Set ^bt before the next tone token to the
        :: left.  If there are no more tone tokens in
        :: the morph (i.e., ^bt has reached ^bm), exit
        :: the loop.
        [%tone !^bt <> _^bt];
        (^bt == ^bm) -> exit;

        :: Set ^bs before the next syllable token to
        :: the left.  If there are no more syllable
        :: tokens in the morph, exit the loop.

        [%syllable !^bs <> _^bs];
        (^bs == ^bm) -> exit;

        :: Merge the sync mark before the tone and the
        :: sync mark before the syllable:

        merge ^bt ^bs;
      od;
  od;
```

In AMAR, the user would define the following rules:

```
Rule "Floating High Tone Metathesis":
Tiers:
  tonal: "]m" H "m[".
Effects:
  H -> "m[" _.

Rule "Floating High Tone Metathesis (part 2)":
```

```
Tiers:
  tonal: "]m" H.
Effects:
  H -> _ "]m".

Rule "Initially Connect Tones":
Tiers:
  skeletal: (V) C0 "]m",
     tonal: (T)    "]m".
Effects:
  V :: T.

Rule "Spread Left":
Tiers:
  skeletal: (V) C0 V,
     tonal:        T.
Connections:
  V[2] -- T.
Effects:
  << T skeletal.
```

The rest is taken care of automatically. Note that the Delta representation looks much like any programming language, and would be almost completely unreadable to a linguist untrained in computer languages, whereas the AMAR representation corresponds exactly to the autosegmental rules depicted in figure 2-11.

Chapter 3

Design and Overview

The AMAR system consists of two major parts: (1) an interface, where the user specifies languages, rules, and inputs; and (2) internals that store the specifications, apply rules to the inputs, and so forth. Thus, the system may be viewed from three points of view: the viewpoint of general linguistics, that of the user, and the point of view of the program itself.

3.1 Linguistic Overview

The design of AMAR involved certain decisions with regard to what sort of linguistic theories to model, exact formalisms, and how to deal with situations in which the literature known to the designer does not specify details about the mechanisms of autosegmental phonology. These decisions include the following: the exact behavior of the association convention, spreading, inclusion versus exclusion of pointers, and behavior with regard to redundant features.

With regard to the association convention, Goldsmith (1990, page 14) states:

> When unassociated vowels and tones appear on the same side of an association line, they will be automatically associated in a one-to-one fashion, radiating outward from the association line.

This statement does not, however, define the extent of this radiation, the behavior at boundaries, or the exact time of application of the convention. The current AMAR system applies the association convention about each association line produced as a result of the application of a rule. The process occurs after each application of every rule, if the application generated one or more association lines. In AMAR, the association convention is bounded by all directly supported boundary types (morpheme, word, and phrase.) When the pairing process reaches a boundary in one tier but not in another, an automatic spreading process will occur, linking all unconnected freely associating segments to the final segment anent to the boundary reached. This spreading process has been assumed by Goldsmith, but other phonologists have discovered systems in which the process does not occur (Pulleyblank 1986). Therefore the next version of AMAR will most likely not perform this spreading automatically. The final interpretation of Goldsmith's statement made by AMAR is that, in addition to vowels and tones, the association convention is assumed to apply to all freely associating segments.

In some theories (such as that described in Kenstowicz (1994, page 335)), spreading extends until it violates the line crossing condition. Thus, a spreading high tone would be predicted to create a contour tone (here, a falling tone) with the next low tone, since there would be no prohibition against it spreading onto the vowel connected to the low tone. In the current version of AMAR, however, spreading was implemented such that it only extends to unconnected segments. The next version of AMAR may redefined spreading to correspond to the former, less restrictive theory, however.

When AMAR was designed, the author was under the impression that articulatory pointers, as introduced by Sagey (1986), were not commonly used. Therefore, no facility currently exists to support this feature. However, it appears that such pointers are coming into increasing use, and the next version of AMAR will include some such facility.

Occasionally a system will occur in which a class node will be linked to a feature such that the class node has two of the same feature. In this case there are three things that could happen (all three are relied upon by different linguists espousing various theories about autosegmental phonology): the old feature could be deleted, both features could be kept, or the old feature could be deleted only if the two features have the same value. AMAR takes the first course, although if one uses the "-ks" option the second will be chosen.

3.2 User Interface

From the viewpoint of the user, AMAR acts much like a programming language. One first uses a semi-programmatic format[1] to specify a language with a system of rules, and one may then filter inputs through these rules. Within this specification file, the user provides the name of the language, a specification of the language's phonemes, a description of the tonal system, a list of free associates, a set of definitions (essentially macros used to simplify rule definitions), and the rules of the language. The format is as follows:

The first line of an AMAR specification file must be

`Language` **identifier**`:`

where **identifier** may be an alphabetical character (upper or lower case "a" through "z" with no diacritics) followed by a sequence of such characters, possibly interspersed with digits. If enclosed with quotation marks, however, an identifier may contain any characters other than tabs, spaces, periods, new lines, sharp signs ("#"), pluses, or quotation marks. For example, in the Mandarin example this line was "Language Mandarin:".

3.2.1 Phoneme Specification

Following the language name specification is the phoneme specification. This begins with a specification of the names of all the phonemes, in the form:

`Phonemes:` **identifier**`,` ..., **identifier**`.`

that is, the word "phonemes" followed by a colon, a list of identifiers (as above), separated by commas, and a period. Note that statements in the AMAR language follow the general form of a keyword followed by a colon, then a specification of some sort—possibly an identifier, a comma-separated list, or some more complicated specification—and finally ending with a period. Capitalization does not matter in the case of keywords such as "language," "phoneme," etc. Note also that, throughout the language specification file, no two identifiers may refer to the same thing, so, for example, the language name may not be the same as one of the phonemes of the language. After the specification of phoneme names, the user must decide among the possible methods of phoneme representation: CV (based on the notation generally used to describe simple tone languages), CV/Matrix or X/Matrix (similar to the notation used in SPE), and CV/Tree or X/Tree (based on the notation of feature geometry developed in Mohanan (1983), Clements (1985), and Sagey (1986).) The method chosen would be specified by:

`SpecMethod:` **method**`.`

where **method** can be any one of `CV`, `CV/Matrix`, `X/Matrix`, etc.

CV is the simplest method, allowing only three tiers (tonal, skeletal, and phonemic). In this method, the skeletal tier is occupied by C's (consonants) and V's (vowels), as well as the various boundaries (word and morpheme). The phonemic tier simply stores the representation of the various phonemes (*e.g.*, the representation for "a" would be "a," as opposed to some indication that "a" is a

[1] See Appendix for full grammar.

$$\begin{bmatrix} -\text{consonantal} \\ +\text{continuant} \\ -\text{high} \\ +\text{low} \\ +\text{back} \\ -\text{round} \\ +\text{sonorant} \\ \vdots \end{bmatrix}$$

Figure 3-1: Feature Matrix Representation of "a"

low back unrounded vowel, etc.) This representational system is sufficient for tonal languages whose properties are unaffected by features other than syllabicity. If CV is chosen, the next line must be:

`Vowels:` **identifier, ..., identifier.**

and then:

`Consonants:` **identifier, . . . , identifier.**

where each identifier must be identical to one of the phonemes specified earlier. Note that either line may be left out or simply consist of the keyword followed by a colon and a period, in which case there are assumed to be none of that list. The list of vowels must precede the list of consonants for this reason, as, if the consonant list is first, the program will assume that the user is attempting to say that there are no vowels. Any phoneme not defined to be a consonant or vowel is assumed to be an X.

CV/Matrix differs from CV only in that its phonemic tier contains, instead of simple featureless representations, matrices of features corresponding to the notations of the SPE theory. Thus, "a" might be represented as in figure 3-1. This system is primarily useful for writing rule systems in which autosegmental notation is only relevant for the tonal and skeletal tiers. If CV/Matrix is chosen, the user must specify the consonants and vowels as for CV above and then the features, in three sections: "Features," "Defaults," and "FullSpecs." One specifies features first by listing all the features used in the language. This specification is accomplished in the "Features" section, with the same syntax as for the "Phonemes" section (following the keyword "Features.") The default section builds sets of features that correspond to all the specified phonemes—when phonemes are encountered in the input (that to which the rules are to apply), they will be replaced with a matrix of features as specified in this section. The matrices specified in the default section can be ambiguous. That is, the program will work fine if two phonemes end up having the same default representation, but something must happen to disambiguate them. The rules can add additional features to various matrices in the chart, and after rules have applied, the program checks matrices in the chart (produced by applying the rules to the input) against those built up for each phoneme in the default and full specification sections to determine which phoneme to output. Ideally, the matrices built up from both the default and full specification sections should not be ambiguous. Default matrices are specified as follows:

`Defaults:` **matrixspec, . . . , matrixspec.**

where **matrixspec** refers to one of the following:

identifier `->` **matrix**
identifier `->` **identifier**
identifier `->` **identifier matrix**
`any ->` **matrix**
`any ->` **identifier**

matrix `->` **matrix**
`vowel ->` **matrix**
`consonant ->` **matrix**

These specifications are applied in order from top to bottom – the program matches phonemes against the left hand side of the specifications (**identifier** matches a particular phoneme, `any` matches every phoneme, **matrix** matches all phonemes with the features listed in the matrix[2], and `consonant` and `vowel` match all consonants or vowels, respectively) and then, if the right hand side is a matrix, copies the features from the right hand side into the phonemes matched. If the right hand side is an identifier, the program will copy the features from the phoneme the identifier specifies into the phoneme matched by the left hand side, and if the right hand side contains both an identifier and a matrix, features will be copied from first the identifier and then the matrix. If a phoneme already has some feature being copied into it, the original feature will be replaced. One specifies a **matrix** as follows:

`[` **feature**, ..., **feature** `]`

where **feature** is a phonological feature (such as ±voice), specified by "+", "-", or nothing followed by an identifier naming the feature, which must already have been specified in the "Features" section. For example, a voiceless obstruant (such as "p," "t," "ch," etc.) might be specified by means of the matrix `[-voice, -sonorant]`.

The "FullSpecs" section is specified in exactly the same manner as the "Defaults" section, except that it is preceded by the keyword `FullSpecs`.

X/Matrix is exactly the same as CV/Matrix except that syllabicity must be specified as a feature in the phonemic tier rather than as an element of the skeletal tier. Instead of C's and V's, the skeletal tier contains X's. This system is useful for writing rule systems similar to those for which CV/Matrix is appropriate, but in which the rule author wishes to treat syllabicity as a feature. To specify phonemes under X/Matrix, one proceeds as described above for CV/Matrix, except that one may not specify lists of consonants and vowels, and the "Defaults" and "FullSpecs" sections may not use the keywords `consonant` and `vowel` to match phonemes.

CV/Tree allows the user to employ any number of tiers, and represents phonemes as trees of features linked by class nodes, as first proposed in Mohanan (1983), Clements (1985), and Sagey (1986). The CV/Tree specification method allows the full range of features available within AMAR.

After one has chosen SpecMethod: CV/Tree, one must specify the consonants and vowels as described above, and for each phoneme the phonemic tree. This involves specifying the general tree and then defaults and full specifications, similarly to the matrix specifications. When specifying the general tree, the skeletal tier and the tonal tier will already have been specified by default. Unless otherwise specified, the skeletal tier acts as the topmost node of the tree (the root), and the tonal tier is the immediate inferior of the skeletal tier. While specifying the tree, either of these tiers may be freely referred to and placed anywhere within the tree structure. The following syntax must be used in order to specify the general phonemic tree:

`Tree` { **node**, **node**, ..., **node** }

where **node** may be any one of:

{ **identifier** }
{ **identifier** : **identifier** }

or

{ **identifier** : **identifier** : [**identifier**], ... , [**identifier**] }

Of these, the first creates a topmost class node[3]. As mentioned previously, this is by default the skeletal tier, but this method may be used, for example, to create a syllable structure dominating

[2] more on this later

[3] A class node names both a tier and a segment. That is, there will be a tier with the name of the class node, and this tier will contain only segments of that class node type.

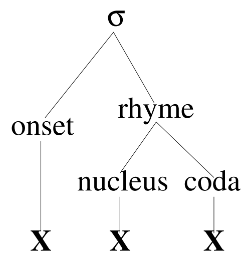


Figure 3-2: Typical Syllable Structure

the skeletal tier, or perhaps even morphemic and word-level structures. The second type of node specification states that the class node referred to by the first identifier is an immediate inferior of (*i.e.*, is directly dominated by) the class node referred to by the second identifier. It is an error if the second identifier has not already been defined to refer to some class node, but if the first identifier has not been previously used, the program will create an empty class node and place it as an inferior of the class node referred to by the second identifier. Note that this process of defining a class node as inferior to another also defines elements of the tier defined along with the class node as freely associating with the elements of the tier defined along with the other node. Finally, the third type of node specification builds a class node referred to by the first identifier (which must be unique and previously unused), defined to be inferior to (and freely associating with) the class node referred to by the second identifier. The class node thus built is also defined to dominate (and freely associate with) the features specified as a list following the second identifier. Each feature is there specified as a previously unused identifier surrounded by square brackets.

As an example, one would define a syllable structure like the one illustrated in figure 3-2 as follows:

```
Tree {
      {sigma},
      {onset:  sigma},
      {rhyme:  sigma},
      {nucleus:  rhyme},
      {coda:  rhyme},
      {skeletal:  onset},
      {skeletal:  nucleus},
      {skeletal:  coda}
     }
```

After the general tree specification, the user will specify defaults and full specifications in a fairly similar manner to that used for the matrix methods above. The defaults section consists of:

`Defaults:` **default**, ... , **default**.

where **default** may be any one of:

identifier -> **segmentspec**
identifier -> **identifier**
identifier -> **identifier matrix**
`any` -> **segmentspec**
`any` -> **identifier**
`featureless` **identifier** -> **segmentspec**
`vowel` -> **segmentspec**
`consonant` -> **segmentspec**
matrix -> **segmentspec**
identifier -> **matrix**

```
any -> matrix
vowel -> matrix
consonant -> matrix
```

or

```
matrix -> matrix
```

where **matrix** is as defined above, and **segmentspec** may be any one of:

```
segment { segspec }
segment { segspec identifier }
segment { segspec : segmentspec, ... , segmentspec }
```

or

```
segment { segspec identifier : segmentspec, ... , segmentspec }
```

where **segspec** should be an identifier (referring to a phoneme or class node)[4], which may either have been defined by the user or be one of a few predefined identifiers referring to general classes of segments and predefined segments, as illustrated in table 3.1. Thus, a **segmentspec** is a structure built around a **segspec**. This structure may simply be the segment referred to by a segspec, or it may have a specified tier (if the segspec is followed by an identifier naming the tier) and inferiors, if the segspec is followed by a colon and a comma-separated list of segmentspecs.

The defaults section matches individual phonemes (by **identifier**), any phoneme (by `any`), consonants or vowels, phonemes with the features specified in a matrix, or phonemes with some particular empty class node (by `featureless` **identifier**, where **identifier** refers to a particular class node). After a phoneme has been matched, four changes could be made to it, depending on the right hand side. If the right hand side contains an identifier, the phoneme's definition is replaced by that of the identifier. If there is a segmentspec, the segmentspec is added to the phoneme definition. If there is a matrix, the features in the matrix are copied (similarly to matrix copying for the Matrix methods) into the phoneme definition. Finally, if the right hand side contains both an identifier and a matrix, the phoneme's definition is replaced by that of the identifier, and the features in the matrix are subsequently copied in. The full specification section is like the defaults section except that it is preceded by the keyword "FullSpecs."

X/Tree is identical to CV/Tree except that syllabicity must be specified as a feature instead of by listing consonants and vowels. By extension, the defaults and full specification sections may not match using the keywords "consonant" or "vowel."

3.2.2 Tone Specification

After phonemes have been specified, the user must next specify the tones of a language, or note that it has none. The first piece of information the user may supply is whether tones specified in the input should begin connected or unconnected to the rest of the input structure[5]. Tones will begin connected if the user includes

```
ConnectTones
```

at the beginning of the tone specification section.

Next, the user must specify the number of tones in the language:

```
NumberofTones: number.
```

[4] A **segspec** may also be a matrix (referring to one or more features), a number (referring to a tone, as will be explained later), or a segspec in parentheses, which is used when segspecs are used in rule definitions, to state that exact matching must be used with the segment.

[5] According to most autosegmental theories today, tones begin unconnected to anything else, and are connected by means of rules. The user may not wish to deal with this, however.

Predefined Identifier	Meaning
P	Phoneme - refers to any phoneme, class node, feature, or feature matrix.
T	Tone - refers to any tone regardless of pitch.
V	Vowel - refers not to a vowel phoneme but rather to a V segment on the skeletal tier (used to identify vowels in CV modes)
C	Consonant - like V above
X	X - refers to an X on the skeletal tier, or to a C or V.
"m["	Morpheme begin
"]m"	Morpheme end
"w["	Word begin
"]w"	Word end
V0	Zero or more vowels
C0	Zero or more consonants
X0	Zero or more X's, vowels, or consonants

Table 3.1: Predefined Identifiers

If the language has no tones, this is the only essential part of the tone specification. After the number of tones, the user specifies the maximum number of tones that can be connected to a vowel at any one time (*e.g.*, in Chinese no more than three tones at a time can be connected to a vowel,) by following `MaxTonesPerVowel:` with a number or the keyword `infinite`. The user may then specify the maximum number of vowels that can be connected to a tone at any one time, in a manner analogous to the previous (with `MaxVowelsPerTone`). If this is greater than one, then a single tone may spread over many vowels, as commonly occurs.

By default, AMAR refers to tones by their level. For example, if there are five tones, the individual tones will be referred to by the numbers one through five. If the user wishes otherwise, for example, if the language has a two tone system and the names "L" and "H" would be convenient, this may be specified as follows:

`ToneNames:` **identifier, identifier, ..., identifier.**

where the identifiers must be previously unused, in order from level one to the highest level, and of the same number as the number of tones.

In order to facilitate input and output, the user may define representations for various phonemes connected to tones. If representations have not been defined, only floating tones (tones not initially connected to anything) may appear in the input, and the program will not know how to print tones in the output, so any phonemes connected to tones will not be printed. These representations are placed in a section preceded by the keyword `ToneReps:`, are separated by commas and terminated by a period. There are two types of tone representations. The first type consists of a phoneme connected to one or more tones, which is represented:

identifier : identifier / segspec segspec ... segspec

where the first identifier must not have been previously used (this will be the representation), the second identifier must refer to a phoneme, and the segspecs must refer to tones. For example, the Mandarin low tone over "a" might be represented as `"ǎ" : a / 3 5 1`. The second type of tone representation is that of a floating tone. This type is completely unnecessary from a practical standpoint, since AMAR treats instances of the tone names in the input as floating tones, but many orthographies treat floating tones differently from connected. At any rate, the user supplies a representation for a floating tone as follows:

identifier : / segspec

where **identifier** is the new representation and **segspec** refers to the tone.

3.2.3 Free Associates

The next possible part of a language specification file consists of a list of freely associating segments (Goldsmith 1990, page 45). In general, only freely associating segments may connect in a given language. Moreover, freely associating segments in some way have a propensity toward connecting, and so, for example, whenever two segments connect, pairs of unconnected but freely associating segments connect to one another in a pattern radiating outward from the initial connection and halting at "blocking" connections (connections whose existence would cause line crossing to occur if a pair of segments were to connect). Freely associating segments may be specified thus:

`Associates:` **assoc, assoc, ..., assoc.**

where **assoc** is a set of two segment specifications:

{segmentspec,segmentspec}

as defined above in the section on defaults. When two segments match the segmentspecs, they are free associates. As previously noted, when defining a tree structure, as in `CV/Tree` or `X/Tree`, AMAR automatically defines immediate superiors and inferiors in the tree to be freely associating. Sometimes the user does not wish this to occur in some particular case. Therefore, one may place a list of non-associating segments before the list of associating segments:

`NonAssociates:` **assoc, assoc, ..., assoc.**

For example, if the user had defined a class node "croot" that only associates with skeletal C nodes and a class node "vroot" that only associates with skeletal V nodes, these class nodes would be initially defined to associate with any skeletal nodes (*i.e.,* X nodes), and the user would have to provide the lists:

```
NonAssociates: {segment{X},segment{croot}}, {segment{X},segment{vroot}}.
Associates: {segment{C},segment{croot}}, {segment{V},segment{vroot}}.
```

3.2.4 Definitions

The user may find that while defining a rule long segment specifications quickly become unwieldy. Thus, AMAR provides a definitions section (preceded by `Definitions:`), wherein one may define a short identifier to refer to a **segspec** or **segmentspec**. Definitions are separated by commas and concluded with a period, and take the following form:

`Define` **identifier segspec**

or

`Define` **identifier segmentspec**

3.2.5 Rule Specification

A phonological rule consists primarily of a situation to match against in the input (for example, an unconnected vowel and tone at the beginning of a word) and an action to take when this match takes place (for example, connecting the vowel to the tone.) In general, the system will match and apply the first rule specified wherever possible, then the second, and so forth.

The rules section consists of the keyword `Rules:` followed by a comma-separated, period-terminated list of rules. A rule is specified as follows:

`Rule` **identifier**:
[`RtoL`]
[`NoWordBounds`]
[`NoMorphBounds`]
`Tiers:` **tier, tier, ..., tier.**
`Connections:` **connection, connections, ..., connection.**
`Effects:` **effect, effect, ..., effect.**

The identifier names the rule, and must be unique. If the keyword `RtoL` is included, the rule will apply right to left. Furthermore, including the keywords `NoWordBounds` or `NoMorphBounds` indicates that the rule will ignore word or morpheme boundaries in matching and application. For example, the tone shortening rule of Mandarin specified `NoWordBounds` and `NoMorphBounds` because two low tones do not have to be in the same word or morpheme for the rule to apply.

The tiers and connections sections of a rule define the situation against which the rule matches. A **tier** is specified as follows:

identifier : segspec segspec ... segspec

where **identifier** names the tier and the **segspecs** can be identifiers naming segments or definitions, matrices, features (identifiers optionally preceded by "+," "-," or "@"[6]), numbers referring to tones, **segspecs** in parentheses[7], or sets of segspecs (specified by { **segspec, segspec, ..., segspec**}.) Note that one may use as segspecs the predefined identifiers listed in table 3.1. For example, one could use "P" to match any phoneme, class node, feature or feature matrix, or "T" to match any tone. Thus, the skeletal tier containing a vowel, zero or more consonants, and the word boundary would be represented:

```
skeletal:  V C0 "]w"
```

In the connections section, the user specifies which of the segments must be connected to which other segments. A connection consists of two **segref**s separated by the symbol `--`, where a **segref** is one of the symbols used to specify a segment in the tiers above. If there is more than one such symbol in the tiers, the symbol must be followed by a number in square brackets indicating which one it is (starting from the top left of the chart and counting down to the bottom right), or a number indicating which one it is on a given tier, followed by the name of that tier, all in square brackets.

The effects section describes what changes are to be made in the input when the rule has been matched. The various different effects are explained in table 3.2 and illustrated in figure 3-3.

3.2.6 Specifying Inputs

AMAR attempts to follow a model of input and output in which a phrase fits onto one text line and appears as similar as possible to standard text. Thus, phrase boundaries are marked by periods or new lines, word boundaries can be marked by spaces, etc. Most characters that can appear in the input or output are defined by the user in the `Phonemes`, `ToneNames`, and `ToneRepresentations` sections. Regardless of whether quotation marks are used in the specification file, they should not be used in the input. For example, a phoneme defined in the specification file as `"ñ"` should appear in the input as `ñ`. All strings other than those specified by the user that would be recognized in inputs and outputs are summarized in table 3.3. Any unrecognized string appearing in the input will be ignored. Thus, one could, for example, use tabs to break up lines and make input files more readable.

Input may be entered from standard input (directly at the keyboard or through a pipe), or it may be read in from a file. For the Chinese example, the file might be the following:

```
wǒ hěn kùn.
% I very tired
% 'I am very tired'
```

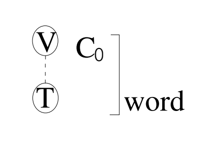
saved under the name "chinese.ipt." In this case, assuming that the language specification file was named "chinese," the user would type

```
amar chinese chinese.ipt
```

[6] @ represents α, which matches either plus or minus.

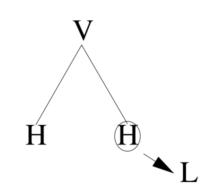
[7] The parentheses indicate that any segment in the input which is to match the segspec must have no more connections to segments on tiers listed in the rule than the segspec has in the rule (in any case, segments must have at least as many connections as specified in the rule, so segspecs in parentheses mean that the matching segment must have exactly the same connections as in the rule)

```
Rule InitiallyConnectTones:
Tiers:
  skeletal: (V) C0 "]w",
     tonal: (T)    "]w".
Effects:
  V :: T.
```



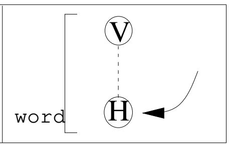
Connect an unconnected vowel separated from the word boundary only by zero or more consonants to the closest unconnected tone to the word boundary.

```
Rule "Mutate Double High":
Tiers:
  skeletal:  V,
     tonal: H H.
Connections:
  V -- H[1],
  V -- H[2].
Effects:
  H[2] -> L.
```



Replace (or mutate) the second of two high tones connected to a vowel with a low tone.

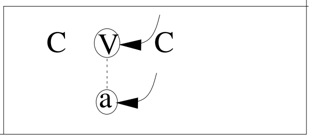
```
Rule "Insert High":
Tiers:
  skeletal: "w[" (V),
     tonal: "w[".
Effects:
  V ::-> H / "w["[2] _ .
```



Insert and connect a high tone to an unconnected vowel immediately at the beginning of a word.

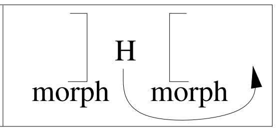
```
Define A segment{V skeletal: segment{a phonemic}}

Rule "Epenthesis":
Tiers:
  skeletal: C C.
Effects:
  0 -> A / C[1] _ C[2].
```



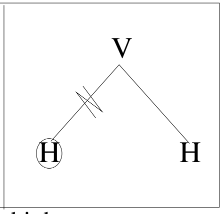
Insert a vowel and an "a" between two consonants, and connect them.

```
Rule "High Tone Metathesis":
Tiers:
  tonal: "]m" H "m[".
Effects:
  H -> "m[" _.
```



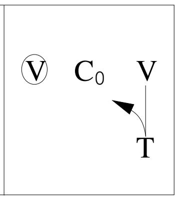
When a high tone is directly between two morpheme boundaries, move the tone into the right–hand morpheme.

```
Rule "Shorten High":
Tiers:
  skeletal:  V,
     tonal: H H.
Connections:
  V -- H[1],
  V -- H[2].
Effects:
  V -Z- H[1].
```



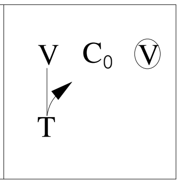
Disconnect the first of two high tones connected to a vowel.

```
Rule "Spread Left":
Tiers:
  skeletal: (V) C0 V,
     tonal:         T.
Connections:
  V[2] -- T.
Effects:
  << T skeletal.
```



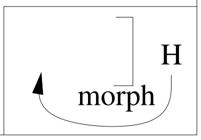
In the environment of an unconnected vowel, followed by zero or more consonants and a vowel connected to a tones, spread that connection left along the skeletal tier.

```
Rule "Spread Right":
Tiers:
  skeletal: V C0 (V),
     tonal: T.
Connections:
  V[1] -- T.
Effects:
  T >> skeletal.
```



In the environment of a tone connected to a vowel that is followed by zero or more consonants, then an unconnected vowel, spread the tonal connection right along the skeletal tier.

```
Rule "High Tone Metathesis (2)":
Tiers:
  tonal: "]m" H.
Effects:
  H -> _ "m[".
```



When a high tone is directly to the right of a morpheme end, move the tone into the left–hand morpheme.

Figure 3-3: Rule Types Allowed by AMAR, with Conventional Equivalents

Effect	Description
segref :: **segref**	Connect two segments
segref -Z- **segref**	Disconnect two segments
segref >> **identifier**	Apply spreading right from segment along the tier specified by the identifier
<< **segref identifier**	Apply spreading left from segment along the tier specified by the identifier
segref -> **segref** _ **segref**	Metathesis: insert **segref** between the two segments
segref -> **segref** _	Metathesis: insert after the segment indicated
segref -> _ **segref**	Metathesis: insert before the segment indicated
segref -> **segref**	replace the first **segref** with the second
segref -> 0	delete the **segref**
segref :: -> **segspec** / _ **segref**	insert **segspec** before the second **segref** and join it to the first
segref :: -> **segspec** / **segref** _	insert **segspec** after the second **segref** and join it to the first
segref :: -> **segspec** / **segref** _ **segref**	insert **segspec** between the second and third **segrefs** and join it to the first
0 -> **segspec** / _ **segref** 0 -> **segspec** / **segref** _ 0 -> **segspec** / **segref** _ **segref**	insert **segspec** before **segref** insert **segspec** after **segref** insert **segspec** between the **segrefs**

Table 3.2: Usable Effects under AMAR

Character	Purpose
`m[`	Beginning of morpheme
`]m`	End of morpheme
`+`	Morpheme boundary (same as `]mm[`)
`w[`	Beginning of word
`]w`	End of word
<space>	Word boundary (same as `]ww[`)
`#`	Word boundary (same as `]ww[`)
`%`	Comment (ignores rest of line)
`.`	End of phrase
<newline>	End of phrase

Table 3.3: Special Input/Output Characters

to see the result. `amar -d` will display internal phoneme and tone representations as rule applications progress. Any error messages produced normally or output produced as a result of "-d" will be sent to standard error output. Normal output (*i.e.*, the results of rule application) will be sent to standard output, where it can be sent to a file, the screen, or wherever the user wishes. To send the output to a file, the user would type (for the Chinese example, assuming output filename "chinese.opt")

```
amar chinese chinese.ipt > chinese.opt
```

By saving the output in a file, the user could compare actual outputs to expected outputs by using a utility such as *diff*. For example, if the expected outputs for Chinese were stored in "chinese.ept," the user could compare by means of the command:

```
diff chinese.opt chinese.ept
```

If there is no output from this command, the expected outputs were the same as the actual outputs.

3.3 Program Design

The Automated Model of Autosegmental Rules was written in C++, an object-oriented programming language, as described in Bjarne Stroustrup's *The C++ Programming Language: Second Edition* (Stroustrup 1991). This language was chosen for its relative speed and portability. The program can be divided into four major modules: the language specification parser, the input/output system, the matching module, and the application module. Central to all of these are the objects themselves.

3.3.1 Objects

The primary objects of the system are *Rules*, *Tiers*, *Charts*, *StrTables* and *Segments*. A Rule consists primarily of a name and two sets of tiers. The first set, called "original" is used to match against the Chart built as a result of reading the input. When the "original" tiers have been matched, the application module modifies the matching section of the Chart such that it matches the other set of tiers held in the rule, the "replacement" tiers. In addition to the name and the tiers, a rule also holds boolean variables specifying whether it is a sandhi rule and whether it ignores word or morpheme boundaries. A Tier is primarily a holder for segments, and as such consists of a name, a list of segments, and a current position within that list. The Chart and StrTable ("String Table") together hold almost all of the data used by AMAR. The chart holds the tiers into which the input is placed after it has been converted into autosegmental form, the name of the language, the rules in the order they are to be applied, a map defining which segments freely associate, and language-specific parameters such as the maximum number of tones per vowel, whether sandhi rules exist, the number of tones, the tree structure, and whether tones should be initially connected. The other half of AMAR's data is held in an StrTable. This structure maps strings to tiers, segments, etc., and contains definitions for anything to which the user refers by means of an identifier in the language specification file.

If the chart and string table are the main containers of data in AMAR, the *Segment* is the main form of that data. A *Segment* could actually be any of a number of object-oriented classes, all of which inherit, directly or indirectly, from the parent class Segment, as shown in figure 3-4. As a consequence of this inheritance scheme, if a rule refers to some segment type from which other segment types inherit, the reference will match anything of that type or of any of the inheriting types. All segments contain a unique identification number, a pointer to the tier they are on, and some indication of type. In addition, connectable segments (all those which inherit from ConnectableSegment) contain pointers to all *superior* and *inferior* segments, indications of whether matching is exact or not, and information relating to spreading (whether the segment spreads right or left, and along which connection). When segments are connected to one another, one segment is always inferior and one superior. This relationship is determined by the user's definition of tree structure for the Tree specification methods, and in the other methods skeletal segments are superior to tones and phonemes. As will become apparent, most routines operating on a given segment can only affect the segment itself and any segments connected inferior to it.

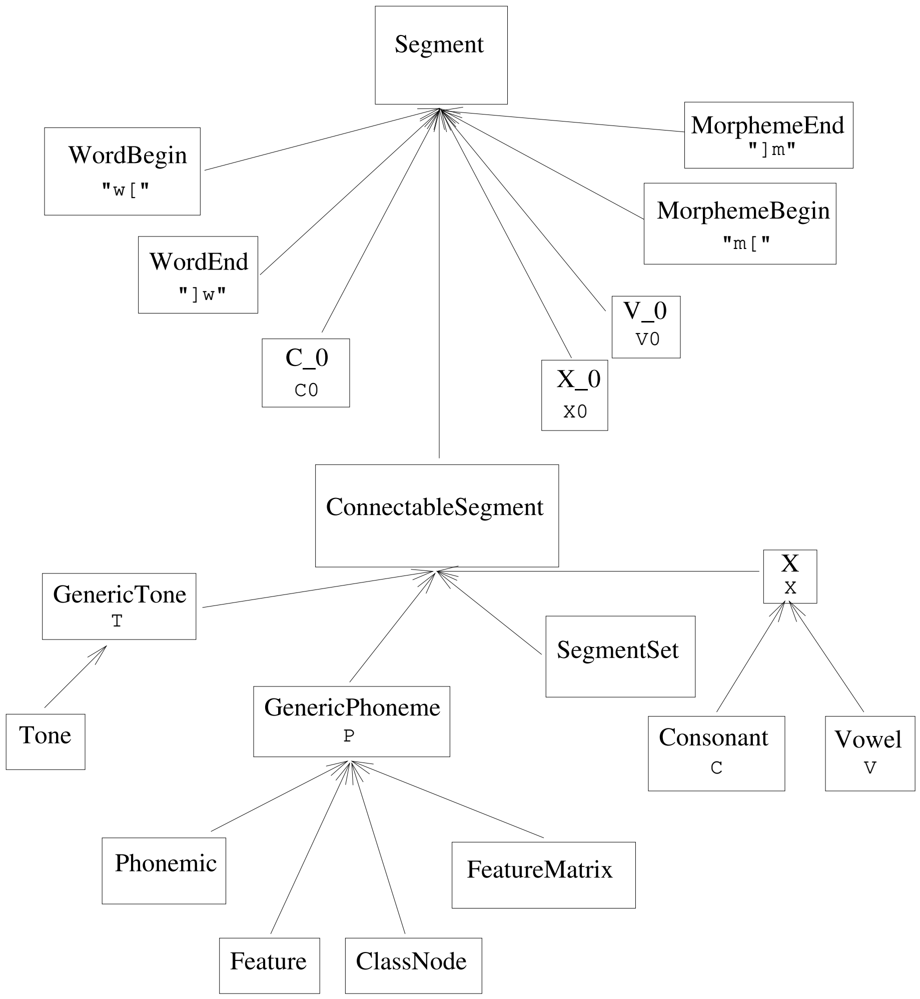


Figure 3-4: Inheritance Structure of Segments within AMAR, with Corresponding Predefined Identifiers

3.3.2 Language Specification Parser

The language specification parser acts mainly to build the objects needed by the other modules and to provide information about how other modules should behave. The parser consists of a C program generated automatically from a grammar file written in *yacc* and a C++ module containing procedures to be called upon receiving various instructions in the language specification file. This program receives input from a lexical analyzer, another C program generated automatically from a specification written in *lex*. The lexical analyzer takes the input from the language specification file and breaks it up into tokens. If the token happens to be an identifier (token "ID"), the procedure `install_id` builds a string table entry which contains the original string entered by the user. String table entries may additionally contain pointers to tiers, segments, full specifications, and "matches," which will be explained later. These entries may be used for four different processes: they may define a "phoneme" (here used to refer to anything that can appear in the input or output other than a boundary or comment); they might define a tier, a class node, or a feature; they could be used for a "match" referencing some segment[8]; or they could simply be used to give a name to something.

For a "phoneme," the entry will contain the corresponding segment, and perhaps a full specification (as generated in the "FullSpecs" section, and used for matching after rules have applied.) The form of a phoneme's segment varies greatly depending on the specification method and the type of phoneme. There are three phoneme types: phonemic segments (defined in the "Phonemes" section), floating tones (defined automatically based on the number of tones and optionally by the "ToneNames" and "ToneRepresentations" sections), and phonemic segments attached to tones (defined in the "ToneRepresentations" section.) A phonemic segment will be an object of type "Phonemic" (a connectable segment which also contains a string representation) in CV mode, a "FeatureMatrix" (a connectable segment which also contains a list of features) in the Matrix modes, or a "ClassNode" (a connectable segment which also contains a name) heading a tree of class nodes and features in the Tree modes. This segment, in whatever mode, will have a "C," "V," or "X" as a superior. A floating tone segment will simply be an unconnected "Tone" object, a connectable segment which also contains an integer level. Finally, a phonemic segment attached to tones will be a "V," "X," or "C," depending on the original phonemic segment. Whichever skeletal segment it is, it will have the tones as inferiors, along with a copy of the original phonemic segment.

When defining a tier, class node, or feature, string table entries will contain both a tier and a segment, because in autosegmental phonology tiers may only contain specific types of segments; any segment to be stored in the tier must match the segment stored along with that tier. Thus, the skeletal tier contains an "X," the tonal tier contains a "T," and the phonemic tier[9] contains a "P." Class nodes and features may only appear on their own tier in the Tree modes, so when they are defined (in the "Tree" section), a tier with the same name is defined and placed in the string table entry along with the segment. In contexts where a tier is expected, the entry will be treated as a tier, and in contexts where a segment is expected, the entry will be treated as such.

When the user defines a rule, the parser creates the "original" and "replacement" tiers, which are initially identical (although the segments in the replacement tiers are copies of the segments in the original tiers and not the actual segments themselves). In addition, it creates a map containing "matches." Each "match" contains a copy of some segment from the rule tiers, along with the position of that segment within the original and replacement tiers. In the "Effects" section, the parser will take the user's **segrefs** and find the "match" segment corresponding to the reference. In the parser, almost all global data must be transferred in string table entries, so this "match" is stored in such an entry. The parser then uses the "matches," the map, and the rule to modify the "replacement" tier (and the map, so that later effects do not get confused) in accordance with the desired effect.

The final, and simplest, process in which the string table is involved is simply that of giving names to objects. Thus, charts, rules, class nodes, phonemes, features, tones, and tone representations have names corresponding to some string table entry. In order to save memory and limit expensive string

[8] This corresponds to a **segref** in the specification file

[9] These three tiers are predefined before parsing begins.

copying operations, the *only* strings stored within AMAR are those found in the string table, and all other objects simply contain references to them.

The only operation in which the parser is involved, other than building objects and storing them in the string table, is building objects and storing them in the chart. Thus, the parser generates the tiers held by the chart (except for the three tiers already stated to have been predefined), the rules to be applied to the chart (also stored in it), the map of free associates (Free associates are defined "manually" in the "Associates" section and automatically when class nodes are defined in the "Tree" section—class nodes automatically freely associate with their direct superiors and inferiors), and all of the parameters: the maximum number of tones per vowel and vowels per tone, the number of tones, whether tones connect, and whether sandhi rules and sandhi rules that apply from right to left exist.

3.3.3 Input and Output

The input/output system of AMAR relies primarily on the string table and the matching system, although it provides a surprising amount of complexity on its own. The overall input/output system begins by going through the string table and finding the maximum length of a phoneme, where a phoneme is specially defined to mean a string that may legitimately appear in the input. It then attempts to read in one word at a time from the input. If there are sandhi rules, AMAR reads in a phrase, then applies the rules in order and outputs the phrase, deleting it from the chart. Otherwise, the system reads in one word at a time, applies rule to it, then prints and deletes it, thus saving memory and minimizing delay between outputs.

The Input System

The input system can be divided into a section that reads one word at a time and one that reads a segment at a time. The former simply places a word begin at the beginning of each tier in the chart, then calls the other section to return one segment at a time until a word end is returned, at which time a word end is placed at the end of each tier. Each tier's current position is set to the beginning of the word read in, and the section returns control to the main loop that called it. As each segment is read in, the "read_word" procedure checks to see whether it is a boundary, a floating tone, a phoneme connected to a tone, or a simple phoneme. Boundaries are copied and added to every tier of the chart. Floating tones are added to the tonal tier. Phonemes, connected or unconnected to tones are sent to the procedure "add_skeletal_segment," which searches through all the connections and adds each segment to the appropriate tier (when a segment is defined in the parser, it is associated with its tier.)

The section that reads in one segment at a time[10] takes one character at a time from the input. These characters are built up into a string whose maximum length equals the maximum length of a phoneme. If the first character read by the function represents a boundary or comment (<space>, <return>, "#", ".", "+", or "%"), the program will go through the input and eliminate redundant boundaries and comment lines, and finally return a morpheme begin or end (for morpheme boundaries) or a word end (for other boundaries), and set the flag "eophrase" ("end of phrase"), if appropriate. Otherwise, "read_segment" will attempt to find the longest string matching a phoneme in the string table. This is accomplished by looking up the string as it is built—if the table contains a phoneme corresponding to the string, this phoneme is stored as the tentative segment to be returned and the length of the string at that point is stored as well. When this string matches the maximum length, the program checks to see if a segment was found. If not, the first character of the string is discarded (in this way, "read_segment" ignores unrecognized characters) and the rest are put back to be read later. The procedure will then return a flag denoting that no segment was found. If a segment was found, any characters which were read after the match are put back into the input[11].

[10] accessed via the function "read_segment"

[11] Actually, for safety the program maintains a stack of characters, into which characters to be reprocessed are placed. When "read_segment" wishes to read a character, it first checks to see if there are any in the stack, and, if so, pops off the top character.

Finally, the segment is returned.

The Output System

AMAR produces two types of output: “process” output used to show what is going on or report errors, and the actual output produced when rules are applied to the input. “Process” output is sent to standard error, and actual output to standard output.

Error reporting is fairly straightforward; when an error occurs, a message is output, and processing halts. The other type of “process” output only appears when AMAR is run with the “-d” (“debug” or “demo”) option. This option displays the contents of the chart before rules have applied, states which rules are applying and whether any change was made, and, if a change was made, displays the chart contents after rule applications. The “-d” output is produced by “print()” members of the ClassNode, Feature, FeatureMatrix, Tone, and Phonemic classes, which simply print the information stored in the class object other than that stored in any connectable segment (thus, names, values (plus, minus, alpha, or undefined), features, levels, and representations, respectively), along with a unique identifier for ClassNodes and Features so that effects such as shared nodes become evident.

To produce actual output, AMAR goes through the skeletal tier of the chart, one segment at a time, and calls the “print()” member for each segment encountered. Since the skeletal tier only contains boundaries and X’s[12], there are only five such members, corresponding to morpheme begins, morpheme ends, word begins, word ends, and X’s. When a morpheme begin immediately follows a morpheme end, the program prints a “+”. Similarly, when a word begin immediately follows a word end, the program prints a space. Otherwise, the boundary print functions do not output anything.

The print member for X’s must match the segment against the full specifications stored in the string table, if any, or the segment specifications found there. To do this, the X is marked for exact matching, and is then matched against every phoneme in the string table, using the matching system to be described later. If there is no match, nothing is printed. If there is only one match, AMAR prints the string stored in the table entry containing the matching phoneme. Otherwise, AMAR will print all of the matching strings in parentheses and separated by slashes.

3.3.4 Matching

Before a rule may be applied, it must be matched. That is, the program must search through the chart and find a position corresponding to the situation described in the rule. In fact, the program must find such a position for each tier mentioned in the rule, and all of these positions must be consistent. That is, if the rule mentions that some segment in one tier is connected to some other segment in another tier, the matching chart segments for the two tiers must connect to one another.

In order to achieve this goal, the matching section, moving from the most superior tier of the rule[13] to the least superior tier, attempts to find a match position for each tier. To ensure consistency and efficiency, for a given tier the program only begins matching with the first rule segment that does not connect to previously matched tiers. When all match positions have been found, the program adjusts them so that each match position actually refers to the first item on that tier that was mentioned in the rule, by checking for each tier to see whether a matched item on a previous tier connects to some segment previous to the current match position for the tier.

The matching process for a tier occurs as follows: the procedure keeps track of a position within the tier, starting from the recorded tier current position (either one position after the last position at which the current rule was applied on the tier or at the beginning of the word,) and starts off by finding the first rule segment both in the proper tier and unconnected to a previously matched tier. This segment is matched against the segment at the current tier position, moving the tier position forward until either the segment matches or there are no more segments in the tier, in which case the rule does not match. If the segment did match, the position is saved as a tentative match

[12] and C’s and V’s, which are here treated the same as X’s

[13] In matching, only the “original” tiers of a rule make a difference. Therefore, any reference to a rule tier in a discussion of matching refers to one of the “original” tiers of that rule.

position, and the procedure attempts to match the rest of the rule. If the rest matches, the saved position is returned. Otherwise, the matching process begins again from the next position after the saved position, until the tier either matches or conclusively does not match (by not having enough segments after the current position to match.)

Matching occurs rather differently for connectable and non-connectable segments. Any given non-connectable segment may either be a boundary or a "zero," which is merely a notation matching zero or more connectable skeletal segments. Boundary matching is fairly simple: any boundary matches another of the same type. A zero always successfully matches; its function is to move the current tier position forward to the first position in which there is not a segment of the type of the zero. For example, after matching against a C_0, the current tier position will be on the next segment which is not a C.

For matching two individual connectable segments, there are three levels of matching: *eq*, *equal*, and *eqv*. That is, one ascertains whether two segments match by calling the function `eq`. This function checks whether the two segments are *equal*, then checks for each of the rule segment's[14] inferiors whether it is *equal* to some other of the chart segment's inferiors. `eq` also makes sure that the rule segment's inferiors are, within their tiers, in the same order as the chart segment's.

The `equal` procedure checks first to see if the rule segment is *eqv* to the chart segment. If this is so, the tiers must have the same name. Finally, if the tiers have the same name, the segments must match *exactly* or *roughly*, depending on whether the rule segment has been marked exact (by surrounding it in parentheses.)[15] To match exactly, the chart segment may have fewer or more connections than the rule segment, but within the tiers accessible from the rule segment (the tiers to which the rule segment is connected), the chart must have the same number of connections. To match roughly, the chart segment must have the same number of inferiors as, or more inferiors than, the rule segment, within the tiers accessible from the rule segment.

For a rule segment to be *eqv* to a chart segment, the chart segment must generally be of the same type as the rule segment, or of a type which inherits from the rule segment type (see figure 3-4.) However, a feature matrix may be *eqv* to a feature, a feature matrix, or a class node. In addition, a segment set may be *eqv* to a segment or a segment set. For a feature matrix to be *eqv* to a feature, every feature in the matrix must either be *eqv* to the feature or *eqv* to some feature connected to the feature. To be *eqv* to another feature matrix, a feature matrix must be such that every feature in it is *eqv* to a feature in the other matrix. Finally, to be *eqv* to a class node, every feature in the matrix must be *eqv* to an inferior of the node. For a segment set to be *eqv* to a segment, some element of the set must be *eqv* to the segment. For the set to be *eqv* to another set, every segment in the set must be *eqv* to some segment in the other set. For a tone to be *eqv* to another tone, the tones' levels must be the same. For phonemes (of the CV mode type), the representations must be the same. For class nodes, the names must be the same. Finally, for features the names must be the same, and the rule feature's value must be "α" (specified by "@", since there is no α key on most keyboards) or the values must be equal (both pluses, minuses, or unspecified.)

3.3.5 Application

After consistent matching positions have been established for every tier, the application section begins by making a correspondence map between the rule and the chart. This map takes essentially the same form as the map used in the parsing section, containing "matches" consisting of a copy of each segment in the "original" tiers paired with the chart location of the matching segment[16]. The section next loops through the rule tiers, in the tier looping through the "replacement" tier and the map corresponding to it. If the current rule element is the same as the current map element (if they have the same identification number), it will check whether the segments have the same number

[14] The equality operations under AMAR are not reflexive, so for any operation of the type $A = B$, the segment A will be referred to as the rule segment, and B as the chart segment, since that is the usual order in which the equality operations are called.

[15] Note that by the Conjunctivity Condition (Goldsmith 1990, page 39), segments to be deleted or replaced should be marked exact. Segments that are explicitly supposed to be unconnected should also be marked exact.

[16] For more insight into the construction of this map, see the function `Rule::application` in appendix C.

of connections, and if they do not or the connection spreads, connections will be adjusted. If the segments are not the same, segments will be adjusted.

Connection Adjustment

The connection adjustment subsection breaks any connections that exist in the map, but not in the replacement chart, and adds those that are in the replacement chart but not the map. Connection adjustment occurs in the map, which is fairly simple, and in the chart, which, as will be seen, is not quite so simple.

The map and the rule charts do not always reflect the actual chart very exactly. If a rule shows two segments connected, they need not be *directly* connected in the chart. Thus, when breaking a connection in the chart, the procedure (`break_connection`) responsible for disassociation first checks to see if the segments are directly connected. If so, they are disconnected. Otherwise, it will check to see which of the superiors of the inferior of the two segments is connected to the superior of the two, and will then disconnect the inferior segment from the connecting superior.

To add a connection, the procedure (`add_connection`) begins with two segments to be connected: a superior and an inferior. Next, the actual two segments to connect must be determined. The general principle used for this is to disturb tree structure as little as possible. Therefore, the connection will be made as close as possible to the inferior segment, and the procedure searches for a new superior segment to connect to the inferior. If the superior segment provided freely associates with the inferior, the new superior will be the same as the old. Otherwise, the procedure looks at all the segments dominated by the superior segment and chooses the segment having the fewest inferiors (to be as far as possible from the top of the tree) amongst those that freely associate with the inferior segment. At this point, the procedure will search through the chart and see whether the new connection will cross any existing connections. If it will, the crossed connections will be broken. Finally, the connection will be made.

If a connection spreads (if, during rule construction, a segment was marked `spread_left` or `spread_right`), the segment marked to spread will connect to every segment that meets the following criteria: it is on the same tier as the connection along which the segment spreads, it is in the direction of spreading, it freely associates with the spreading segment, it is not separated from the spreading segment by any sort of boundary or crossing connection, and connecting to it will not cause a violation of the tones per vowel and vowels per tone parameters.

Segment Adjustment

If the current map segment is not the same as the current replacement chart segment, then the rule involved segment deletion, metathesis, insertion, or replacement. If the current rule segment is found in the map somewhere, and the map segment is not found in the replacement chart, then the chart segment pointed to by the current map segment must be deleted. If the rule segment is in the map *and* the map segment is in the replacement chart, then the chart segment must undergo metathesis. If the rule segment is not in the map, and the current map segment is in the replacement chart, then a new segment must be inserted in the chart. Finally, if the rule segment is not in the map, and the map segment is not in the replacement chart, then the chart segment must be replaced.

To delete a segment from the chart, it is first detached from all its connections, then simply removed from the tier. The same process then occurs to the corresponding map segment.

Metathesis essentially deletes the chart segment, then reïnserts it at the proper position, which it finds by looking for the chart segment corresponding to the map segment that precedes the proper position. This process is then repeated on the map segment that corresponded to the chart segment moved.

Inserting a segment first involves making two surface copies (*i.e.*, copies which do not have connections) of the rule segment to be inserted, and then insert one copy into the chart and one into the map. Next, the insertion process duplicates the rule segment's connections in the map and chart. If the rule segment is connected to something that does not appear in the map, the connection is not duplicated, as it will be taken care of when the other segment is inserted.

The final process, replacement, is generally identical to insertion of the new segment followed by deletion of the old segment except in the case where the segment involved is a feature matrix. For a feature matrix, replacement is just a feature change, so this type of replacement is accomplished by copying the new features into the chart matrix and replacing the map matrix as would happen in a normal replacement.

Association Convention

If any new connections are made during application of a rule, the program will attempt to apply the association convention at every modified point of connection (any connection made as the result of a rule) in the chart.

To apply the association convention at a given connection, the procedure[17] begins just left of the connection and connects pairs of unconnected (in terms of the two tiers concerned) but freely associating segments until it reaches either a boundary or a connection between the two tiers. If a boundary is reached, the procedure will keep attaching segments from the tier in which a boundary was not reached until either there are no more freely associating segments in that tier, a boundary is reached in that tier, a connection between the tiers is reached, or further connections would violate restrictions on numbers of tones per vowel or vice versa. After the association convention has been applied in one direction, the procedure then attempts to apply it in the other.

Back to Output

After the association convention has been applied, the application section returns and either the next rule is applied, or the modified chart is printed and emptied.

[17] `Chart::apply_assoc_convention`—see appendix C

Chapter 4

Examples

To give the reader a feel for the workings of AMAR, this chapter will begin with a very simple example based on an artificial tone language with only three phonemes and proceed through increasingly complicated examples based on Bambara, Spanish, and finally Arabic. A listing of previously incompletely specified examples can be found in Appendix B.

4.1 Simple Example

This section will model the imaginary language "Abc." In Abc, there are three phonemes: "a," "b," and "c." The first of these is a vowel, and the rest are consonants. Abc has two tone levels—low (L) and high (H). It allows any number of tones to be connected to a vowel, but only three vowels may be connected to a single tone. Finally, Abc associates tones to vowels from right to left, and connected tones *spread* to connect to toneless vowels to their left.

Thus, the language would be represented:

```
Language Abc:

Phonemes:  a, b, c.

SpecMethod:  CV.

Vowels:  a.
Consonants:  b, c.

ToneLevels:  2.

MaxTonesperVowel:  INFINITE.
MaxVowelsperTone:  3.

ToneNames:  L, H.

ToneReps:  "à":  a / L, "á":  a / H, "â":  a / H L, "ǎ":  a / L H, "ā":  a
   / H H.

Associates:  {segment{T}, segment{V}}, {segment{X}, segment{P}}.

Rules:

Rule "Initially Connect Tones":
Tiers:
  skeletal: (V) C0 "]w",
     tonal: (T)    "]w".
```

```
Effects:
  V :: T.

Rule "Spread Left":
Tiers:
  skeletal: (V) C0 V,
     tonal:        T.
Connections:
  V[2] -- T.
Effects:
  << T skeletal.
```

The parsing process would proceed as follows: The parser would first create a *Chart* object with the name "Abc." Next, it would create three objects of the class "phoneme," with the representations "a," "b," and "c," respectively. These would be stored in the string table. Then, the parsing mode would be set to "CV," and the phonemes would be connected to a "C" or a "V" depending on whether they were listed as "Consonants" or "Vowels." The system will then note in the chart that there are two tone levels and that there is no limit on the number of tones per vowel, but that one may only associate up to three vowels to a single tone. After this, the system will reäct to the "ToneNames" field by entering "L" and "H" into the string table referring to low and high tone levels. As the last part of the tone definition section, the system will then create string table entries corresponding to the vowel "a" connected to a low tone, a high tone, a falling tone, a rising tone, and a long high tone. In the "Associates" section, the parser enters the pairs "Tone"/"Vowel" and "X"/"Phoneme" into an internal list of freely associating segments, signifying that tones freely associate with vowels and that phonemes freely associate with X's, consonants, and vowels. At the first rule, the parser will create a *Rule* object with the name "Initially Connect Tones." It will then create three tiers called "skeletal": a mapping tier, an "original" tier (for matching), and a "replacement" tier for application. Into these tiers will go a V segment marked for exact matching, a C0 segment, and a word-end segment. Next, three tonal tiers will be created, into which will go a T segment (which matches any tone) marked for exact matching and a word-end segment. Finally, in the "replacement" chart and the map (but not the chart used for matching), the V will be connected to the T. At the second rule, the parser will create the same tiers as before, putting a V marked for exact matching, a C0 segment, and a V into the skeletal tier and a T into the tonal. Next, in the "original" chart, the "replacement" chart, and the map, the second V will be connected to the T. Finally, the T in the "replacement" chart will be annotated to indicate that it spreads left along its connection to the skeletal tier.

If, after parsing, the system were to receive the input "abcaaaaacL", the input and output section would put a word begin on the phonemic, skeletal and tonal tiers. It would then put the segments "abcaaaaac" on the phonemic tier, "VCCVVVVVC" on the skeletal tier, and "L" on the tonal tier. The segments on the phonemic tier would each be connected to a segment on the skeletal tier, but the "L" would be a floating tone, unconnected to any other tier[1]. Each tier would then receive a word-end segment.

The first rule would then look throughout the chart for an unconnected V followed by zero or more consonants and the word end on the skeletal tier and, on the tonal tier, an unconnected T followed immediately by the word end. This matches against the last vowel in the word (surprisingly enough, "a") and the low tone. These two segments are joined, and the association convention attempts to apply. However, there are no adjoining pairs of unconnected tones and vowels, so the association convention does not apply. The second rule then matches against the vowel/tone pair just connected and connects the low tone to the second to last vowel on the skeletal tier. It then continues to spread, attaching the low tone to the third to last vowel. However, any further attachments would cause there to be more than three vowels attached to the tone, so spreading ceases.

[1] This would be the case also if the low tone were introduced as, *e.g.*, à, in the input. If the user wished tones to start out connected, he or she would put the keyword "ConnectTones" before the "ToneLevels" line.

Thus, the input "abcaaaaacL" produces the output "abcaaàààc." This output is produced by going through the skeletal tier and matching each X (consonant or vowel) in it against elements in the string table. When an X matches exactly one of the "phonemes" in the string table, the output section prints that phoneme's string representation. Thus, "a" is the string representation of a skeletal C connected to a phonemic *Phonemic* with the representation "a," etc.

If the input were "báaHcL," the "L" would be connected to the last "a," and the association convention would connect both high tones (represented by "á" and "H") to the first "a." The spreading rule would not apply (since there would be no unconnected V elements), and the output would be "bāàc."

4.2 Bambara

After having described the imaginary language Abc, this section will turn to a small subset of the phonology of Bambara, a Mande tone language spoken in Mali (Hertz 1990). Bambara has a twelve-vowel system, containing the standard five vowels of Spanish, Japanese, and many other languages ("a," "e," "i," "o," and "u") paired with nasalized equivalents. Completing Bambara's vowel inventory are the vowels "I" and "E." Hertz (1990) lists the following consonants: "p," "b," "m," "t," "c," "d," "s," "n," "r," "j," "k," "g," and "ŋ." There are two tone levels, and high, low, rising and falling tones appear on the surface. In Bambara, tones do not begin connected to specific consonants, but are rather grouped with entire morphemes and attached via the two rules discussed earlier for language Abc. Before these rules apply, however, there are two rules that apply to tones found outside of normal morphemes (here termed "floating tones.") In general, floating tones may be found underlyingly in the pattern "H" or "HL." The rules are as follows: when a high tone ("H") immediately precedes a morpheme begin, it is moved into the following morpheme. If, however, this rule does not apply (typically because the high tone was blocked by a floating low tone ("L")) the high tone is moved into the previous morpheme.

The subset of Bambara phonology described above might be specified by the following:

```
Language Bambara:

Phonemes:  m, n, "ŋ", b, d, j, g, p, r, t, c, k, s, i, in, I, e, en, E, a,
   an, u, un, o, on.

SpecMethod:  CV.

Vowels:  i, in, I, e, en, E, a, an, u, un, o, on.
Consonants:  m, n, "ŋ", b, d, j, g, p, r, t, c, k, s.

ToneLevels:  2.

ToneNames:  L, H.

ToneReps:  "á" :  a / H, "à" :  a / L, "â" :  a / H L, "ă" :  a / L H, "án"
   :  an / H, "àn" :  an / L, "ân" :  an / H L, "ăn" :  an / L H, "í" :  i
   / H, "ì" :  i / L, "î" :  i / H L, "ĭ" :  i / L H, "ín" :  in / H, "ìn"
   :  in / L, "în" :  in / H L, "ĭn" :  in / L H, "é" :  e / H, "è" :  e /
   L, "ê" :  e / H L, "ĕ" :  e / L H, "én" :  en / H, "èn" :  en / L, "ên"
   :  en / H L, "ĕn" :  en / L H, "ó" :  o / H, "ò" :  o / L, "ô" :  o / H
   L, "ŏ" :  o / L H, "ón" :  on / H, "òn" :  on / L, "ôn" :  on / H L, "ŏn"
   :  on / L H, "ú" :  u / H, "ù" :  u / L, "û" :  u / H L, "ŭ" :  u / L H,
   "ún" :  un / H, "ùn" :  un / L, "ûn" :  un / H L, "ŭn" :  un / L H, "Í"
   :  I / H, "Ì" :  I / L, "Î" :  I / H L, "Ĭ" :  I / L H, "É" :  E / H, "È"
   :  E / L, "Ê" :  E / H L, "Ĕ" :  E / L H.

Associates:  {segment{T}, segment{V}}, {segment{X}, segment{P}}.
```

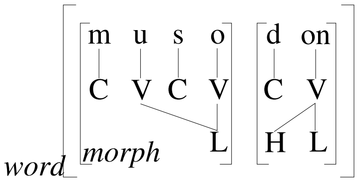


Figure 4-1: Internal representation of *mùsò+dôn*

```
Rules:

Rule "Floating High Tone Metathesis":
Tiers:
        tonal: "]m" H "m[".
Effects:
        H -> "m[" _.

Rule "Floating High Tone Metathesis (part 2)":
Tiers:
        tonal: "]m" H.
Effects:
        H -> _ "]m".

Rule "Initially Connect Tones":
Tiers:
        skeletal: (V) C0 "]m",
           tonal: (T)    "]m".
Effects:
        V :: T.

Rule "Spread Left":
Tiers:
        skeletal: (V) C0 V,
           tonal:        T.
Connections:
        V[2] -- T.
Effects:
        << T skeletal.
```

The parser, upon receiving the above specification, would act in a manner almost identical to the parsing of Abc above, except that the first two rules involve metathesis. Thus, for the first of these rules the replacement chart differs from the original chart in that the "H" moves to the right side of the morpheme begin, and the second in that the "H" moves to the left side of the morpheme end.

Upon receiving the input:

```
w[   m[   musoL   ]m   H   m[   donL   ]m   ]w
```

("It is a woman")[2], the input system enters a word begin into each tier, followed by a morpheme begin. Next, the segments "muso" are entered into the phonemic tier, corresponding to the skeletal "CVCV." "L" is simultaneously added to the tonal tier. Input continues in this manner until the word end is reached.

The first rule matches against the "H" between morpheme boundaries, and moves it into the morpheme "don." Because of this movement, the second rule does not match. The third rule matches

[2]Note that the spaces between characters are actually tabs—AMAR reads spaces as word breaks.

the final "o" in "muso," attaching it to "L." The association convention attempts to apply, but there are no further tone/vowel pairs. Next, the third rule matches the vowel "on" in "don," attaching it to "L." The association convention then attaches the moved "H" to the "on" as well. Finally, the fourth rule applies, spreading the connection from the first low tone to the "u" in "muso," and the output system produces "mùsò+dôn" from the internal representation depicted in figure 4-1.

From the input:

```
w[   m[   musoL   ]m   HL   m[   donL   ]m   ]w
```

("It is the woman") the system behaves much as before, except that the floating "L" blocks the first rule from applying. The second rule thus matches and applies, moving the "H" into the morpheme "muso." Thus, when the third rule applies, "H" is connected to the last vowel in "muso." The association convention now connects the "L" to the "u" in muso. In the morpheme "don," only "L" is connected to the vowel. Thus, no vowels are left unconnected, and the spreading rule does not apply. The output is "mùsó+dòn."

4.3 Spanish

The aspects of Spanish phonology modeled here are continuancy specification in voiced obstruants and nasal/lateral assimilation. In contrast to the previous examples, this section will examine two models for Spanish. In addition, it will be noted that both of these models are incomplete, and a third model will be proposed, but not fully specified. All of these models attempt to explain the following feature of Spanish: voiceless obstruants in Spanish (written "b," "v," "d," and "g") are underlyingly unspecified for the feature [continuant] (Lozano 1978, Goldsmith 1981, Clements 1987). The voiceless obstruants receive a value of [+continuant] in most environments, except for the phrase-initial environment ([-continuant] is preferred, but [+continuant] is optional), the environment following a nasal, and the environment following a lateral, if the obstruant is a coronal.

4.3.1 Matrix Model

The first, and most incomplete, model uses a feature-matrix based, non-autosegmental approach. In this model, a consonant unspecified for [continuant] and preceded by a nasal[3] will become [-continuant]. Otherwise, consonants unspecified for [continuant] will become [+continuant].

This model is specified as follows:

```
Language Spanish:

Phonemes:  a, b, B, "β", "č", d, D, "ð", e, f, g, G, "ɣ", i, x, k, l, "λ",
   m, n, "ñ", o, p, r, rr, s, t, u, w, y.

SpecMethod:  CV/Matrix.

Vowels:  a, e, i, o, u.
Consonants:  b, B, "β", "č", d, D, "ð", f, g, G, "ɣ", x, k, l, "λ", m, n,
   "ñ", p, r, rr, s, t, w, y.

Features:  high, low, back, round, cont, son, ant, cons, nasal, cor, delrel,
   stri, voice, asp, lat.

Defaults:

       any -> [-nasal, -low, -back, -high, -stri, -delrel, -asp, -lat,
       -round, -voice],
```

[3]Here represented as a skeletal "C" segment connected to a feature matrix containing the feature [+nasal].

```
        vowel -> [-cons, +cont, +son, -round, -ant, -cor],

        a -> [+low, +back],
        i -> [+high],
        o -> e [+back],
        u -> i [+back],

        [+back, -low] -> [+round],

        consonant -> [+cor, -son, +cons, +ant, -cont],

        p -> [-cor],
        b -> p [+voice],
        f -> p [+cont],
        m -> p [+son, +nasal],
        B -> b [cont],
        "β" -> b [+cont],

        w -> u,
        y -> i,

        d -> [+voice],
        n -> [+son],
        s -> [+cont],
        "č" -> [-ant, +delrel, +stri],
        D -> d [cont],
        "ð" -> d [+cont],

        [+cont, -voice] -> [+stri],

        r -> n [+cont],
        n -> [+nasal],
        l -> r [+lat],

        rr -> r [+stri],

        "ñ" -> n [-ant],
        "λ" -> l [-ant],

        [+son] -> [+voice],

        k -> [-ant, -cor],
        g -> k [+voice],
        x -> k [+cont],
        G -> g [cont],
        "ɣ" -> g [+cont].

ToneLevels:  0.

Rules:

Rule "Continuancy 1":
NoWordBounds
Tiers:
  phonemic: [+nasal]  ([cont]),
  skeletal:     C        C.
```

```
Connections:
  [+nasal] -- C[1],
  [cont] -- C[2].
Effects:
  [cont] -> [-cont].

Rule "Continuancy 2":
Tiers:
  phonemic: ([cont]),
  skeletal:    C.
Connections:
  [cont] -- C.
Effects:
  [cont] -> [+cont].
```

The parsing process for this specification occurs similarly to that of the previous examples until the keyword `Features` is reached. When the parser reaches this section, each identifier will be assigned to a feature and stored in the string table. The parser will then, in the string table, go through every phoneme defined above and represent it as an empty feature matrix. At the defaults section, the parser will first go through every phoneme and copy into its matrix the features [-nasal], [-low], [-back], [-high], [-stri], [-delrel], [-asp], [-lat], [-round], and [-voice]. Then, it will go through only the vowels in the string table and copy into the matrices the features [-cons], [+cont], etc. At the line "`a -> ...`," the parser will copy the specified features into the matrix for the phoneme "a," as it will do for all other lines of this type. At the line "`[+back, -low] -> [+round]`," the parser will search through the table, and for every phoneme specified [+back, -low], it will copy in [+round]. At the line "`b -> p [+voice]`," the parser will copy the specification of "p" into "b," adding the feature [+voice]. The parser will then proceed similarly until it reaches the keyword "ToneLevels," at which point it will note in the chart that there are no tones and go on to define rules. The first rule will be defined to ignore word boundaries (as is typical for postlexical rules), and will be represented as three phonemic and three skeletal tiers. In the "original" phonemic tier there will be a feature matrix containing the feature [+nasal] and a feature matrix marked for exact matching containing the feature [cont] (*i.e.,* the rule will match a segment specified [+nasal] followed by a segment unspecified for [continuant]). The "replacement" and map phonemic tiers will contain [-cont] instead of [cont], and all three skeletal tiers will contain two C segments. Thus, this rule will replace an unspecified [continuant] feature will [-continuant] in the environment described. The last rule, which does not need to ignore word boundaries, is simply defined to replace unspecified [continuant] features with [+continuant].

Upon receiving the input *su DeDo* ("his/her/your/their finger"), the system will build the representation depicted in figure 4-2. Since there are no nasals in this input, the first rule will not apply, and the second rule will apply to the two "D" 's, since both are unspecified for the feature [continuant]. Thus, the chart will now contain the representation depicted in figure 4-3, and the output will be *su ðeðo.*

From the input *un DeDo* ("a finger"), the system will build the representation depicted in figure 4-4. The first rule will match the "D" after "n," specifying it [-continuant]. Next, the second rule will match the other "D" and specify it [+continuant]. The chart will now contain the representation depicted in figure 4-5, and the output will be *su deðo.*

4.3.2 Tree Model

The second model uses an autosegmental approach to represent the relevant features of Spanish phonology more completely, and more elegantly. The model makes use of two rules unmodeled by the previous example. The first is that a nasal consonant in syllable-final position will receive its point of articulation from the consonant to its right (Harris 1984). This process is here modeled by assimilation of the following consonant's place node. The second rule states that a lateral segment assimilates the point of articulation of a following coronal consonant (Harris 1969). Using these two

	−high	+high	−high	−high	−high	−high
	−low	−low	−low	−low	−low	−low
	−back	+back	−back	−back	−back	+back
	−round	+round	−round	−round	−round	+round
	+cont	+cont	cont	+cont	cont	+cont
	−son	+son	−son	+son	−son	+son
	+ant	−ant	+ant	−ant	+ant	−ant
	+cons	−cons	+cons	−cons	+cons	−cons
	−nasal	−nasal	−nasal	−nasal	−nasal	−nasal
	+cor	−cor	+cor	−cor	+cor	−cor
	−delrel	−delrel	−delrel	−delrel	−delrel	−delrel
	+stri	−stri	−stri	−stri	−stri	−stri
	−voice	+voice	+voice	+voice	+voice	+voice
	−asp	−asp	−asp	−asp	−asp	−asp
	−lat	−lat	−lat	−lat	−lat	−lat
word	C	V	C	V	C	V

Figure 4-2: Internal Representation of "su DeDo"

	−high	+high	−high	−high	−high	−high
	−low	−low	−low	−low	−low	−low
	−back	+back	−back	−back	−back	+back
	−round	+round	−round	−round	−round	+round
	+cont	+cont	$\boxed{\text{+cont}}$	+cont	$\boxed{\text{+cont}}$	+cont
	−son	+son	−son	+son	−son	+son
	+ant	−ant	+ant	−ant	+ant	−ant
	+cons	−cons	+cons	−cons	+cons	−cons
	−nasal	−nasal	−nasal	−nasal	−nasal	−nasal
	+cor	−cor	+cor	−cor	+cor	−cor
	−delrel	−delrel	−delrel	−delrel	−delrel	−delrel
	+stri	−stri	−stri	−stri	−stri	−stri
	−voice	+voice	+voice	+voice	+voice	+voice
	−asp	−asp	−asp	−asp	−asp	−asp
	−lat	−lat	−lat	−lat	−lat	−lat
word	C	V	C	V	C	V

Figure 4-3: Internal Representation of Output from "su DeDo"

word

V	C	C	V	C	V
+high	−high	−high	−high	−high	−high
−low	−low	−low	−low	−low	−low
+back	−back	−back	−back	−back	+back
+round	−round	−round	−round	−round	+round
+cont	−cont	cont	+cont	cont	+cont
+son	+son	−son	+son	−son	+son
−ant	+ant	+ant	−ant	+ant	−ant
−cons	+cons	+cons	−cons	+cons	−cons
−nasal	+nasal	−nasal	−nasal	−nasal	−nasal
−cor	+cor	+cor	−cor	+cor	−cor
−delrel	−delrel	−delrel	−delrel	−delrel	−delrel
−stri	−stri	−stri	−stri	−stri	−stri
+voice	+voice	+voice	+voice	+voice	+voice
−asp	−asp	−asp	−asp	−asp	−asp
−lat	−lat	−lat	−lat	−lat	−lat

Figure 4-4: Internal Representation of "un DeDo"

word

V	C	C	V	C	V
+high	−high	−high	−high	−high	−high
−low	−low	−low	−low	−low	−low
+back	−back	−back	−back	−back	+back
+round	−round	−round	−round	−round	+round
+cont	−cont	−cont	+cont	+cont	+cont
+son	+son	−son	+son	−son	+son
−ant	+ant	+ant	−ant	+ant	−ant
−cons	+cons	+cons	−cons	+cons	−cons
−nasal	+nasal	−nasal	−nasal	−nasal	−nasal
−cor	+cor	+cor	−cor	+cor	−cor
−delrel	−delrel	−delrel	−delrel	−delrel	−delrel
−stri	−stri	−stri	−stri	−stri	−stri
+voice	+voice	+voice	+voice	+voice	+voice
−asp	−asp	−asp	−asp	−asp	−asp
−lat	−lat	−lat	−lat	−lat	−lat

Figure 4-5: Internal Representation of Output from "un DeDo"

rules, the current model postulates (Goldsmith 1990, pages 70–71) that a consonant unspecified for [continuant] assimilates the value of [continuant] of the previous consonant, if the two consonants share the same place node. Just as in the previous example, consonants unspecified for [continuant] default to [+continuant].

The model would be specified as follows:

```
Language Spanish:

Phonemes:  a, b, B, "β", "č", d, D, "ð", e, f, g, G, "ɣ", i, x, k, l, "λ",
   m, n, "ñ", "ŋ", o, p, r, rr, s, t, u, w, y.

SpecMethod:  CV/Tree.

Vowels:  a, e, i, o, u.
Consonants:  b, B, "β", "č", d, D, "ð", f, g, G, "ɣ", x, k, l, "λ", m, n,
   "ñ", "ŋ", p, r, rr, s, t, w, y.

Tree {

       {root :  skeletal},
       {stricture :  root :  [cons], [son], [cont], [stri], [lat]},
       {laryngeal :  root :  [voice], [delrel]},
       {supralaryngeal :  root},
       {softpalate :  supralaryngeal :  [nasal]},
       {place :  supralaryngeal},
       {labial :  place :  [round]},
       {coronal :  place :  [ant]},
       {dorsal :  place :  [high], [low], [back]}

}

Defaults:

       any -> segment{root : segment{stricture : segment{cons},
                                                 segment{son},
                                                 segment{cont},
                                                 segment{stri},
                                                 segment{lat}},
                             segment{laryngeal : segment{voice},
                                                 segment{delrel}},
                             segment{supralaryngeal :
                                     segment{softpalate : segment{nasal}},
                                     segment{place}}},

       vowel -> segment{place : segment{labial : segment{-round}},
                                segment{dorsal : segment{-high},
                                                 segment{-low},
                                                         segment{-back}}},

       vowel -> [+cont, +son, -nasal, -cons, -stri, -lat, +voice, -delrel],

       a -> [+low, +back],
       i -> [+high],
       o -> [+back],
       u -> [+high, +back],
```

```
        [+back, -low] -> [+round],

        consonant -> [-cont, -son, -nasal, +cons, -stri, -lat, -voice,
        -delrel],

        p -> segment{place :  segment{labial :  segment{-round}}},
        b -> p [+voice],
        f -> p [+cont],
        m -> p [+son, +nasal],
        B -> b [cont],
        "β" -> b [+cont],

        w -> u,
        y -> i,

        t -> segment{place :  segment{coronal :  segment{+ant}}},
        d -> t [+voice],
        n -> t [+son],
        s -> t [+cont],
        "č" -> t [-ant, +delrel, +stri],
        D -> d [cont],
        "ð" -> d [+cont],

        [+cont, -voice] -> [+stri],

        r -> n [+cont],
        l -> n [+lat],
        n -> [+nasal],

        rr -> r [+stri],

        "ñ" -> n [-ant],

        "λ" -> l [-ant],

        k -> segment{place :  segment{dorsal :  segment{+high}, segment{-low},
        segment{+back}}},
        "ŋ" -> k [+son, +nasal],
        g -> k [+voice],
        x -> k [+cont],
        G -> g [cont],
        "ɣ" -> g [+cont],

        [+son] -> [+voice].

ToneLevels:  0.

Rules:

Rule "Nasal Assimilation":
NoWordBounds
Tiers:
     place:(place)    place,
  skeletal:   C         C,
     nasal: +nasal .
Connections:
```

```
  place[1] -- C[1],
  place[2] -- C[2],
  C[1] -- +nasal.
Effects:
  place[1] -Z- C[1],
  C[1] :: place[2].

Rule "Lateral Assimilation":
NoWordBounds
Tiers:
   coronal:          coronal,
     place:  place    place,
  skeletal:    C        C,
       lat:  +lat.
Connections:
  place[1] -- C[1],
  C[1] -- +lat,
  coronal -- place[2],
  place[2] -- C[2].
Effects:
  place[1] -Z- C[1],
  C[1] :: place[2].

Rule "Continuancy 1":
NoWordBounds
Tiers:
     place:     (place),
  skeletal:   C         C,
      cont: -cont      cont.
Connections:
  place -- C[1],
  place -- C[2],
  C[1] -- -cont,
  C[2] -- cont.
Effects:
  C[2] :: -cont,
  C[2] -Z- cont.

Rule "Continuancy 2":
Tiers:
      cont:(cont),
  skeletal:  C.
Connections:
  cont -- C.
Effects:
  cont -> +cont.
```

In the previous examples, there have automatically been three tiers: the skeletal tier, the phonemic, and the tonal. In this example, however, the parser will indeed create those three tiers automatically, but, upon reading the `Tree` section on the specification file, it will create tiers for each class node and feature defined there. In addition, it will create and place in the chart the tree structure depicted in figure 4-6. In the default section most lines behave similarly to those in the matrix example, and those that contain `segmentspec`s build tree structures. Thus, every phoneme first receives a generic tree structure, then vowels add a dorsal and labial node, and the various

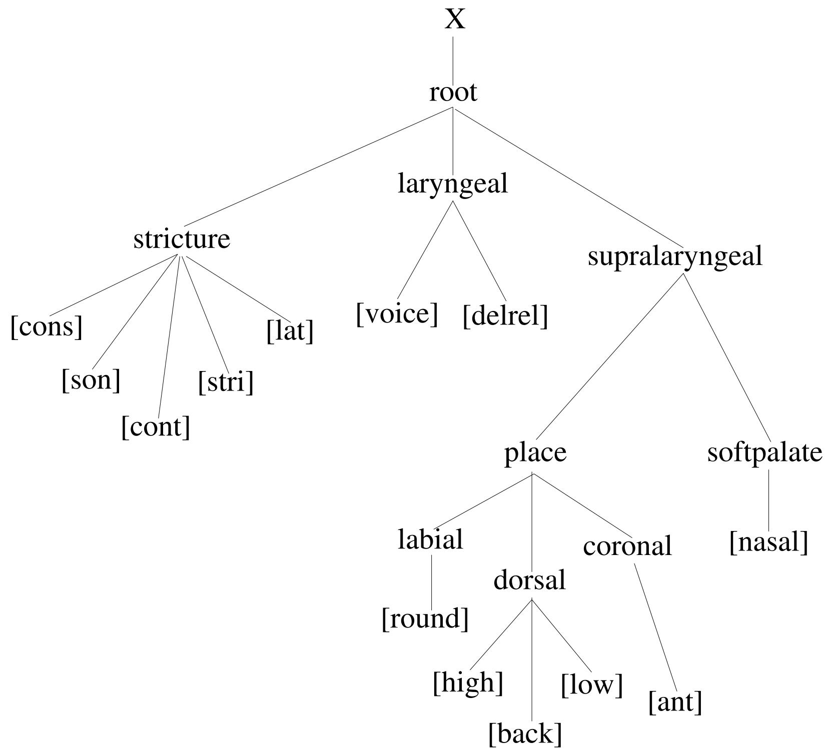


Figure 4-6: Tree Structure

consonants add the appropriate place nodes. The rules are as discussed above, and are build in the same manner as in the previous examples.

Upon receiving the input "un Beso," the system will build the structure depicted in figure 4-7. The rule of nasal assimilation will match "n B" and produce the structure shown in figure 4-8. Because the "n" (now an "m," sharing the "B" 's labial point of articulation) shares a place node with "B," the first continuancy rule will apply, and "B" will become [-continuant]. Thus, the chart will contain the structure shown in figure 4-9, and the system will produce the output "um beso." Given the input "su Beso," the nasal assimilation rule will not apply, so the second continuancy rule will apply instead of the first, and the output will be "su βeso." Similarly, "al Gato" produces "al ɣato" (since "G" is not coronal, and therefore the lateral assimilation rule does not apply, thus disallowing the first continuancy rule), "al DeDo" produces "al deðo" (since "D" *is* coronal and the lateral assimilation applies, with similar results to the case of "un Beso"), and "al λano" produces "aλ λano" (since "λ" is a palatal, and thus its place node contains the feature [-anterior], which becomes shared by the previous consonant.)

4.3.3 Hypothetical Model

As it turns out, the data for Spanish suggest that the previous model is incorrect (Personal Communication, Harris). There are cases in which the nasal assimilation rule is blocked, but the following voiced obstruant still becomes a [-continuant]. Thus, another model might postulate that voiceless obstruants assimilate the continuancy of the previous sonorant, with a default value of [-continuant]. Thus, phrase-initial obstruants would be [-continuant], as predicted by none of the previous models but supported by the data. Voiceless obstruants preceded by vowels would become [+continuant], and when preceded by [-continuant] sonorants (such as "l" and "n," but not "r") they would become [-continuant]. The rules for this model would be as follows:

```
Rules:
```

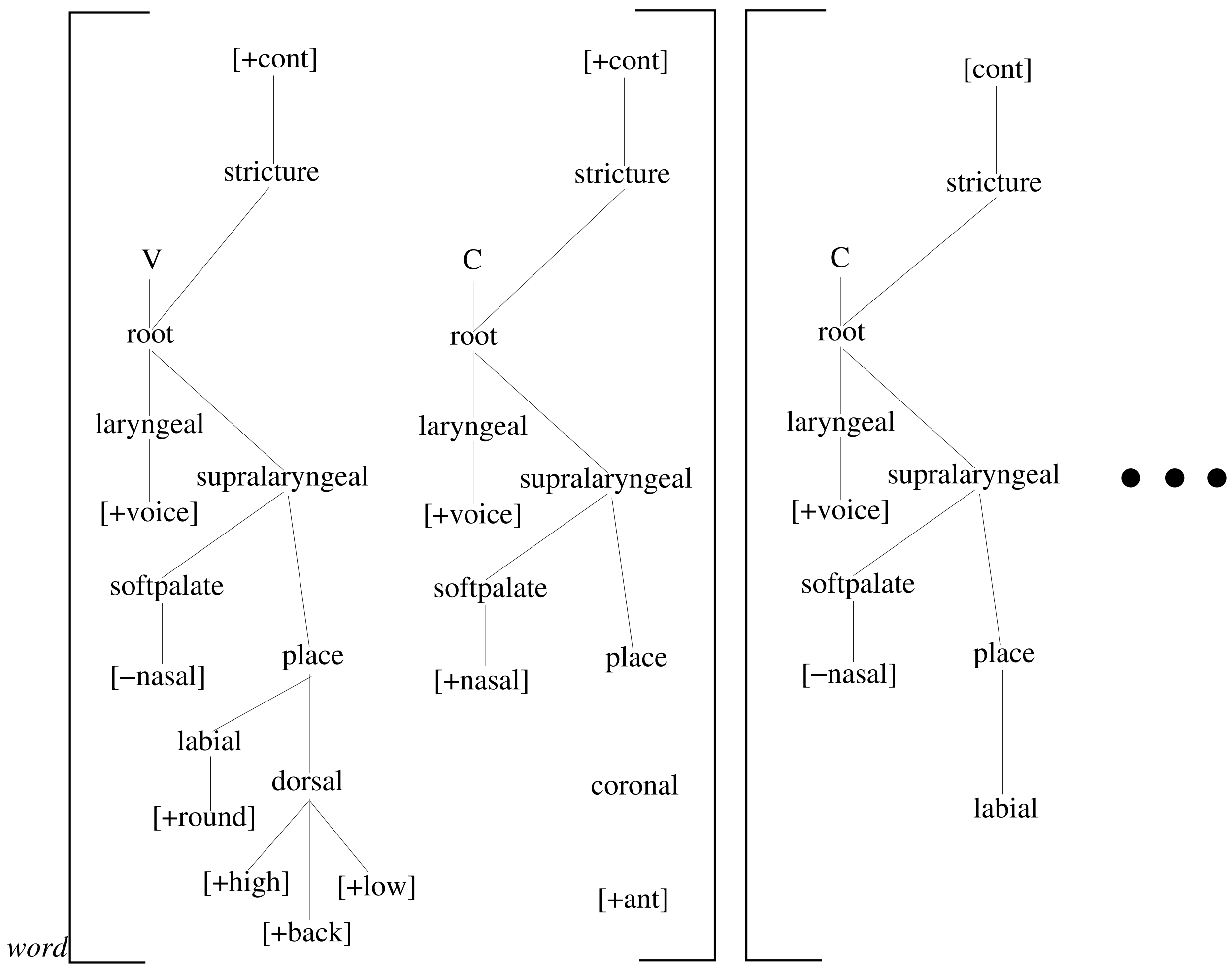


Figure 4-7: Partial Representation of "un Beso"

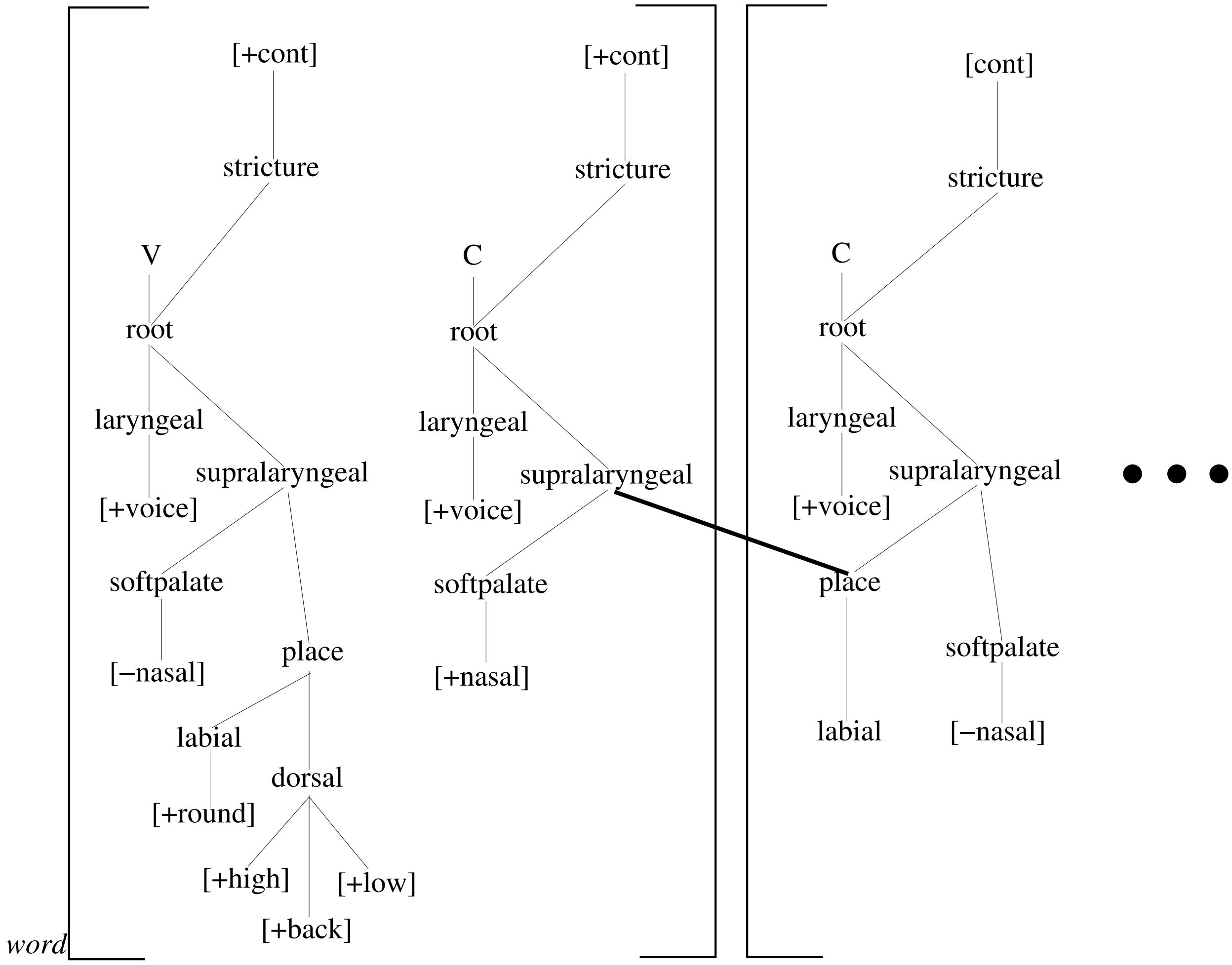


Figure 4-8: "un Beso" after Application of Nasal Assimilation

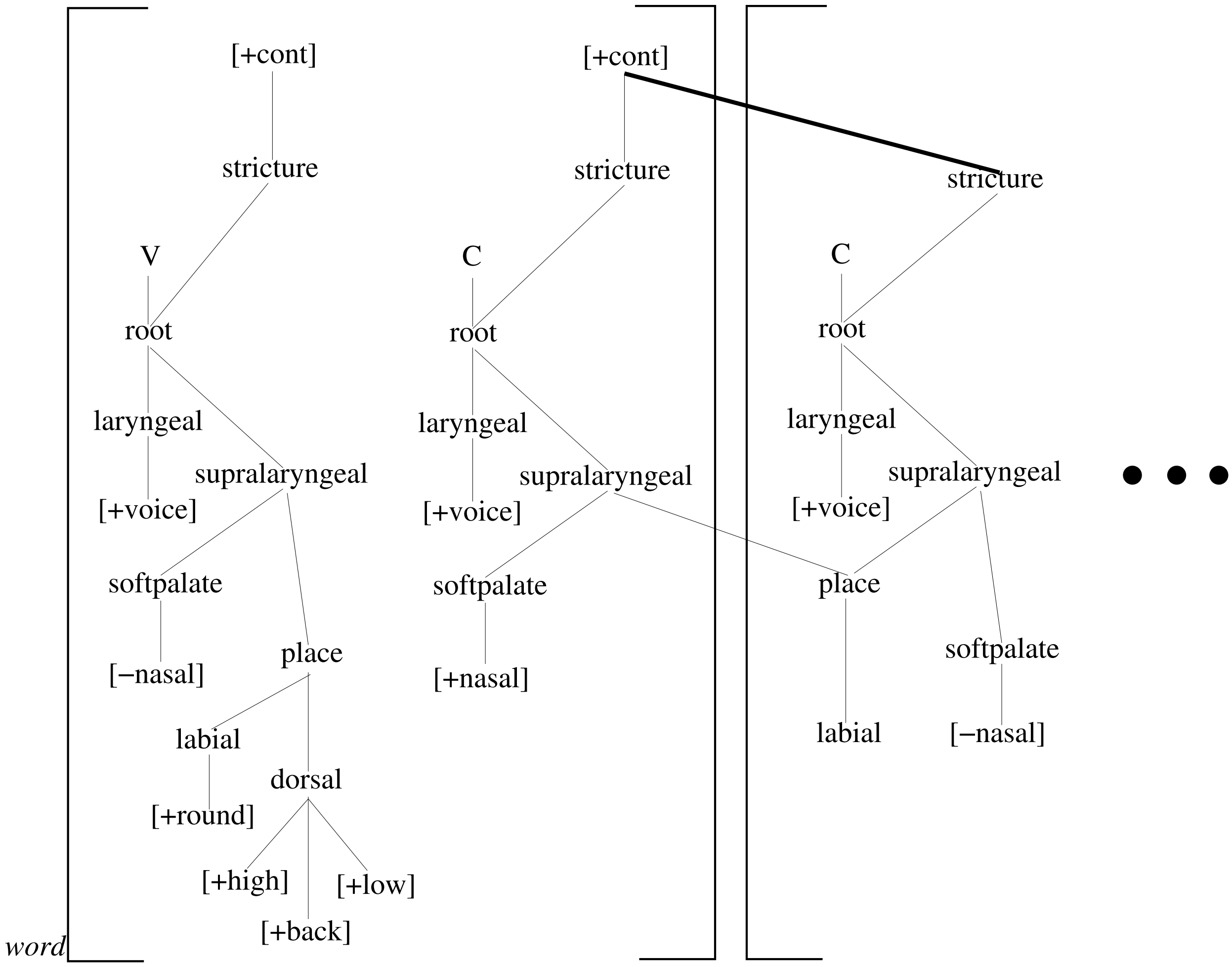


Figure 4-9: "un Beso" after Application of Continuancy Rule One

```
Rule "Nasal Assimilation":
NoWordBounds
Tiers:
     place:(place)    place,
  skeletal:   C         C,
     nasal: +nasal .
Connections:
  place[1] -- C[1],
  place[2] -- C[2],
  C[1] -- +nasal.
Effects:
  place[1] -Z- C[1],
  C[1] :: place[2].

Rule "Lateral Assimilation":
NoWordBounds
Tiers:
   coronal:          coronal,
     place:  place    place,
  skeletal:    C        C,
       lat:  +lat.
Connections:
  place[1] -- C[1],
  C[1] -- +lat,
  coronal -- place[2],
  place[2] -- C[2].
Effects:
  place[1] -Z- C[1],
  C[1] :: place[2].

Rule "Continuancy 1":
NoWordBounds
Tiers:
       son: +son,
  skeletal:   X    C0   C,
      cont: @cont      cont.
Connections:
  +son -- C[1],
  C[1] -- @cont,
  C[2] -- cont[2].
Effects:
  C[2] :: @cont,
  C[2] -Z- cont[2].

Rule "Continuancy 2":
Tiers:
      cont:(cont),
  skeletal:  C.
Connections:
  cont -- C.
Effects:
  cont -> -cont.
```

Conjugation	Output Form
I	katab
II	kattab
III	kaatab
IV	?aktab
V	takattab
VI	takaatab
VII	nkatab
VIII	ktatab
IX	ktabab
X	staktab
XI	ktaabab
XII	ktawbab
XIII	ktawwab
XIV	ktanbab
XV	ktanbay

Table 4.1: Conjugation of *ktb* in Classical Arabic

Note that the segment reference for [cont] in the first continuancy rule has the index "[2]." This index is necessary because both `@cont` and `cont` match the specification `cont` (since features specified with `@` match any other feature of the same name, regardless of value), so the "[2]" signifies that the second feature matching `cont` is desired. Note also that this hypothetical rule cannot actually be implemented in the current system, because one needs to specify where the skeletal contains C0 that the matching system should skip over all obstruant consonants following the sonorant that are specified for continuant. The notation for this does not yet exist in AMAR, but will be implemented in the next version.

4.4 Arabic

The final example will deal with verbal conjugation in Classical Arabic, following to some extent McCarthy (1975) and Goldsmith (1990). There are fifteen basic "conjugations" in Classical Arabic, each one consisting of a pattern of consonant and vowel positions. The actual vowels making up a conjugated verb depend on the tense and voice, and the consonants are determined lexically. Thus, the morpheme *ktb* would be "conjugated" as in table 4.1.

In the case modeled here, and shown in the table, the verb is conjugated in the active perfective, so there is only one vowel, "a," that spreads across all vowel slots. Thus, for any given conjugation, a rule would apply to build the proper consonantal and vowel positions on the skeletal tier, the consonants would be connected to the C slots by means of a connective rule and the association convention, and the vowel "a" would be connected to the vowel slots. In addition, some of the conjugations include prefixes or infixes, which would be added afterwards.

The (somewhat incomplete) AMAR specification for Arabic is as follows:

```
Language Arabic:

Phonemes:  b, f, "θ", "ð", m, t, d, s, z, n, l, r, k, "ǰ", "š", q, x, "ɣ",
   "ħ", "ʔ", "ʕ", h, y, w.

SpecMethod:  CV/Tree.

Vowels:  i, a, u.
```

```
Consonants:  b, f, "θ", "ð", m, t, d, s, z, n, l, r, k, "ǰ", "š", q, x, "ɣ",
   "ħ", "ʔ", "ʕ", h, y, w.

Tree {

       {vroot :  skeletal},
       {croot :  skeletal :  [son]},
       {cons :  croot :  [cont], [lat], [stri]},
       {"soft palate" :  croot :  [nasal]},
       {guttural :  croot},
       {"tongue root" :  guttural :  [RTR]},
       {larynx :  guttural :  [stiff], [slack], [constr], [spread]},
       {place :  croot},
       {place :  vroot},
       {coronal :  place :  [ant], [dist]},
       {peripheral :  place},
       {dorsal :  peripheral :  [back], [high], [low]},
       {labial :  peripheral :  [rounded]}

  }

Defaults:

       vowel -> segment{vroot : segment{place :
                                          segment{peripheral :
                                             segment{dorsal :
                                                       segment{+back},
                                                       segment{+low}}}}},

       i -> [-back, -low], u -> [-low],

       consonant -> segment{croot : segment{-son},
                               segment{cons : segment{-cont},
                                              segment{-lat},
                                              segment{-stri}},
                               segment{"soft palate" : segment{-nasal}},
                               segment{guttural :
                                       segment{"tongue root":
                                               segment{-RTR}},
                                       segment{larynx :
                                               segment{-stiff},
                                               segment{+slack},
                                               segment{-constr},
                                               segment{-spread}}},
                               segment{place}},

       b -> segment{place :  segment{peripheral :  segment{labial}}},
       b -> [+stiff],
       f -> b [+cont, -stiff],
       m -> b [+son, +nasal],

       y -> segment{croot :  segment{+son}, segment{place}},
       y -> segment{place :  segment{coronal :  segment{-ant}, segment{+dist}}},
       w -> y,
       w -> segment{place :  segment{peripheral :  segment{labial}}},
```

```
       t -> segment{place :  segment{coronal :  segment{+ant}, segment{+dist}}},
       d -> t [+stiff],
       "θ" -> t [+cont],
       "ð" -> d [+cont],
       s -> "θ" [+stri],
       z -> "ð" [+stri],
       n -> d [+son, +nasal],
       l -> d [+son, +lat],
       r -> d [+son, -dist, +cont],
       "š" -> s [-ant],
       "ǰ" -> z [-ant],

       k -> segment{place :  segment{peripheral :  segment{dorsal}}},

       q -> k [+RTR],
       x -> q [+cont],
       "ɣ" -> x [+stiff],

       "ħ" -> [+constr],
       "ʕ" -> "ħ" [+stiff],

       "ʔ" -> [+constr, +stiff, -slack],
       h -> [+spread].

ToneLevels:  15.

NonAssociates:  {segment{X}, segment{croot}}, {segment{X}, segment{vroot}}.
   Associates:  {segment{V}, segment{vroot}}, {segment{C}, segment{croot}}.

Rules:

Rule "Skeletal Cleansing 1":
Tiers:
  skeletal:   C,
     croot: croot.
Connections:
  C -- croot.
Effects:
  C -Z- croot,
  C -> 0.

Rule "Skeletal Cleansing 2":
Tiers:
  skeletal:   V,
     vroot: vroot.
Connections:
  V -- vroot.
Effects:
  V -Z- vroot,
  V -> 0.

Rule "Conjugation 1":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 1 "]w".
```

```
Effects:
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 2":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 2 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 3":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 3 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 4":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 4 "]w".
Effects:
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 5":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 5 "]w".
Effects:
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].
```

```
Rule "Conjugation 6":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 6 "]w".
Effects:
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 7":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 7 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 8":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 8 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 9":
Tiers:
  skeletal:"w["   "]w",
     tonal:"w[" 9 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 10":
Tiers:
  skeletal:"w["    "]w",
     tonal:"w[" 10 "]w".
Effects:
```

```
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 11":
Tiers:
  skeletal:"w["    "]w",
     tonal:"w[" 11 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 12":
Tiers:
  skeletal:"w["    "]w",
     tonal:"w[" 12 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 13":
Tiers:
  skeletal:"w["    "]w",
     tonal:"w[" 13 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 14":
Tiers:
  skeletal:"w["    "]w",
     tonal:"w[" 14 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
```

```
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> C / _ "]w"[1].

Rule "Conjugation 15":
Tiers:
  skeletal:"w["    "]w",
     tonal:"w[" 15 "]w".
Effects:
  0 -> C / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> /C/ / _ "]w"[1],
  0 -> C / _ "]w"[1],
  0 -> V / _ "]w"[1],
  0 -> /C/ / _ "]w"[1].

Rule InitiallyConnect:
Tiers:
     croot: "w["  (croot),
  skeletal: "w[" V0 (C).
Effects:
  C :: croot.

Rule "Conjugation 4":
Tiers:
     croot: "w[",
  skeletal: "w[",
     tonal: "w[" 4.
Effects:
  0 -> "?" / "w["[1,croot] _,
  "?" ::-> C / "w["[1,skeletal] _.

Rule "Conjugation 5":
Tiers:
     croot: "w[",
  skeletal: "w[",
     tonal: "w[" 5.
Effects:
  0 -> t / "w["[1,croot] _,
  t ::-> C / "w["[1,skeletal] _.

Rule "Conjugation 6":
Tiers:
     croot: "w[",
  skeletal: "w[",
     tonal: "w[" 6.
Effects:
  0 -> t / "w["[1,croot] _,
  t ::-> C / "w["[1,skeletal] _.

Rule "Conjugation 7":
Tiers:
     croot: "w[",
```

```
  skeletal: "w[",
     tonal: "w[" 7.
Effects:
  0 -> n / "w["[1,croot] _,
  n ::-> C / "w["[1,skeletal] _.

Rule "Conjugation 8":
Tiers:
     croot: "w["croot,
  skeletal: "w["  C   (C),
     tonal: "w["  8.
Effects:
  C[2] ::-> t /  croot _.

Rule "Conjugation 10":
Tiers:
     croot: "w[",
  skeletal: "w[",
     tonal: "w["  10.
Effects:
  0 -> t / "w["[1,croot] _,
  0 -> s / "w["[1,croot] _,
  croot[2] ::-> C / "w["[1,skeletal] _,
  croot[1] ::-> C / "w["[1,skeletal] _.

Rule "Conjugation 12":
Tiers:
     croot:      croot,
  skeletal: (C) V0 C,
     tonal: 12.
Connections:
  C[2] -- croot.
Effects:
  C[1] ::-> w / _ croot.

Rule "Conjugation 13":
Tiers:
     croot:      croot,
  skeletal: (C) V0 C,
     tonal: 13.
Connections:
  C[2] -- croot.
Effects:
  C[1] ::-> w / _ croot.

Rule "Conjugation 14":
Tiers:
     croot:    croot,
  skeletal: (C)  C,
     tonal: 14.
Connections:
  C[2] -- croot.
Effects:
  C[1] ::-> n / _ croot.
```

```
Rule "Conjugation 15":
Tiers:
     croot:    croot,
  skeletal: (C)  C,
     tonal: 15.
Connections:
  C[2] -- croot.
Effects:
  C[1] ::-> n / _ croot.

Rule "Conjugation 15":
Tiers:
     croot:     "]w",
  skeletal: (C) "]w",
     tonal: 15  "]w".
Effects:
  C ::-> j / _ "]w"[1].

Rule InsertA:
Tiers:
    vroot: "w["     "]w",
 skeletal: "w[" C0 (V).
Effects:
  V ::-> a / _ "]w".

Rule "SpreadRight":
Tiers:
     vroot: vroot,
  skeletal:   V C0 (V).
Connections:
  vroot -- V[1].
Effects:
  vroot >> skeletal.

Rule Geminate:
Tiers:
     croot:   croot,
  skeletal: (C) C.
Connections:
  croot -- C[2].
Effects:
  croot :: C[1].

Rule SpreadCRight:
Tiers:
     croot: croot,
  skeletal:   C V0 (C).
Connections:
  croot -- C[1].
Effects:
  croot >> skeletal.
```

Despite the large number of rules and the complicated consonants[4], most of the example is dealt with in exactly the same manner as the Spanish example. The only major features utilized here and not anywhere else are controls for associativity. The model used here states that consonants and vowels are found on different tiers (*croot* and *vroot*, respectively), and therefore it is not appropriate for C segments to freely associate with *vroot*s or V segments with *croot*s. However, when the tree is defined, AMAR automatically assumes that *croot* and *vroot*, as inferiors of the skeletal tier, freely associate with all skeletal segments. Thus, the `NonAssociates` section is used to state that X segments (*i.e.*, any skeletal segments) do not associate freely with *croot* and *vroot*, and the `Associates` section states that C associates freely with *croot*, and V with *vroot*. In addition, use is made of *inert* segments, segments defined to be ignored during the association convention. For example, the second conjugation builds the skeletal segments:

C V /C/ C V C

where the second C segment is marked as inert. Given the input *ktb*, when the "k" is connected to the first C, the association convention will not see the second consonant position, and will thus connect the "t" to the third C segment. This allows a later gemination rule to spread the "t" to the second C as well. For input, this model expects simply a consonant morpheme (such as *ktb* 'write' or *fʔl* 'do') along with a number (represented as a floating tone) indicating the desired conjugation. From this, it will build an appropriate skeletal tier, connect things appropriately, connect the vowel "a," and output the conjugated form. Thus, upon receiving the input "fʔl2," the system will build the skeletal segments above, connect the consonants appropriately and add "a" to produce the output "faʔʔal." Similarly, if the input had been "ktb2," the output would be "kattab." If the system were to receive a "sentence" (*i.e.*, a number of words separated by spaces and ended with a period) such as "ktb3 fʔl4 ktb14 fʔl10.", the output would be "kaatab ʔafʔal ktanbab stafʔal."

[4]Note that the tree used here is a slightly modified rendition of that presented in Halle (1993).

Chapter 5

Discussion

The current system, while powerful and fairly useful, has nonetheless many possibilities that have not yet been fully explored. A number of improvements are envisioned, to be carried out by the author and perhaps later users. However, there remains a multitude of uses for this version of AMAR, and for future versions, in the fields of phonology, morphology, phonetics, and speech generation.

5.1 Improvements and Extensions

In the future, improvements might be sought in the areas of interface, linguistic accuracy and generality, and reversibility. As one can see from the many diagrammatic representations of various aspects of this thesis, autosegmental phonology lends itself rather poorly to a text-based interface and most likely rather well to a graphical one. The user might, for example, wish to specify rules in the exact conventional notation rather than a text-based representation. Thus, a user-interface could allow the user to enter the segments in a tier and draw various lines to connect them. The general tree structure of a language might be specified simply by drawing it. It would be fairly straightforward to implement such an interface as the one described here. The user could then choose to edit whichever representation best suits his or her present purpose.

Currently, many aspects and mechanics of autosegmental theory remain unsupported. For example, several recent papers (Halle 1993, Keyser and Stevens 1993) make use of "pointers," which are directed from one class node to another in order to indicate by which articulator various features such as continuancy are implemented[1]. The system contains some internal support for pointers, but the mechanics of pointers has not been fully fleshed out within the system, and no specification mechanism yet exists for the user. In addition to pointers, the system does not support headed trees (as used in stress assignment), feature-containing class nodes (Halle 1993), the notion of "markedness" (there are some rules that do not "see" unmarked features, and some that do (Halle, personal communication)), and a number of other theoretical and mechanical aspects of autosegmental phonology. Many of these problems could be solved simply by allowing the user to specify the contents of newly defined tiers. For example, if a class tier were allowed to contain binary valued segments, headed trees would be possible with the current system. In addition, class tiers containing feature matrices would be equivalent to feature-containing class nodes. Finally, markedness could be implemented simply by providing a flag in feature segments specifying whether or not they are marked, as well as one for rules to specify whether or not they ignore unmarked features.

Leaving aside the problem of unsupported aspects of autosegmental theory, there still exists a number of inadequacies in the features supported by AMAR. For example, word and morpheme boundaries are treated as individual segments. For a given word boundary, the system inserts one segment for every tier in the chart. If the user wishes to delete a boundary, each tier's segment must be individually deleted. This program would be fairly simple to fix, by linking all segments

[1] The concept of pointers originates in Sagey (1986).

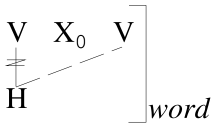


Figure 5-1: Digo End Run Rule

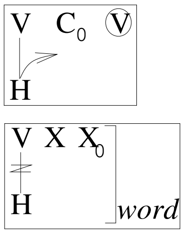


Figure 5-2: Restatement of Digo End Run

corresponding to a given boundary and deleting the entire group when the user indicates that the boundary should be deleted. Another flaw of the current system lies in matching. In Digo, a Bantu language of northeastern Tanzania, there is a rule of End Run (Kisseberth 1984), in which a high tone reässociates to the final vowel of a word (see figure 5-1.) This rule requires the system to match "X_0 V]w," interpreted as "zero or more skeletal segments, followed by a vowel at the end of a word." However, actually implementing this matching behavior is fairly difficult (since "X_0" matches the final "V" as well), and the current system, upon encountering C_0, V_0, or X_0, simply moves forward up to the first non-matching segment—for example, the current system could handle "X_0]w", but not "X_0 V]w", or "C_0 V", but not "C_0 C." Presumably numerous other problems of this nature will be discovered and, it is to be hoped, eliminated during the use of the system. It may be tentatively hypothesized, however, that any example using this sort of matching might better be recast to avoid it. For example, the rule of end run could be restated as in figure 5-2. This restatement would further hypothesize—most likely reasonably—that end run is blocked before vowels already connected to tones.

As of now, AMAR handles generation but not recognition. That is, the user may provide an underlying representation and be returned the surface representation for a given utterance, but the reverse is not possible. Since rules in AMAR are symmetrical (each rule stores a "before" and "after" representation of the chart, and application changes the chart from a "before" state to an "after" state), any rule whose application does not cause information loss could be reversed simply by switching the "original" and "replacement" charts. In addition, one could provide rules specifically marked only to apply during generation, or only during recognition. Thus, it would not be too difficult to provide a system which could take its own surface output and recover the lexical form. However, converting from the autosegmental notation to the type of notation in which AMAR receives its input involves a great deal of information loss. For example, given a simple string of the type AMAR outputs, the system would have to somehow guess whether two adjoining symbols that are similar in some way are such accidentally or as a result of some assimilation process. This problem could perhaps, however, be addressed by strict compliance with well-formedness conditions such as the obligatory contour principle, which states, more or less, that "At the melodic level (*i.e.*, on non-skeletal tiers), adjacent identical elements are prohibited" (McCarthy 1986). Thus, in the example given before, the system would always assume that what appears in the output as two segments with some similarity would actually be represented as two skeletal positions sharing some portion of their information—for example, the output form "mb" would be presumed to share a place node, since both are labial.

5.2 Possible Uses for the System

Regardless of the imperfections of the current system, AMAR presents a number of possibilities for applications in phonology, morphology, phonetics, and computerized speech. As stated in the introduction, AMAR was designed to allow phonologists to model linguistic systems and check their hypotheses against large bodies of data. Within this general framework, a phonologist attempting, for example, to work out aspects of Universal Phonology might try to model as many languages as possible with a single set of basic mechanics and a single tree structure. Using AMAR it becomes easy to check whether a theoretical change motivated by some particular language still makes the right predictions for the languages previously studied—one simply updates the old specifications and checks against the inputs and expected outputs already generated. Another use for the system is in the areas of computational morphology. Since, as we have seen, AMAR can perform most, if not all, of the computations of which the leading computational morphology system, KIMMO, is capable, it could be used for the applications in which KIMMO is currently employed, adding simplicity, elegance, and an advanced capability for describing complex structures. Work in phonetics, another field touching closely upon phonology, often begins from an underlying phonological representation. Thus, computational phoneticians could use AMAR as a back end to provide such representations. Furthermore, with only minor modifications (such as allowing floating point values in various tiers), AMAR could become a powerful mechanism for phonetics in its own right. For example, the user might be able to specify and modify in some way tone frequencies, segment durations, or any number of similar values. Additionally, many phonetic facts may be described in the current version of AMAR (*e.g.,* the Spanish example of section 4.3 is generally said to be phonetic rather than phonological.) Finally, since AMAR converts simple unilinear strings into complex tree representations—representations containing detailed articulatory instructions—it might be possible to set up speech generation systems in which one begins with simple text and uses AMAR to convert the text into articulatory instructions, which would be fed into an articulation-based speech generator such as that developed by Stevens (1992).

Appendix A

Full Grammar for Specification File

The following is a complete description of the grammar for an AMAR language specification file. Words in bold face represent non-terminal symbols. Any symbol in non-bold courier font represents a terminal symbol, any of which would be entered by the user in the form it appears here, except that letters do not have to be in any particular case. The symbol → represents a regular expression, in which expressions in square brackets represent a choice (*e.g.*, [A-Za-z] matches a single alphabetical character, regardless of case.) An expression in square brackets in which the first symbol is a caret matches any character except for the characters found after the caret. An asterisk indicates that there may be zero or more of the previous expression. The symbol ⟶ represents a grammar rule. In both grammar rules and regular expressions, the symbol | represents "or." Finally, in grammar rules, and *not* in regular expressions, large square brackets around an expression indicate that it is optional, and a comma-separated list with ellipses periods (*e.g.*, "**thing**, **thing**, **...**, **thing**") represents a list with one or more items, as would be represented in more standard grammar as:

list ⟶ **list** , **item** |
item

The grammar as shown here attempts to be readable to a human, rather than to a computer, and is not an immediately parsable Backus-Naur Form grammar. However, with a minimum of labor it could be converted into such. For a parsable grammar, the start symbol would be **language**.

identifier → (`[A-Za-z]` (`[A-Za-z]` | **digit**)*) | `"` `[^"\t.+#\n]*` `"`

number → **digit** `[0-9]*`

digit → `[1-9]`

language ⟶ `Language`[1] **identifier**: **langspec**

langspec ⟶ **phonemespec** **tonespec** [**associates**] [**definitions**]
[**rulespec**]

phonemespec ⟶ **phonemelist** **method** **spec**

phonemelist ⟶ `Phonemes:` [**identifier**, **identifier**, **...**, **identifier**] .

method ⟶ `SpecMethod:` `CV` | `CV/Matrix` | `X/Matrix` | `CV/Tree` | `X/Tree` .

spec ⟶ **cvspec** | **cvmatrixspec** | **xmatrixspec** | **cvtreespec** | **xtreespec**

[1] Note that capitalization does not matter for terminal symbols.

cvspec ⟶ [**vowels**] [**consonants**]

cvmatrixspec ⟶ [**vowels**] [**consonants**] [**features**] [**cvmatrixdefaults**] [**cvmatrixfullspecs**]

xmatrixspec ⟶ [**features**] [**xmatrixdefaults**] [**xmatrixfullspecs**]

cvtreespec ⟶ [**vowels**] [**consonants**] [**tree**] [**cvtreedefaults**] [**cvtreefullspecs**]

xtreespec ⟶ [**tree**] [**xtreedefaults**] [**xtreefullspecs**]

vowels ⟶ `Vowels:` [**identifier**, **identifier**, ..., **identifier**] .

consonants ⟶ `Consonants:` [**identifier**, **identifier**, ..., **identifier**] .

features ⟶ `Features:` [**identifier**, **identifier**, ..., **identifier**] .

cvmatrixdefaults ⟶ `Defaults:` [**cvmatrixdefault**, **cvmatrixdefault**, ..., **cvmatrixdefault**] .

xmatrixdefaults ⟶ `Defaults:` [**xmatrixdefault**, **xmatrixdefault**, ..., **xmatrixdefault**] .

cvtreedefaults ⟶ `Defaults:` [**cvtreedefault**, **cvtreedefault**, ..., **cvtreedefault**] .

xtreedefaults ⟶ `Defaults:` [**xtreedefault**, **xtreedefault**, ..., **xtreedefault**] .

cvmatrixfullspecs ⟶ `FullSpecs:` [**cvmatrixdefault**, **cvmatrixdefault**, ..., **cvmatrixdefault**] .

xmatrixfullspecs ⟶ `FullSpecs:` [**xmatrixdefault**, **xmatrixdefault**, ..., **xmatrixdefault**] .

cvtreefullspecs ⟶ `FullSpecs:` [**cvtreedefault**, **cvtreedefault**, ..., **cvtreedefault**] .

xtreefullspecs ⟶ `FullSpecs:` [**xtreedefault**, **xtreedefault**, ..., **xtreedefault**] .

cvmatrixdefault ⟶ `vowel ->` **matrix** | `consonant ->` **matrix** | **xmatrixdefault**

xmatrixdefault ⟶ **identifier** `->` **matrix** | **matrix** `->` **matrix** | **identifier** `->` **identifier** | **identifier** `->` **identifier matrix** | `any ->` **matrix** | `any ->` **identifier**

cvtreedefault ⟶ `vowel ->` **matrix** | `vowel ->` **segmentspec** | `consonant ->` **matrix** | `consonant ->` **segmentspec** | **xtreedefault**

xtreedefault ⟶ **identifier** `->` **segmentspec** | **identifier** `->` **identifier** | **identifier** `->` **identifier matrix** | `any ->` **segmentspec** | `any ->` **identifier** | `featureless` **identifier** `->` **segmentspec** | `vowel ->` **segmentspec** | `consonant ->` **segmentspec** | **matrix** `->` **segmentspec** | **identifier** `->` **matrix** | `any ->` **matrix** | **matrix** `->` **matrix**

matrix ⟶ [**feature**, **feature**, ... **feature**] segmentspec ⟶ segment{segspec} | segment{segspec identifier} | segment{segspec : segmentspec, segmentspec, ..., segmentspec} | segment{segspec : identifier : segmentspec, segmentspec, ..., segmentspec}

segspec ⟶ identifier | matrix | feature | number | (segspec) | {segspec, segspec, ..., segspec}

tree ⟶ Tree {node, node, ..., node}

node ⟶ {identifier} | {identifier : identifier} | {identifier : identifier : featuredef, featuredef, ..., featuredef}

featuredef ⟶ [identifier]

tonespec ⟶ [ConnectTones] numtones maxspec [tonenames] [tonereps]

numtones ⟶ NumberofTones: (number | 0).

maxspec ⟶ [maxtvspec] [maxvtspec]

maxtvspec ⟶ MaxTonesperVowel: [number | infinite] .

maxvtspec ⟶ MaxVowelsperTone: [number | infinite] .

tonenames ⟶ ToneNames: [identifier, identifier, ..., identifier] .

tonereps ⟶ ToneRepresentations: [tonerep, tonerep, ..., tonerep] .

tonerep ⟶ identifier : identifier / segspec segspec ... segspec | identifier : / segspec

associates ⟶ [Associates: [assoc, assoc, assoc] .]

assoc ⟶ { segmentspec, segmentspec }

definitions ⟶ Definitions: [definition, definition, ..., definition] .

definition ⟶ Define identifier segmentspec | Define identifier segmentspec

rulespec ⟶ Rules: [rule, rule, ..., rule]

rule ⟶ Rule identifier : [RtoL] [NoWordBounds] [NoMorphBounds] [tierspec] [conspec] [effectspec]

tierspec ⟶ Tiers: [tier, tier, ..., tier].

conspec ⟶ Connections: [connection, connection, ..., connection].

effectspec ⟶ Effects: [effect, effect, ..., effect].

tier ⟶ identifier: segspec segspec ... segspec

connection ⟶ segref -- segref

segref ⟶ segspec | segspec[number] | segspec[number identifier]

effect ⟶ segref -Z- segref | segref :: segref | segref ≫ identifier | ≪ segref identifier | segref -> [segref] _ [segref] | segref -> 0 | segref -> segspec | segref ::-> segspec / [segref] _ [segref] | 0 -> segspec / [segref] _ [segref]

Appendix B

Examples Incompletely Specified in the Text

This appendix lists, without explanation, all working examples incompletely specified in the text.

B.1 Mende

Specification File:

```
Language Mende:

Phonemes:  n, a, v, m, b, o.

SpecMethod:  CV.

Vowels:  a, o.
Consonants:  n, v, m, b.

ConnectTones
ToneLevels:  2.

ToneReps:  "á" :  a / 2, "à" :  a / 1, "ǎ" :  a / 1 2, "ó" :  o / 2, "ò" :
   o / 1, "ǒ" :  o / 1 2.

Associates: {segment{T}, segment{V}}, {segment{X}, segment{P}}.

Rules:

Rule "Tone Assimilation":
NoMorphBounds
Tiers:
     tonal: 2    1,
  skeletal: V C0 V.
Connections:
  V[1] -- 2,
  V[2] -- 1.
Effects:
  V[2] :: 2,
  V[2] -Z- 1.
```

```
Rule "Rising to Low":
NoMorphBounds
Tiers:
     tonal: 1    2,
  skeletal: V C0 V.
Connections:
  V[1] -- 1,
  V[1] -- 2,
  V[2] -- 2.
Effects:
  V[1] -Z- 2.
```

Sample Inputs and Outputs:

Input	Output
nàvó+mà	nàvó+má
mbǎ+mà	mbà+má

B.2 Tagalog

Specification File:

```
Language Tagalog:

Phonemes:  p, i, l, n, t, a, h, k, u, m, N, "ŋ", RE.

SpecMethod:  CV/Tree.

Vowels:  a, i, u.
Consonants:  h, k, l, m, n, "ŋ", N, p, t, RE.

Tree {

       {vroot :  skeletal},
       {croot :  skeletal},
       {stricture :  croot :  [lat]},
       {supralaryngeal1 :  croot},
       {supralaryngeal2 :  vroot},
       {softpalate :  supralaryngeal1 :  [nasal]},
       {place1 :  supralaryngeal1},
       {place2 :  supralaryngeal2},
       {labial1 :  place1},
       {labial2 :  place2},
       {coronal :  place1},
       {dorsal2 :  place2 :  [high], [back]},
       {dorsal1 :  place1}

  }

Defaults:
vowel -> segment {vroot : segment{supralaryngeal2 :
          segment{place2}}},

consonant -> segment {croot : segment{stricture : segment{-lat}},
segment{supralaryngeal1}},
```

```
vowel -> segment{place2 :segment{labial2},
 segment{dorsal2 :segment{-high},
  segment{+back}}},

i -> [+high, -back],
u -> [+high],

t -> segment{supralaryngeal1 : segment{place1 : segment{coronal}},
       segment{softpalate : segment{-nasal}}},
l -> t [+lat],
n -> t [+nasal],

RE -> segment{supralaryngeal1 : segment{softpalate : segment{+nasal}}},
RE -> [+lat],
% just to distinguish it from h...

k -> segment{supralaryngeal1 : segment{place1 : segment{dorsal1}},
       segment{softpalate : segment{-nasal}}},

"ŋ" -> k [+nasal],

p -> segment{supralaryngeal1 : segment{place1 : segment{labial1}},
       segment{softpalate : segment{-nasal}}},
m -> p [+nasal],

N -> RE [-lat].

ToneLevels: 0.

Rules:

Rule "Reduplication":
Tiers:
       lat: +lat,
     nasal: +nasal,
     vroot:                         vroot,
  skeletal:    C    "]m" "m["  C     V    C0,
     croot:                   croot.
Connections:
  C[1] -- +nasal,
  C[1] -- +lat,
  C[2] -- croot,
  V -- vroot.
Effects:
  "]m" -> 0,
  "m[" -> 0,
  C[1] -> 0,
  croot ::-> C / V _,
  vroot ::-> V / _ C0.

Rule "Coalescence":
Tiers:
     nasal:+nasal  "]m" "m[",
    place1:        "]m" "m[" place1,
```

```
    skeletal:  (C)   "]m" "m["   C,
        croot:croot  "]m" "m[" croot.
   Connections:
     C[1] -- +nasal,
     C[1] -- croot[1],
     C[2] -- place1,
     C[2] -- croot[2].
  Effects:
    "]m"[1, skeletal] -> 0,
    "m["[1, skeletal] -> 0,
    C[2] :: +nasal,
    C[1] -Z- +nasal,
    C[1] -> 0.

  Rule "Infixation":
  Tiers:
       back: -back        "]m" "m[",
       high: +high        "]m" "m[",
      nasal:       +nasal "]m" "m[",
      vroot: vroot        "]m" "m["      vroot,
   skeletal:   V      C   "]m" "m["   C    V,
      croot:        croot "]m" "m[" croot.
  Connections:
    -back -- V[1],
    +high -- V[1],
    vroot[1] -- V[1],
    croot[1] -- C[1],
    +nasal -- C[1],
    croot[2] -- C[2],
    vroot[2] -- V[2].
  Effects:
    0 -> i / vroot[2] _,
    0 -> n / croot[2] _,
    n ::-> C / C[2] _,
    i ::-> V / C[2] _,
    "]m"[1, skeletal] -> 0,
    "m["[1, skeletal] -> 0,
    V[1] -> 0,
    C[1] -> 0.
```

Sample Inputs:

```
pili       RE+pili     maN+pili      maN+RE+pili     in+pili       in+RE+pili
% 'choose'

tahi       RE+tahi     maN+tahi      maN+RE+tahi     in+tahi       in+RE+tahi
% 'take'

kuha       RE+kuha     maN+kuha      maN+RE+kuha     in+kuha       in+RE+kuha
% 'sew'
```

Corresponding Outputs:

```
pili pipili mamili mamimili pinili pinipili
```

```
tahi tatahi manahi mananahi tinahi tinatahi
kuha kukuha maŋuha maŋuŋuha kinuha kinukuha
```

B.3 Turkish

```
Language Turkish:

Phonemes:  A, I, a, e, i, o, u, "ü", "ö", "ı", p, b, m, f, v, t, d, s, z, n,
   l, r, "ç", c, "ş", j, y, k, g, h.

SpecMethod:  CV/Tree.

Vowels:  A, I, a, e, i, o, u, "ü", "ö", "ı".
Consonants:  p, b, m, f, v, t, d, s, z, n, l, r, "ç", c, "ş", j, y, k, g, h.

Tree {
        {root : skeletal},
        {stricture : root : [cont], [lat], [son], [stri]},
        {laryngeal : root : [voice]},
        {supralaryngeal : root},
        {softpalate : supralaryngeal : [nasal]},
        {place : supralaryngeal},
        {labial : place : [round]},
        {coronal : place : [ant]},
        {dorsal : place : [high], [back]}
}

Defaults:
           any -> segment{root : segment{stricture : segment{+son},
                                                     segment{+cont},
                                                     segment{-stri},
                                                     segment{-lat}},
                                 segment{laryngeal : segment{voice}},
                                 segment{supralaryngeal :
                                   segment{softpalate : segment{-nasal}},
                                   segment{place}}},

           vowel -> segment{place : segment{labial : segment{-round}},
                                    segment{coronal : segment{-ant}},
                                    segment{dorsal : segment{-back},
                                            segment{-high}}},

       a -> [+back],
       A -> [back],
       I -> [+high, back, round],
       i -> [+high],
       "ö" -> [+round],
       o -> a [+round],
       "ü" -> i [+round],
       "ı" -> i [+back],
       u -> "ü" [+back],

       consonant -> [-son, -cont, -voice],
```

```
        p -> segment{place : segment{labial}},
        f -> p [+cont],
        m -> p [+nasal, +son],
        b -> p [+voice],
        v -> b [+cont],

        t -> segment{place : segment{coronal : segment{+ant}}},
        d -> t [+voice],
        "ç" -> t [-ant, +stri],
        c -> d [-ant, +stri],
        n -> t [+nasal, +son],
        z -> d [+cont],
        s -> t [+cont],
        "ş" -> s [-ant],
        l -> d [+lat, +son, +cont],
        r -> d [+son, +cont],

        y -> i,

        k -> segment{place : segment{dorsal}},
        g -> k [+voice],

        [+son] -> [+voice],
        [-voice, +cont] -> [+stri].

ToneLevels: 0.

   Rules:

   Rule "Iyor Deletion":
   Tiers:
         root: A "]m" "m[" I y o r,
     skeletal: V "]m" "m[" V C V C.
   Connections:
     A[1] -- V[1],
     I[2] -- V[2],
     y[3] -- C[1],
     o[4] -- V[3],
     r[5] -- C[2].
   Effects:
     A[1] -> 0,
     V[1] -> 0.

   Rule "High Vowel Deletion":
   Tiers:
         high:     "]m" "m["+high,
     skeletal:  V   "]m" "m["  V.
   Connections:
     V[2] -- +high.
   Effects:
     V[2] -> 0.

   Rule "Back spreading":
   NoMorphBounds
   Tiers:
```

```
      back: @back  back,
  skeletal:   V  C0 V.
Connections:
  V[1] -- @back[1],
  V[2] -- back[2].
Effects:
  V[2] :: @back[1].

Rule "Round spreading":
NoMorphBounds
Tiers:
     round: @round       round,
      high:              +high,
  skeletal:    V    C0     V.
Connections:
  V[1] -- @round[1],
  V[2] -- round[2],
  V[2] -- +high.
Effects:
  V[2] :: @round.

Rule "Morpheme Deletion 1":
Tiers:
  skeletal: "]m".
Effects:
  "]m" -> 0.

Rule "Morpheme Deletion 2":
Tiers:
  skeletal: "m[".
Effects:
  "m[" -> 0.
```

Sample Inputs:

```
% Sing.      Plural.          Genetive 1psg.   Genetive 1ppl.
%-------------------------------------------------------------

diş.........diş+lAr..........diş+Im...........diş+lAr+Im

% 'tooth'    'teeth'          'my tooth'       'my teeth'

ev..........ev+lAr...........ev+Im............ev+lAr+Im

% 'house'    'houses'         'my house'       'houses'

gün.........gün+lAr..........gün+Im...........gün+lAr+Im

% 'day'      'days'           'my day'         'my days'

göz.........göz+lAr..........göz+Im...........göz+lAr+Im

% 'eye'      'eyes'           'my eye'         'my eyes'

baş.........baş+lAr..........baş+Im...........baş+lAr+Im
```

```
% 'head'     'heads'          'my head'        'my heads'

kız.........kız+lAr.........kız+Im..........kız+lAr+Im

% 'girl'     'girls'          'my girl'        'my girls'

kol.........kol+lAr.........kol+Im..........kol+lAr+Im

% 'arm'      'arms'           'my arm'         'my arms'

mum.........mum+lAr.........mum+Im..........mum+lAr+Im

% 'candle'   'candles'        'my candle'      'my candles'

masa+Im

ilaç+lA+Iyor
```

Corresponding Outputs:

```
diş
dişler
dişim
dişlerim
ev
evler
evim
evlerim
gün
günler
günüm
günlerim
göz
gözler
gözüm
gözlerim
baş
başlar
başım
başlarım
kız
kızlar
kızım
kızlarım
kol
kollar
kolum
kollarım
mum
mumlar
mumum
mumlarım
masam
ilaçlıyor
```

Appendix C

Selected Code

The following represents the subset of AMAR code most clearly implementing autosegmental phonology. All procedures explicitly referred to in the text may be found here.

C.1 Objects

```
/* TONES.H */

#ifndef __tones
#define __tones 1
#include <fstream.h>
#include <libc.h>
#include "strings.h"
#include "Pix.h"
#include <stdio.h>

#define TRUE (1)
#define FALSE (0)
#define STE StrTableEntry

class Rule;
class Tier;
class Chart;
class WordBoundary;
class SegmentSet;
class FeatureMatrix;
class X;
class SegList;
class RuleList;
class CSMap;
class ConnectableSegment;
class Segment;
extern class StrTableEntry;
extern class StrStack;
extern class Map;

extern void make_tier(char*);
extern StrTableEntry* create_topmost_classnode(StrTableEntry*);
extern StrTableEntry* create_empty_classnode(StrTableEntry*, StrTableEntry*);
extern STE* create_filled_classnode(STE*, STE*, STE*);
extern void enter_free_associates(STE*, STE*);
extern void remove_free_associates(STE*, STE*);
```

```
extern void create_rule(STE*);
extern void sort_tiers();
extern void make_rule_tier(STE*);
extern void enter_seg_in_tier(STE*);
extern main(int, char**);
extern void check_sandhi();
extern void maybe_add_tonal_and_phonemic_tiers();
extern void enter_tone_rep(STE*, STE*, STE*);
extern StrTableEntry* next_vowel(Pix&);
extern StrTableEntry* next_consonant(Pix&);
extern void xify_phonemes();
extern void fill_empty_classnode(StrTableEntry*, StrTableEntry*);
extern void enter_seg_in_tier(StrTableEntry*);
extern void break_connection(STE*, STE*);
extern Segment* fewest_inferiors(SegList*);
extern int apply(Rule&, Chart&);
extern int ste_matches(STE*, X*);
extern void error(const char* s1, const char* s2 = "");
extern ConnectableSegment* convert(SegList*, Pix);
extern Pix in_map(Segment*, Map*, int, int&);
extern void spread_right(StrTableEntry* segref, StrTableEntry* tr);
extern void spread_left(StrTableEntry* segref, StrTableEntry* tr);
extern void duplicate_connections(Segment*, Segment*, Segment*, Segment*);

int wordend;
int morphend;
int demo;
int ks;

class Rule {
  Tier **original, **replacement;
  int num_tiers;
  int no_worddivs;
  int no_morphdivs;
  int rtol;
  void init() { name = 0; no_morphdivs=no_worddivs=is_sandhi=rtol = FALSE; }
  int sz;
public:
  Rule(int size);
  ~Rule() { free((char *)original); free((char *)replacement); }
  inline void operator delete(void* vd);
  inline void* operator new(size_t);

  char* name;
  int is_sandhi;

  inline void set_name(char* nm);
  void set_no_worddivs() { no_worddivs = TRUE; is_sandhi = TRUE; }
  void set_no_morphdivs() { no_morphdivs = TRUE; }
  void set_rtol() { rtol = TRUE; }
  int sandhi();
  int rtol_sandhi();

  void application(Chart&, Pix*, int*);
  int adjust_connections(ConnectableSegment*, ConnectableSegment*,
                         Map*, Pix, Chart&, int, int*);
  Pix match(Tier&, Tier**, int, int);
  int unconnected(int);
  void connect_map(Map* map);
```

```
  void find_closest_usable_rule_segment(int);
  void sort_tiers();

  friend int apply(Rule& rule, Chart& chart);
  friend void create_rule(STE*);
  friend void sort_tiers();
  friend void make_rule_tier(STE*);
  friend void enter_seg_in_tier(STE*);
  friend void break_connection(StrTableEntry*, StrTableEntry*);
  friend void spread_right(StrTableEntry* segref, StrTableEntry* tr);
  friend void spread_left(StrTableEntry* segref, StrTableEntry* tr);

  void make_connection(STE*, STE*);
  void join(STE*, STE*);
  void metathesize(STE*, STE*, STE*);
  void metathesize_after(STE*, STE*);
  void metathesize_before(STE*, STE*);
  void replace(STE*, STE*);
  void del(STE*);
  void insert_joined_before(STE*, STE*, STE*);
  void insert_joined_after(STE*, STE*, STE*);
  void insert_before(STE*, STE*);
  void insert_after(STE*, STE*);

  void print(int);
};

class RuleLink {
  friend class RuleList;
private:
  Rule* rule;

  RuleLink* pre;
  RuleLink* suc;

  RuleLink(Rule& r) { rule = new Rule(r); }
  ~RuleLink() { if (suc) suc->protect(); delete suc; } // delete links rcsively

  void protect() { rule = NULL; }
  inline void operator delete(void* vd);
  inline void* operator new(size_t);
};

class RuleList {
  RuleLink* head;
  RuleLink* tail;
  int sz;

  void init() { sz = 0; head = 0; tail = 0;}
public:
  RuleList() { init(); }
  RuleList(const RuleList&);
  ~RuleList() { if (head) head->protect(); delete head; }

  void operator delete(void* vd) { ::free((char *)vd); }
  inline void* operator new(size_t);

  RuleList& operator= (const RuleList&);
```

```
  Pix first() const { return (Pix)(head); }
  void next(Pix& p) const { p = (p == 0) ? 0 : (Pix)(((RuleLink *)p)->suc); }
  void prev(Pix& p) const { p = (p == 0) ? 0 : (Pix)(((RuleLink *)p)->pre); }
  Rule& operator[](const Pix p) const { return *((RuleLink *)p)->rule; }
  int length() const { return sz; }
  int empty() const { return (head == 0); }
  void prepend(Rule&);
  void append(Rule&);
  void del(Pix&);
  Pix ins_before(Pix, Rule&);
  Pix ins_after(Pix, Rule&);
};

// *************************************************
// Map for dealing with Freely Associating Segments
// *************************************************

class CSMap;

class CSLink {
  friend class CSMap;
private:
  ConnectableSegment* key;
  ConnectableSegment* value;

  CSLink* pre;
  CSLink* suc;

  CSLink(ConnectableSegment* k, ConnectableSegment* v) { key = k; value = v; }
  ~CSLink() { delete suc; } // delete all links recursively
  void operator delete(void* vd) { ::free((char *)vd); }
  void protect() { key = NULL; value = NULL; }
  inline void* operator new(size_t);
};

class CSMap {
  CSLink* head;
  CSLink* tail;
  int sz;

  void init() { sz = 0; head = 0; tail = 0; }
public:
  CSMap() { init(); }
  CSMap(const CSMap&);
  ~CSMap() { delete head; } // delete all links recursively
  void operator delete(void* vd) { ::free((char *)vd); }
  inline void* operator new(size_t);

  CSMap& operator= (const CSMap&);

  void enter(ConnectableSegment* k, ConnectableSegment* v);
  void remove(ConnectableSegment* k, ConnectableSegment* v);
  void enter_aux(ConnectableSegment* k, ConnectableSegment* v);
  void remove_aux(ConnectableSegment* k, ConnectableSegment* v);
  SegmentSet* associates(ConnectableSegment* k);
  int freely_assoc(Segment* k, Segment* v);

  int size() const { return sz; }
};
```

```
/***********/
/* End Map */
/***********/

class Chart {
  Tier **tier; // Initially null, then an array of tiers.
  int num_tiers;
  RuleList rules;
  CSMap free_associates;
  ConnectableSegment *tree;
  int sz;
  inline void init();
public:
  Chart() { init(); }
  ~Chart() { }
  void operator delete(void* vd) { ::free((char *)vd); }
  inline void* operator new(size_t);

  int max_tones_per_vowel;
  int max_vowels_per_tone;
  int sandhi_rules_exist;
  int rtol_sandhi_rules_exist;
  int num_tones;
  int no_connect;
  char* name;

  inline void set_name(char* nm);
  void apply_assoc_convention(Pix, Tier&, Tier&);
  void assoc_convention(int* tier_index, int num_tiers);

  Tier& operator[](const int i) const { return *tier[i]; }
  int empty();
  Tier& skeletal() { return *tier[0]; }

  void add_tier(Tier& tier);
  void add_skeletal_seg(ConnectableSegment*);
  void read_word(istream&, StrStack*);
  void print_and_delete_word();
  void print_and_delete_phrase();
  void main_loop(istream&);
  ConnectableSegment* tier_in_tree(Tier&);

  inline int freely_associate(Segment* s1, Segment* s2);
  int is_tier_superior(Tier&, Tier&);
  int is_superior(ConnectableSegment*, ConnectableSegment*);

  friend int apply(Rule& rule, Chart& chart);
  friend StrTableEntry* create_topmost_classnode(StrTableEntry*);
  friend STE* create_empty_classnode(STE*, STE*);
  friend STE* create_filled_classnode(STE*, STE*, STE*);
  friend void enter_free_associates(STE*, STE*);
  friend void remove_free_associates(STE*, STE*);
  friend main(int, char**);
  friend int ste_matches(STE*, X*);
  friend void check_sandhi();
  friend void maybe_add_tonal_and_phonemic_tiers();
  friend void enter_tone_rep(STE*, STE*, STE*);
};
```

```
class Segment {
  static int class_unique_num;
  int id_num;
  void init() { typ = S; id_num = class_unique_num++; tier = 0; modified = 0; }
public:
  enum { S=0, CS=1, SS=2, XX=3, WE=4, WB=5, MB=6, ME=7, FM=8, C=9, V=10,
         GT=11, TN=12, GP=13, P=14, C0=15, V0=16, X0=17, CN=18, F=19 } typ;
  Tier *tier;

  int modified;

  Segment() { init(); }
  Segment(Segment* seg) { init(); tier = seg->tier; }
  virtual ~Segment() { }

  inline void operator delete(void* vd);
  inline void* operator new(size_t);

  // Equality Tests:
  virtual Pix matches(Pix, Tier&, Tier**, int) { return (Pix)(-1); }
  int operator==(Segment& segment) { return (segment.id_num == id_num); }
  int operator!=(Segment& segment) { return (segment.id_num != id_num); }
  virtual int is_zero() { return FALSE; } // Is a C_0 type thing.
  virtual Pix zeromatches(Pix, Tier&, Tier&) { abort(); return NULL; }

  int is_in_tier(Tier&);
  int is_actually_in_tier(Tier&);

  // Connection related
  virtual int is_connectable() { return FALSE; }
  virtual int inert() { return TRUE; }
  virtual int connects_directly_to_tier(Tier&) { return FALSE; }
  virtual int connects_directly_to(Segment*) { return FALSE; }
  virtual int connects_to_tier(Tier&) { return FALSE; }
  virtual int unconnected(int*, int) { return TRUE; }
  virtual void detach() { }
  virtual void safe_detach() { }
  int connect(Segment*) { abort(); }

  // Spreading:
  virtual int spreads() { return FALSE; }

  // Copying:
  virtual Segment* copy() { return (new Segment(this)); }
  virtual Segment* surface_copy() { return (new Segment(this)); }
  virtual void csub(Segment* seg) { seg->tier = tier; }
  void make_identical_to(Segment* seg) { id_num = seg->id_num; }

  friend void enter_seg_in_tier(StrTableEntry*);
  friend main(int, char**);

  virtual void print(int pos = 0) { }
  void print_id() { cerr << id_num; }
  virtual void print_aux(int pos = 0) { }

  virtual int type_eq(Segment*) { return FALSE; }
  virtual int eqv(Segment*) { return FALSE; }
```

```
  int is_a_morphemeboundary() { return (typ == MB || typ == ME); }
  int is_a_wordboundary() { return (typ == WB || typ == WE); }
  int is_a_boundary() { return(is_a_wordboundary()||is_a_morphemeboundary()); }
  int is_end() { return (typ == ME || typ == WE); }
  int is_begin() { return (typ == MB || typ == WB); }
};

class ConnectableSegment : public Segment {
  ConnectableSegment **inferior;
  int infsz; // Size of **inferior - starts out as five.
  int num_inferiors;
  ConnectableSegment **superior;
  int supsz; // Size of **superior - starts out as 1.
  int num_superiors;
  int spreads_left;
  int spreads_right;
  int *inferior_to_spread_along; // first el is the number of items.
  int *superior_to_spread_along; // first el is the number of items.
  inline void init(int inf, int sup);
public:
  ConnectableSegment() { init(5,1); }
  ConnectableSegment(const ConnectableSegment& cs) {
    init(cs.infsz, cs.supsz);
    num_inferiors = cs.num_inferiors;
    num_superiors = cs.num_superiors;
  }
  inline virtual ~ConnectableSegment();

  int is_exact; // Only applies to segments in rules -- if true, the segment
                // needs to be matched exactly.
  int is_inert; // If true, the segment is ignored by the Association Conv.
  int inert() { return is_inert; }

  // Equality Tests:
  Pix matches(Pix, Tier&, Tier**, int);
  int eq(ConnectableSegment*, Tier**, int);
  int eq(Segment*, Tier**, int) { return FALSE; }
  int equal(ConnectableSegment*, Tier**, int);
  int delete_best_match(SegList*, Tier**, int);
  virtual int type_eq(Segment* s);
  virtual int eqv(Segment* s);

  // Copying
  void copy_aux(ConnectableSegment*, Map*, int, int*, Chart&, Tier**);
  virtual Segment* copy();
  virtual Segment* surface_copy();
  void copy_sub(ConnectableSegment* seg);
  void csub(Segment* seg);

  // Relate to connections:
  SegList *topmost_superiors();
  int is_connectable() { return TRUE; }
  int connects_directly_to_tier(Tier&);
  int connects_to_tier(Tier&);
  int unconnected(Tier**,int);
  int connect(ConnectableSegment* cs);
  void disconnect(ConnectableSegment* cs); // if seg & cs are def. connected.
  void safe_detach(); // use to completely disconnect a seg, but leave usable.
  void detach(); // use before destroying a segment.
```

```
    void delete_fully(Map*, int, Chart&, int*);
    void break_connection(ConnectableSegment*); // call this one.
    SegList *connects_to(ConnectableSegment*); // Note - tests by eqv.
    int connects_directly_to(Segment* seg);
    int not_too_many_vowels();
    int not_too_many_tones();

    friend void
      Rule::adjust_connections(ConnectableSegment*, ConnectableSegment*,
                                     Map*, Pix, Chart&, int, int*);
    void no_duplicate_features(ConnectableSegment*, Map*, int, Chart&, int*);
    friend void Tier::metathesize(Pix&, Pix, Pix, int, Tier**, Map*);

    SegList* inferiors();
    SegList* superiors();
    SegList* direct_superiors();
    void sort();

    Tier **tiers_in(int& num_tiers);
    Tier **add_tiers(Tier** tlist, int& num_tiers, int& sz);

    // Relates to spreading:
    int spreads() { return (spreads_left || spreads_right); }
    void spread(Pix, Tier&, Chart&);

    friend void Chart::apply_assoc_convention(Pix, Tier&, Tier&);
    friend Segment* fewest_inferiors(SegList*);

    // Parsing friends!
    friend StrTableEntry* next_vowel(Pix&);
    friend StrTableEntry* next_consonant(Pix&);
    friend void fill_empty_classnode(StrTableEntry*, StrTableEntry*);
    friend void break_connection(StrTableEntry*, StrTableEntry*);
    friend void Rule::replace(StrTableEntry*, StrTableEntry*);
    friend void enter_tone_rep(STE*, STE*, STE*);
    friend void Chart::read_word(istream& infile, StrStack*);
    friend int ste_matches(STE*, X*);
    friend void Rule::connect_map(Map* map);
    friend void duplicate_connections(Segment*, Segment*, Segment*, Segment*);
    friend void xify_phonemes();

    int make_spread(Tier*);
    void set_right_spread();
    void set_left_spread();

    virtual void print(int pos = 0) { };
    void print_aux(int);
};

class SegList;

class SegLink {
    friend class SegList;
private:
    Segment* seg;

    SegLink* pre;
    SegLink* suc;
```

```
  SegLink(Segment* s) { seg = s; }
  ~SegLink() { if (suc) suc→protect(); delete suc; } // delete links rcsively

  void protect() { seg = NULL; }
  inline void operator delete(void *vd);
  inline void* operator new(size_t);
};

class SegList {
  SegLink* head;
  SegLink* tail;
  int sz;

  void init() { sz = 0; head = 0; tail = 0;}
public:
  SegList() { init(); }
  ~SegList() { if (head) head→protect(); delete head; }

  SegList(const SegList&);
  SegList& operator= (const SegList&);

  void operator delete(void* vd) { ::free((char *)vd); }
  inline void* operator new(size_t);

  Pix first() const { return (Pix)(head); }
  Pix last() const { return (Pix)(tail); }
  void next(Pix& p) const { p = (p == 0) ? 0 : (Pix)(((SegLink *)p)→suc); }
  void prev(Pix& p) const { p = (p == 0) ? 0 : (Pix)(((SegLink *)p)→pre); }
  inline Segment* operator[](const Pix p) const;
  void insert_at(const Pix p, Segment* seg) { ((SegLink *)p)→seg = seg; }
  int length() const { return sz; }
  int empty() const { return (head == 0); }
  void prepend(Segment*);
  Pix append(Segment*);
  void join(SegList* sl);
  void del(Pix&);
  Pix ins_before(Pix, Segment*);
  Pix ins_after(Pix, Segment*);
};

class Tier {
  SegList segments;
  static int class_unique_num;
  int id_num;
  void init() { name = NULL; current = NULL; id_num = class_unique_num++; }
public:
  Tier() { init(); }
  Tier(char* nm) { init(); set_name(nm); }
  Tier(const Tier& t);
  ~Tier() { }

  void operator delete(void* vd) { ::free((char *)vd); }
  inline void* operator new(size_t);

  char* name;
  Pix current;

  inline void set_name(char* nm);
  void make_identical_to(Tier *tr) { id_num = tr→id_num; }
```

```
  Tier& operator= (const Tier& t) { segments = t.segments; return *this; }

  int operator==(const Tier& tier) const { return (tier.id_num == id_num); }
  int operator≠(const Tier& tier) const { return (tier.id_num ≠ id_num); }
  int name_eq(const Tier& tier) const { return (tier.id_num == id_num); }

  Pix first() const { return (segments.first()); }
  Pix last() const { return (segments.last()); }
  void next(Pix& p) const { segments.next(p); }
  void prev(Pix& p) const { segments.prev(p); }
  void del(Pix& loc);
  Pix insert(Pix loc, Segment* seg);
  Pix ins_after(Pix loc, Segment* seg);
  Pix append(Segment* seg);
  void prepend(Segment* seg) { seg→tier = this; segments.prepend(seg); }
  Segment* operator[] (Pix p) const { return segments[p]; }
  void insert_at(const Pix p, Segment* seg);
  int length() const { return (segments.length()); }

  int is_applicable(Tier**, int);
  Pix preceding(Segment* s1, Segment* s2);
  int precedes(Segment* s1, Segment* s2);
  Pix first_to_tier(Tier&, Pix, Pix, Tier&, ConnectableSegment*, Tier**,
                    int, int, int);
  void metathesize(Pix&, Pix, Pix, int, Tier**, Map*);
  Pix find(Segment*);

  friend void make_tier(char*);
  friend void enter_seg_in_tier(STE*);
  friend int Segment::is_in_tier(Tier&);
  friend main(int, char**);
};

class SegmentSet : public ConnectableSegment {
  SegList segments;
public:
  SegmentSet() { typ = SS; }
  SegmentSet(const SegmentSet& ss) : segments(ss.segments) { typ = SS; }
  ~SegmentSet() { }
  void operator delete(void *vd);

  void insert(Segment*);
  void remove(Segment*);
  Pix first() { return segments.first(); }
  void next(Pix& i) { segments.next(i); }
  int empty() { return segments.empty(); }
  int length() { return segments.length(); }
  Segment* operator[] (Pix p) { return segments[p]; }

  Segment* copy();
  Segment* surface_copy();

  int eqv(Segment* s);
  int type_eq(Segment* s) { return (s→typ == SS); }
};

class X : public ConnectableSegment {
public:
  X() { typ = XX; }
```

```
  X(const X&) { typ = XX; }

  virtual int eqv(Segment* s);
  virtual int type_eq(Segment* s);

  virtual Segment* copy() { X *x = new X; copy_sub(x); return x; }
  virtual Segment* surface_copy() { X* x = new X; csub(x); return x; }
  void print(int);
};

class Consonant: public X {
public:
  Consonant() { typ = C; }
  Consonant(const Consonant&) { typ = C; }
  int eqv(Segment* p) { return (p->typ == C); }
  int type_eq(Segment* p) { return (p->typ == C); }
  Segment* copy() { Consonant *c = new Consonant; copy_sub(c); return c; }
  Segment* surface_copy() { Consonant *c = new Consonant; csub(c); return c; }
};

class Vowel: public X {
public:
  Vowel() { typ = V; }
  Vowel(const Vowel&) { typ = V; }
  int eqv(Segment* p) { return (p->typ == V); }
  int type_eq(Segment* p) { return (p->typ == V); }
  Segment* copy() { Vowel *v = new Vowel; copy_sub(v); return v; }
  Segment* surface_copy() { Vowel *v = new Vowel; csub(v); return v; }
};

class GenericTone : public ConnectableSegment {
public:
  GenericTone() { typ = GT; }
  GenericTone(const GenericTone&) { typ = GT; }
  virtual ~GenericTone() { }

  virtual int eqv(Segment* p) { return (p->typ == GT || p->typ == TN); }
  virtual int type_eq(Segment* p) { return (p->typ == GT || p->typ == TN); }

  virtual Segment* copy();
  virtual Segment* surface_copy();
  virtual void print(int pos =0) { }
};

class Tone: public GenericTone {
public:
  Tone(int l) : level(l) { typ = TN; }
  Tone(const Tone& t) : level(t.level) { typ = TN; }
  virtual ~Tone() { }

  int level;

  int eqv(Segment* t) { return (t->typ == TN && ((Tone *)t)->level == level); }
  int type_eq(Segment* p) { return (p->typ == TN); }
  Segment* copy() { Tone *t = new Tone(level); copy_sub(t); return t; }
  Segment* surface_copy() { Tone *t = new Tone(level); csub(t); return t; }
  void print(int pos =0) { cerr << level << "\n"; print_aux(pos); }
};
```

```
class GenericPhoneme : public ConnectableSegment {
public:
   GenericPhoneme() { typ = GP; }
   virtual ~GenericPhoneme() { typ = GP; }

   virtual int eqv(Segment* gp);
   virtual int type_eq(Segment* gp);
   virtual Segment* copy();
   virtual Segment* surface_copy();
   virtual void print(int pos=0) { }
};

class Phonemic : public GenericPhoneme {
   char *representation;
   void init() { representation = NULL; typ = P; }
public:
   Phonemic() { init(); }
   Phonemic(char *c) { init(); set_rep(c); }
   Phonemic(const Phonemic& p) { init(); set_rep(p.representation); }
   virtual ~Phonemic() { }

   inline void set_rep(char* rep);
   int eqv(Segment* p);
   int type_eq(Segment* p) { return (p->typ == P); }
   virtual Segment* copy();
   virtual Segment* surface_copy();
   void print(int pos=0) { cerr << representation << "\n"; print_aux(pos); }
};

class WordBegin : public Segment { // w[
public:
   WordBegin() { typ = WB; }
   WordBegin(const WordBegin& wb) { tier = wb.tier; typ = WB; }
   Pix matches(Pix seg, Tier& tier, Tier **applicable_tier, int n);
   int type_eq(Segment* wb) { return (wb->typ == WB); }
   Segment* copy() { return (new WordBegin(*this)); }
   Segment* surface_copy() { return (new WordBegin(*this)); }
   void print(int pos=0) { if (wordend) cout << " "; wordend = FALSE; }
};

class WordEnd : public Segment { // ]w
public:
   WordEnd() { typ = WE; }
   WordEnd(const WordEnd& we) { tier = we.tier; typ = WE; }
   Pix matches(Pix seg, Tier& tier, Tier **applicable_tier, int n);
   int type_eq(Segment* we) { return (we->typ == WE); }
   Segment* copy() { return (new WordEnd(*this)); }
   Segment* surface_copy() { return (new WordEnd(*this)); }
   void print(int pos=0) { wordend = TRUE; morphend = FALSE; }
};

class MorphemeBegin : public Segment { // m[
public:
   MorphemeBegin() { typ = MB; }
   MorphemeBegin(const MorphemeBegin& mb) { tier = mb.tier; typ = MB; }
   Pix matches(Pix seg, Tier& tier, Tier **applicable_tier, int n);
   int type_eq(Segment* mb) { return (mb->typ == MB); }
   Segment* copy() { return (new MorphemeBegin(*this)); }
   Segment* surface_copy() { return (new MorphemeBegin(*this)); }
```

```
  void print(int pos=0) { if (morphend) cout << "+"; morphend = FALSE; }
};

class MorphemeEnd : public Segment { // ]m
public:
  MorphemeEnd() { typ = ME; }
  MorphemeEnd(const MorphemeEnd& me) { tier = me.tier; typ = ME; }
  Pix matches(Pix seg, Tier& tier, Tier **applicable_tier, int n);
  int type_eq(Segment* s) { return (s->typ == ME); }
  Segment* copy() { return (new MorphemeEnd(*this)); }
  Segment* surface_copy() { return (new MorphemeEnd(*this)); }
  void print(int pos=0) { morphend = TRUE; }
};

class C_0 : public Segment {
  Consonant cons;
public:
  C_0() { typ = C0; }
  C_0(const C_0& c0) : cons(c0.cons) { tier = c0.tier; typ = C0; }
  int is_zero() { return TRUE; }
  Pix matches(Pix, Tier&, Tier**, int) { abort(); return NULL; }
  Pix zeromatches(Pix, Tier&, Tier&);
  Segment* copy() { return (new C_0(*this)); }
  Segment* surface_copy() { return (new C_0(*this)); }
  int type_eq(Segment* s) { return (s->typ == C0); }
};

class V_0 : public Segment {
  Vowel vow;
public:
  V_0() { typ = V0; }
  V_0(const V_0& v0) : vow(v0.vow) { tier = v0.tier; typ = V0; }
  int is_zero() { return TRUE; }
  Pix matches(Pix, Tier&, Tier**, int) { abort(); return NULL; }
  Pix zeromatches(Pix, Tier&, Tier&);
  Segment *copy() { return (new V_0(*this)); }
  Segment *surface_copy() { return (new V_0(*this)); }
  int type_eq(Segment* s) { return (s->typ == V0); }
};

class X_0 : public Segment {
  X x;
public:
  X_0() { typ = X0; }
  X_0(const X_0& x0) : x(x0.x) { tier = x0.tier; typ = X0; }
  int is_zero() { return TRUE; }
  Pix matches(Pix, Tier&, Tier**, int) { abort(); return NULL; }
  Pix zeromatches(Pix, Tier&, Tier&);
  Segment* copy() { return (new X_0(*this)); }
  Segment* surface_copy() { return (new X_0(*this)); }
  int type_eq(Segment* s) { return (s->typ == X0); }
};

// The program should replace occurances of C_+, etc. with C C_0, etc.

/****************************************/
/* */
/* Features! */
/* */
```

```
/****************************************/

class ClassNode : public GenericPhoneme {
    char* name;
    void init() { name = NULL; typ = CN; major_articulator = NULL; }
    ClassNode* major_articulator; // Major articulator for ks mode.
public:
    ClassNode() { init(); }
    ClassNode(const ClassNode& cn) { init(); set_name(cn.name); tier = cn.tier; }
    ClassNode(char* n) { init(); set_name(n); }
    ~ClassNode() { }

    inline void set_name(char* nm);
    int eqv(Segment* cn);
    int type_eq(Segment* cn) { return (cn->typ == CN); }
    Segment* copy();
    Segment* surface_copy();
    void print(int);
};

class Feature;

class FeatureMatrix : public GenericPhoneme {
    Feature **feature;
    int num_features;
    int sz;
public:
    FeatureMatrix();
    FeatureMatrix(const FeatureMatrix& fm);
    ~FeatureMatrix() { free((char *)feature); }
    inline void operator delete(void* vd);

    void add_feature(Feature* f);
    int eqv(Segment* s);
    int type_eq(Segment* fm) { return (fm->typ == FM); }
    Segment* copy();
    Segment* surface_copy();

    void add_features_to_tree_seg(Segment*);
    void copy_features(FeatureMatrix*);
    friend STE* create_filled_classnode(STE*, STE*, STE*);
    void print(int);
};

class Feature : public GenericPhoneme {
    friend class FeatureMatrix;
    char* name;
    void init() { name = NULL; typ = F; }
    int value; // -1 = -, 0 = unspecified, +1 = +, 2 = alpha
public:
    Feature(char* nm, int val=0) { init(); set_name(nm); value = val; }
    Feature(const Feature& f);
    virtual ~Feature() { }

    inline void set_name(char* nm);
    int eqv(Segment* f);
    int type_eq(Segment* f) { return (f->typ == F); }
    int name_eq(Feature* f) { return (strcmp(name, f->name) == 0); }
    Segment* copy();
```

```
   Segment* surface_copy();
   void print(int);
};

// Inlines:

// Delete operators:

inline void Segment::operator delete(void* vd) {
   Segment *s = (Segment *)vd;
   s->tier = 0;
   ::free((char *)vd);
}

inline void SegLink::operator delete(void *vd) {
   SegLink *s = (SegLink *)vd;
   s->seg = 0;
   ::free((char *)vd);
}

inline void RuleLink::operator delete(void* vd) {
   RuleLink *r = (RuleLink *)vd;
   r->rule = 0;
   ::free((char *)vd);
}

inline void FeatureMatrix::operator delete(void* vd) {
   FeatureMatrix *fm = (FeatureMatrix *)vd;
   ::free((char *)fm->feature);
   ::free((char *)vd);
}

// new operators:

inline void* Rule::operator new(size_t size) {
   void* ptr = ::malloc(size);
   return ptr;
}

inline void* RuleLink::operator new(size_t size) {
   void* ptr = ::malloc(size);
   return ptr;
}

inline void* RuleList::operator new(size_t size) {
   void* ptr = ::malloc(size);
   return ptr;
}

inline void* CSLink::operator new(size_t size) {
   void* ptr = ::malloc(size);
   return ptr;
}

inline void* CSMap::operator new(size_t size) {
   void* ptr = ::malloc(size);
   return ptr;
}
```

```
inline void* Chart::operator new(size_t size) {
  void* ptr = ::malloc(size);
  return ptr;
}

inline void* Segment::operator new(size_t size) {
  void* ptr = ::malloc(size);
  return ptr;
}

inline void* SegLink::operator new(size_t size) {
  void* ptr = ::malloc(size);
  return ptr;
}

inline void* SegList::operator new(size_t size) {
  void* ptr = ::malloc(size);
  return ptr;
}

inline void* Tier::operator new(size_t size) {
  void* ptr = ::malloc(size);
  return ptr;
}

// Constructors:

inline Rule::Rule(int size=10)
{
  init();
  sz = size;
  original = (Tier **)malloc(sz * sizeof(Tier *));
  replacement = (Tier **)malloc(sz * sizeof(Tier *));
  num_tiers = 0;
}

inline Tier::Tier(const Tier& t) : segments(t.segments) {
  init(); set_name(t.name); id_num = class_unique_num++;
}

inline Feature::Feature(const Feature& f) {
  init(); set_name(f.name); value = f.value; tier = f.tier;
}

inline FeatureMatrix::FeatureMatrix() {
  sz = 10; num_features = 0;
  feature = (Feature **)malloc(sz*sizeof(Feature *)); typ = FM;
}

inline FeatureMatrix::FeatureMatrix(const FeatureMatrix& fm) {
  int sz = fm.num_features;
  feature = (Feature **)malloc(sz*sizeof(Feature *));
  for (int i = 0; i<sz; i++)
    feature[i] = fm.feature[i];
  num_features = sz;
  typ = FM;
}

// destructors
```

```
inline ConnectableSegment::~ConnectableSegment() {
  free((char *)inferior);
  free((char *)superior);
  free((char *)inferior_to_spread_along);
  free((char *)superior_to_spread_along);
}

// Set name

inline void Tier::set_name(char* nm) {
  name = nm;
}

inline void Rule::set_name(char* nm) {
  name = nm;
}

inline void Chart::set_name(char* nm) {
  name = nm;
}

inline void ClassNode::set_name(char* nm) {
  name = nm;
}

inline void Feature::set_name(char* nm) {
  name = nm;
}

// copy

inline Segment* ConnectableSegment::copy() {
  ConnectableSegment *copy = new ConnectableSegment;
  copy_sub(copy);
  return copy;
}

inline Segment* ConnectableSegment::surface_copy() {
  ConnectableSegment *copy = new ConnectableSegment;
  csub(copy);
  return copy;
}

inline Segment* SegmentSet::copy() {
  SegmentSet *ss = new SegmentSet;
  for (Pix p = first(); p ≠ NULL; next(p))
    ss→insert(operator[](p));
  copy_sub(ss);
  return ss;
}

inline Segment* SegmentSet::surface_copy() {
  SegmentSet *ss = new SegmentSet;
  for (Pix p = first(); p ≠ NULL; next(p))
    ss→insert(operator[](p));
  csub(ss);
  return ss;
}
```

```
inline Segment* GenericTone::copy() {
  GenericTone *gt = new GenericTone; copy_sub(gt); return gt;
}

inline Segment* GenericPhoneme::copy() {
  GenericPhoneme *gp = new GenericPhoneme; copy_sub(gp); return gp;
}

inline Segment* Phonemic::copy() {
  Phonemic *p = new Phonemic(representation); copy_sub(p); return p;
}

inline Segment* ClassNode::copy() {
  ClassNode *c = new ClassNode(name); copy_sub(c); return c;
}

inline Segment* Feature::copy() {
  Feature *f = new Feature(name, value); copy_sub(f); return f;
}

inline Segment* GenericTone::surface_copy() {
  GenericTone *gt = new GenericTone; csub(gt); return gt;
}

inline Segment* GenericPhoneme::surface_copy() {
  GenericPhoneme *gp = new GenericPhoneme; csub(gp); return gp;
}

inline Segment* Phonemic::surface_copy() {
  Phonemic *p = new Phonemic(representation); csub(p); return p;
}

inline Segment* ClassNode::surface_copy() {
  ClassNode *c = new ClassNode(name); csub(c); return c;
}

inline Segment* Feature::surface_copy() {
  Feature *f = new Feature(name, value); csub(f); return f;
}

// eqv

inline int ConnectableSegment::eqv(Segment* s) {
  return (s→typ == CS || s→typ == C || s→typ == V || s→typ == SS ||
          s→typ == GT || s→typ == TN || s→typ == GP || s→typ == P ||
          s→typ == CN || s→typ == FM || s→typ == F || s→typ == XX);
}

inline int X::eqv(Segment* s) {
  return (s→typ == XX || s→typ == C || s→typ == V);
}

inline int GenericPhoneme::eqv(Segment* gp) {
  return (gp→typ == GP || gp→typ == P || gp→typ == CN || gp→typ == F ||
         gp→typ == FM);
}

inline int Phonemic::eqv(Segment* p) {
```

```
  return (p→typ == P &&
          !strcmp(((Phonemic *)p)→representation,representation));
}

inline int ClassNode::eqv(Segment* cn) {
  return (cn→typ == CN &&
          (strcmp(name, ((ClassNode *)cn)→name) == 0) &&
          ((!major_articulator && !((ClassNode *)cn)→major_articulator) ||
           (major_articulator && ((ClassNode *)cn)→major_articulator &&
            major_articulator→eqv(((ClassNode *)cn)→major_articulator))));
}

// type_eq

inline int ConnectableSegment::type_eq(Segment* s) {
  return (s→typ == CS || s→typ == C || s→typ == V || s→typ == SS ||
           s→typ == GT || s→typ == TN || s→typ == GP || s→typ == P ||
           s→typ == CN || s→typ == FM || s→typ == F || s→typ == XX);
}

inline int X::type_eq(Segment* s) {
 return (s→typ == XX || s→typ == C || s→typ == V);
}

inline int GenericPhoneme::type_eq(Segment* gp) {
  return (gp→typ == GP || gp→typ == P || gp→typ == CN || gp→typ == F ||
          gp→typ == FM);
}

// matches

inline Pix
WordBegin::matches(Pix seg, Tier& tier, Tier**, int) {
  Pix currpos = seg;
  tier.next(currpos);
  return ((type_eq(tier[seg])) ? currpos : (Pix)(-1));
}

inline Pix
WordEnd::matches(Pix seg, Tier& tier, Tier **, int) {
  Pix currpos = seg;
  tier.next(currpos);
  return ((type_eq(tier[seg])) ? currpos : (Pix)(-1));
}

inline Pix
MorphemeBegin::matches(Pix seg, Tier& tier, Tier **, int) {
  Pix currpos = seg;
  tier.next(currpos);
  return ((type_eq(tier[seg])) ? currpos : (Pix)(-1));
}

inline Pix
MorphemeEnd::matches(Pix seg, Tier& tier, Tier **, int) {
  Pix currpos = seg;
  tier.next(currpos);
  return ((type_eq(tier[seg])) ? currpos : (Pix)(-1));
}
```

```
// inits

inline void ConnectableSegment::init(int inf, int sup) {
   typ = CS;
   infsz = inf; supsz = sup;
   inferior=(ConnectableSegment **)malloc(infsz*sizeof(ConnectableSegment *));
   superior=(ConnectableSegment **)malloc(supsz*sizeof(ConnectableSegment *));
   inferior_to_spread_along = NULL;
   superior_to_spread_along = NULL;
   num_inferiors = 0;
   num_superiors = 0;
   is_exact = is_inert = FALSE;
   spreads_left = FALSE;
   spreads_right = FALSE;
}

inline void Chart::init() {
   name = NULL;
   sz = 10;
   tier = (Tier **)malloc(sz*sizeof(Tier *));
   num_tiers = 0;
   tree = NULL;
   max_tones_per_vowel = -1;
   max_vowels_per_tone = -1;
   sandhi_rules_exist = FALSE;
   rtol_sandhi_rules_exist = FALSE;
   num_tones = 0;
   no_connect = TRUE;
}

// misc

inline void Phonemic::set_rep(char* rep) {
   representation = rep;
}

inline Segment* SegList::operator[](const Pix p) const {
   SegLink* sl = (SegLink *)p;
   return sl->seg;
}

inline Pix Tier::insert(Pix loc, Segment* seg) {
   seg->tier = this;
   return (segments.ins_before(loc, seg));
}

inline Pix Tier::ins_after(Pix loc, Segment* seg) {
   seg->tier = this;
   return (segments.ins_after(loc, seg));
}

inline Pix Tier::append(Segment* seg) {
   seg->tier = this; segments.append(seg); return segments.last();
}

inline void Tier::insert_at(const Pix p, Segment* seg) {
   seg->tier = this; segments.insert_at(p, seg);
}
```

```
inline int Chart::freely_associate(Segment* s1, Segment* s2) {
  return free_associates.freely_assoc(s1, s2);
}

#endif /* tones */
```

C.2 Application

```
// Apply.cc
#include "Map.h"
extern Chart chart;

void make_set_of_segs_in_tier(Segment* seg, SegmentSet* segs, Tier& intier,
                              ConnectableSegment* rseg, Tier** ap_tier,
                              int num_tiers)
     // Insert into the set segs all segments that are:
     // 1. Connected to seg
     // 2. equal to rseg
     // 3. In the tier intier
{
  if (seg→is_connectable()) {
    ConnectableSegment *cs = (ConnectableSegment *)seg;
    SegList* infs = cs→inferiors();

    for (Pix p = infs→first(); p ≠ NULL; infs→next(p))
      if ((*infs)[p]→is_in_tier(intier) &&
          rseg→equal(convert(infs, p), ap_tier, num_tiers))
        segs→insert((*infs)[p]);

    delete infs;
  }
}

Pix Tier::preceding(Segment* s1, Segment* s2)
     // Returns position of the first of
     // the two segments, or NULL if neither is in the tier.

{
  Pix currpos = segments.first();
  while (currpos && *segments[currpos] ≠ *s1 && *segments[currpos] ≠ *s2)
    segments.next(currpos);

  return (currpos);
}

int Tier::precedes(Segment* s1, Segment* s2)
     // Is s1 found on Tier before or at the same time as s2?
{
  Pix currpos = segments.first();
  while (currpos) {
    if (*segments[currpos] == *s1)
      return TRUE;
    if (*segments[currpos] == *s2)
      return FALSE;
    segments.next(currpos);
  }
```

```
  return FALSE;
}

SegList* matching_segs(Tier& tr, Tier& intier, Tier& ot, Pix tierpos,
                       int no_worddivs, int no_morphdivs)
     // Returns a list of the segments in tr starting at
     // tierpos that match segments
     // in the rule tier ot that connect to intier
{
  SegList* segs = new SegList;
  Pix curc = tierpos;
  Pix oldc;
  Pix curr;
  for (curr = ot.first(); curr != NULL; ot.next(curr)) {
    if (ot[curr]->is_connectable())
      if (((ConnectableSegment *)ot[curr])->connects_to_tier(intier))
        segs->append(tr[curc]);
    if (ot[curr]->is_zero())
      do {
        oldc = curc;
        curc = ot[curr]->zeromatches(curc, tr, ot);
        while ((no_worddivs && tr[curc]->is_a_wordboundary()) ||
               (no_morphdivs && tr[curc]->is_a_morphemeboundary()))
          tr.next(curc);
      } while (curc != oldc);
    else {
      tr.next(curc);
      while ((no_worddivs && tr[curc]->is_a_wordboundary()) ||
             (no_morphdivs && tr[curc]->is_a_morphemeboundary()))
        tr.next(curc);
    }
  }
  return segs;
}

Pix Tier::first_to_tier(Tier& tr, Pix tier1pos, Pix tier2pos, Tier& ot,
                        ConnectableSegment* rseg, Tier **ap_tier,
                        int ntiers, int no_wdivs, int no_mdivs)
     // Finds the first position in tr that connects to a
     // previously matched segment in this tier and equals
     // rseg. If this position precedes tier2pos, returns it.
     // Otherwise returns tier2pos. If no position is found, returns
     // tier2pos.
{
  SegmentSet *testsegs = new SegmentSet;
  Pix currpos = tier1pos;
  Pix i;
  SegList* msegs = matching_segs(*this, tr, ot, tier1pos, no_wdivs, no_mdivs);

  if (currpos == NULL) {
    delete testsegs;
    return tier2pos;
  }

  for (i = msegs->first(); i != NULL; msegs->next(i))
    make_set_of_segs_in_tier((*msegs)[i], testsegs, tr, rseg, ap_tier, ntiers);

  currpos = NULL;
  if (tier2pos == NULL) {
```

```
    if (!testsegs→empty()) {
      i = testsegs→first();
      currpos = tr.find((*testsegs)[i]);
      testsegs→next(i);
      while (i ≠ NULL) {
        currpos = tr.preceding(tr.segments[currpos], (*testsegs)[i]);
        testsegs→next(i);
      }
    }
  } else {
    currpos = tier2pos;
    for (i = testsegs→first(); i ≠ NULL; testsegs→next(i))
      currpos = tr.preceding(tr.segments[currpos], (*testsegs)[i]);
  }

  delete testsegs;
  return currpos;
}

int apply(Rule& rule, Chart& chart)
      // Matches rule against chart. If matched, applies
      // rule to chart, and continues to match and apply in the
      // order specified in rule until it no longer matches.
{
  int *tier_idx;
  int applied = FALSE;
  Tier **ap_tier;
  int num_ap_tiers = 0;
  int different;
  int i, j;
  int someleft = TRUE;
  Pix *origpos = (Pix *)malloc(rule.num_tiers * sizeof(Pix));

  ap_tier = (Tier **)malloc(rule.num_tiers*sizeof(Tier *));
  tier_idx = new int[rule.num_tiers];

  for (i=0; i<rule.num_tiers; i++)
    for (j=0; j<chart.num_tiers; j++)
      if (chart[j].name_eq(*rule.original[i])) {
        ap_tier[num_ap_tiers++] = &chart[j];
        tier_idx[num_ap_tiers-1] = j;
        break;
      }

  if (num_ap_tiers ≠ rule.num_tiers)
    abort();

  if (rule.is_sandhi) // Start at the beginning, so as to get word boundaries.
    for (i=0; i<rule.num_tiers; i++) {
      origpos[i] = chart[tier_idx[i]].current;
      chart[tier_idx[i]].current = chart[tier_idx[i]].first();
    }

  if (rule.rtol) // Start at end, for rtol rule.
    for (i=0; i<rule.num_tiers; i++) {
      origpos[i] = chart[tier_idx[i]].current;
      chart[tier_idx[i]].current = chart[tier_idx[i]].last();
    }
```

```
Pix ∗match_idx = (Pix ∗)malloc(rule.num_tiers ∗ sizeof(Pix));
Pix ∗old_match = (Pix ∗)malloc(rule.num_tiers ∗ sizeof(Pix));
Pix ∗start_pos = (Pix ∗)malloc(rule.num_tiers ∗ sizeof(Pix));

for (i=0; i<rule.num_tiers; i++) {
  start_pos[i] = chart[tier_idx[i]].current;
  old_match[i] = NULL;
}

someleft = TRUE;

while (someleft) {
  for (i=0; i<rule.num_tiers; i++)
    do {
      match_idx[i]=rule.match(chart[tier_idx[i]],ap_tier, rule.num_tiers, i);
      if (rule.rtol && match_idx[i] == (Pix)(-1))
        chart[tier_idx[i]].prev(chart[tier_idx[i]].current);
    } while (rule.rtol && match_idx[i] == (Pix)(-1) &&
             chart[tier_idx[i]].current ≠ NULL);

  // Did it actually match?
  for (i=0; i<rule.num_tiers; i++)
    if (match_idx[i] == (Pix)(-1)) { // It didn't match, so can't apply.
      if (rule.is_sandhi || rule.rtol) // Reset so other rules won't break
        for (j=0; j<rule.num_tiers; j++)
          chart[tier_idx[j]].current = origpos[j];
      else if (!rule.rtol)
        for (j=0; j<rule.num_tiers; j++)
          chart[tier_idx[j]].current = start_pos[j];
      free((char ∗)origpos);
      free((char ∗)old_match);
      free((char ∗)match_idx);
      free((char ∗)start_pos);
      return applied;
    }

  for (i=1; i<rule.num_tiers; i++)
    for (j=0; j<i; j++) {
      Tier∗ tr = rule.original[i];
      ConnectableSegment∗ cs = (ConnectableSegment ∗)(∗tr)[tr→first()];
      match_idx[i] = chart[tier_idx[j]].first_to_tier(chart[tier_idx[i]],
                                                       match_idx[j],
                                                       match_idx[i],
                                                       ∗rule.original[j],
                                                       cs, ap_tier,
                                                       rule.num_tiers,
                                                       rule.no_worddivs,
                                                       rule.no_morphdivs);
    }

  different = FALSE;
  for (i=0; i<rule.num_tiers; i++)
    if (match_idx[i] ≠ old_match[i])
      different = TRUE;

  if (different) {
    // Now apply it!
    rule.application(chart, match_idx, tier_idx);
    applied = TRUE;
```

```
            for (i=0; i<rule.num_tiers; i++)
                old_match[i] = match_idx[i];
        }

        someleft = TRUE;
        for (i=0; i<rule.num_tiers; i++) {
            if (different)
                if (match_idx[i]) // Will change if segment was deleted
                    chart[tier_idx[i]].current = match_idx[i];
                else
                    chart[tier_idx[i]].current = start_pos[i];
            if (rule.rtol)
                chart[tier_idx[i]].prev(chart[tier_idx[i]].current);
            else
                chart[tier_idx[i]].next(chart[tier_idx[i]].current);
            if (chart[tier_idx[i]].current == NULL)
                someleft = FALSE;
        }
    }
    if (rule.is_sandhi || rule.rtol) // Reset so other rules won't break
        for (j=0; j<rule.num_tiers; j++)
            chart[tier_idx[j]].current = origpos[j];
    else if (!rule.rtol)
        for (j=0; j<rule.num_tiers; j++)
            chart[tier_idx[j]].current = start_pos[j];
    free((char *)origpos);
    free((char *)old_match);
    free((char *)match_idx);
    free((char *)start_pos);
    return applied;
}

Pix matches_any_map_el(Segment* seg, Map *map, int i)
        // Does seg match any element in map tier number i?
        // Returns the position in map if so, or NULL.
{
    for (Pix c = map[i].first(); c ≠ NULL; map[i].next(c))
        if (*seg == *(map[i][c]→rule_seg))
            return c;
    return NULL;
}

Pix matches_any_replacement_el(Segment* seg, Tier **replacement, int i)
        // Does seg match any element in replacement, tier number
        // i? Returns the position in replacement if so, or NULL.
{
    for (Pix c = replacement[i]→first(); c ≠ NULL; replacement[i]→next(c))
        if (*seg == *(*replacement[i])[c])
            return c;
    return NULL;
}

Pix in_map(Segment* seg, Map *map, int map_size, int& i)
        // Finds rule segment seg in map (which has map_size
        // tiers). Sets i to the tier on which seg is found, and
        // returns the location on the tier (or NULL if not found)
{
    for (i=0; i<map_size; i++)
        for (Pix j=map[i].first(); j ≠ NULL; map[i].next(j))
```

```
      if (*(map[i][j]→rule_seg) == *seg)
        return j;
  return NULL;
}

Pix skeletal_in_map(Segment* seg, Map* map, int map_size, int *tier_idx,
                    Chart& chart, int& i)
      // Finds chart segment seg in map (which has map_size
      // tiers). Sets i to the tier on which seg is found, and
      // returns the location on the tier (or NULL if not found)
{
  Segment* mseg;
  for (i=0; i<map_size; i++)
    for (Pix p = map[i].first(); p ≠ NULL; map[i].next(p)) {
      mseg = chart[tier_idx[i]][map[i][p]→chart_pos];
      if (*mseg == *seg)
        return p;
    }
  return NULL;
}

Pix in_replacement_chart(Segment* seg, Tier **replacement, int num_tiers,
                         int& i)
      // Finds rule segment seg in replacement (which has
      // num_tiers tiers). Sets i to the tier on which seg
      // is found, and returns the location on the tier (or NULL if not found)
{
  for (i=0; i<num_tiers; i++)
    for (Pix j=replacement[i]→first(); j ≠ NULL; replacement[i]→next(j))
      if (*(*replacement[i])[j] == *seg)
        return j;
  return NULL;
}

void ConnectableSegment::sort()
      // Puts inferiors and superiors in the order they are found on their tiers.
{
  int i, j;
  int swapped = TRUE;
  ConnectableSegment* tmp;

  while (swapped) {
    swapped = FALSE;
    for (i=0; i<num_inferiors; i++)
      for (j=i+1; j<num_inferiors; j++)
        if (inferior[i]→tier && inferior[j]→tier &&
            inferior[i]→is_actually_in_tier(*inferior[i]→tier) &&
            inferior[j]→is_actually_in_tier(*inferior[j]→tier) &&
            *inferior[i]→tier == *inferior[j]→tier &&
            inferior[j]→tier→precedes(inferior[j], inferior[i])) {
          tmp = inferior[i];
          inferior[i] = inferior[j];
          inferior[j] = tmp;
          swapped = TRUE;
        }
    for (i=0; i<num_superiors; i++)
      for (j=i+1; j<num_superiors; j++)
        if (superior[i]→tier && superior[j]→tier &&
            superior[i]→is_actually_in_tier(*superior[i]→tier) &&
```

```
        superior[j]→is_actually_in_tier(*superior[j]→tier) &&
        *superior[i]→tier == *superior[j]→tier &&
        superior[j]→tier→precedes(superior[j], superior[i])) {
      tmp = superior[i];
      superior[i] = superior[j];
      superior[j] = tmp;
      swapped = TRUE;
    }
  }
}

int ConnectableSegment::not_too_many_tones()
     // Returns FALSE if this is a Vowel connected to a number of tones
     // greater than or equal to the maximum allowable. Otherwise, returns TRUE.
{
  if (typ ≠ V)
    return TRUE;
  int num_tones = 0;
  for (int i=0; i<num_inferiors; i++)
    if (inferior[i]→typ == TN)
      num_tones++;
  if (chart.max_tones_per_vowel == -1 || num_tones < chart.max_tones_per_vowel)
    return TRUE;
  return FALSE;
}

int ConnectableSegment::not_too_many_vowels()
     // Returns FALSE if this is a tone connected to a number of vowels
     // greater than or equal to the maximum allowable. Otherwise, returns TRUE.
{
  if (typ ≠ TN)
    return TRUE;
  int num_vowels = 0;
  for (int i=0; i<num_superiors; i++)
    if (superior[i]→typ == V)
      num_vowels++;
  if (chart.max_vowels_per_tone == -1 || num_vowels<chart.max_vowels_per_tone)
    return TRUE;
  return FALSE;
}

int ConnectableSegment::connect(ConnectableSegment* seg)
     // Requires -- this segment is assumed to be the SUPERIOR, and seg is
     // assumed to be the INFERIOR.
     // Effects: Modifies this and seg such that they are connected,
     // dealing with the case where the connection would exceed the
     // tone/vowel connection limit.
{
  int i, j, delete_num;
  int connected_properly = TRUE;
  for (i=0; i<num_inferiors; i++) // Make sure is not already connected.
    if (*this == *seg)
      return TRUE;

  ConnectableSegment **inf;
  Vowel* vow = new Vowel;
  Tone* t = new Tone(1);
  if (!t→type_eq(seg) || not_too_many_tones())
    if (num_inferiors < infsz)
```

```
      inferior[num_inferiors++] = seg;
    else {
      infsz = 2*infsz; // Double the number of connections possible!
      inf = (ConnectableSegment **)malloc(infsz*sizeof(ConnectableSegment *));
      for (i=0; i<num_inferiors; i++)
        inf[i] = inferior[i];
      inf[num_inferiors++] = seg;
      free((char *)inferior);
      inferior = inf;
    }
  else {
    connected_properly = FALSE;
    for (i=0; i<num_inferiors; i++)
      if (t→type_eq(inferior[i])) {
        for (j=0; j<inferior[i]→num_superiors; j++)
          if (*inferior[i]→superior[j] == *this) {
            delete_num = j;
            break;
          }
        for (j=delete_num; j<(inferior[i]→num_superiors-1); j++)
          inferior[i]→superior[j] = inferior[i]→superior[j+1];
        inferior[i]→num_superiors--;
        inferior[i] = seg;
        break;
      }
  }
  sort();

  // Connect seg to this segment.
  ConnectableSegment **sup;
  if (!vow→type_eq(this) || seg→not_too_many_vowels())
    if (seg→num_superiors < seg→supsz)
      seg→superior[seg→num_superiors++] = this;
    else {
      seg→supsz = 2 * seg→supsz; // Double the number of connections possible
      sup = (ConnectableSegment **)malloc(seg→supsz *
                                          sizeof(ConnectableSegment *));
      for (i=0; i<seg→num_superiors; i++)
        sup[i] = seg→superior[i];
      sup[seg→num_superiors++] = this;
      free((char *)seg→superior);
      seg→superior = sup;
    }
  else {
    connected_properly = FALSE;
    for (i=0; i<num_superiors; i++)
      if (vow→type_eq(superior[i])) {
        for (j=0; j<superior[i]→num_inferiors; j++)
          if (*superior[i]→inferior[j] == *this) {
            delete_num = j;
            break;
          }
        for (j=delete_num; j<(superior[i]→num_inferiors-1); j++)
          superior[i]→inferior[j] = superior[i]→inferior[j+1];
        superior[i]→num_inferiors--;
        superior[i] = seg;
        break;
      }
  }
```

```
  seg→sort();
  delete t;
  delete vow;
  return connected_properly;
}

void ConnectableSegment::disconnect(ConnectableSegment* seg)
      // Requires: this is the superior, seg is the inferior
      // Effect: disconnects the segments.
{
  int delete_num = num_inferiors+1, i;
  for (i=0; i<num_inferiors; i++)
    if (*(inferior[i]) == *seg) {
      delete_num = i;
      break;
    }
  if (delete_num > num_inferiors)
    return; // Not connected.
  for (i=delete_num; i<(num_inferiors-1); i++)
    inferior[i] = inferior[i+1];
  num_inferiors--;

  delete_num = seg→num_superiors+1;
  for (i=0; i<seg→num_superiors; i++)
    if (*(seg→superior[i]) == *this) {
      delete_num = i;
      break;
    }
  if (delete_num > seg→num_superiors)
    return; // Not connected.
  for (i=delete_num; i<(seg→num_superiors-1); i++)
   seg→superior[i] = seg→superior[i+1];
  seg→num_superiors--;
}

void break_crossings(ConnectableSegment* seg1, ConnectableSegment* seg2)
      // Breaks any association lines crossed by the connection of seg1
      // and seg2. Requires that seg1 and seg2 are not
      // yet connected.
{
  Pix s1pos = seg1→tier→find(seg1);
  Pix s2pos = seg2→tier→find(seg2);
  Pix i, j;
  ConnectableSegment *s1, *s2;

  i = s1pos;
  j = s2pos;
  seg1→tier→prev(i);
  seg2→tier→next(j);
  while (i ≠ NULL && j ≠ NULL) {
    if((*seg1→tier)[i]→connects_directly_to((*seg2→tier)[j]))
      {
        s1 = (ConnectableSegment *)((*seg1→tier)[i]);
        s2 = (ConnectableSegment *)((*seg2→tier)[j]);
        s1→disconnect(s2);
      }
    seg1→tier→prev(i);
    seg2→tier→next(j);
  }
```

```
  i = s1pos;
  j = s2pos;
  seg1→tier→next(i);
  seg2→tier→prev(j);
  while (i ≠ NULL && j ≠ NULL) {
    if((*seg1→tier)[i]→connects_directly_to((*seg2→tier)[j]))
      {
        s1 = (ConnectableSegment *)((*seg1→tier)[i]);
        s2 = (ConnectableSegment *)((*seg2→tier)[j]);
        s1→disconnect(s2);
      }
    seg1→tier→next(i);
    seg2→tier→prev(j);
  }
}

SegList *find_free_associates(SegList* choices, ConnectableSegment* seg,
                              Chart& chart)
     // Return all the elements of choices that freely associate with
     // seg.
{
  SegList *list = new SegList;
  for (Pix p=choices→first(); p ≠ NULL; choices→next(p))
    if (chart.freely_associate((*choices)[p], seg))
      list→append((*choices)[p]);
  return list;
}

void ConnectableSegment::no_duplicate_features(ConnectableSegment* seg,
                                               Map* map, int ntiers,
                                               Chart& chart, int* tier_idx)
     // Requires - this has not yet been connected to seg, and
     // this is not connected to more than one feature with
     // the same name.
     // Effects - If this is a classnode and seg is a feature,
     // disconnects this from any features with the same name as
     // seg.
{
  ClassNode* cn = new ClassNode;
  Feature* f = new Feature("blah");
  int loc;
  Pix match;
  if (cn→type_eq(this) && f→type_eq(seg))
    for (int i=0; i<num_inferiors; i++)
      if (f→type_eq(inferior[i]) &&
          ((Feature *)inferior[i])→name_eq((Feature *)seg)) {
        match = skeletal_in_map(inferior[i], map, ntiers, tier_idx, chart,loc);
        if (match) {
          ConnectableSegment *minf, *msup;
          minf = (ConnectableSegment *)map[loc][match]→rule_seg;
          match = skeletal_in_map(this, map, ntiers, tier_idx, chart, loc);
          if (match) {
            msup = (ConnectableSegment *)map[loc][match]→rule_seg;
            msup→disconnect(minf);
          }
        }
        disconnect(inferior[i]);
        break;
```

```
        }
    delete cn;
    delete f;
}

void add_connection(ConnectableSegment* sup, ConnectableSegment* inf,
                    Chart& chart, Map* map, int ntiers, int* tier_idx,
                    int remove_dups =0)
      // Make a connection between sup and inf, accounting for things like
      // freely associating segments, line crossing, and language-specific
      // tonal parameters.
      // Note - assumes sup is superior to inf.
{
    ConnectableSegment* seg;

    if (chart.freely_associate(sup, inf))
      seg = sup;
    else {
      SegList *infs = sup->inferiors();
      SegList *fass = find_free_associates(infs, inf, chart);
      if (fass->empty())
        error("Unable to make connection.");
      seg = (ConnectableSegment *)fewest_inferiors(fass);
    }
    break_crossings(seg, inf);
    if (remove_dups && !ks)
      seg->no_duplicate_features(inf, map, ntiers, chart, tier_idx);
    seg->modified = TRUE;
    inf->modified = TRUE;
    seg->connect(inf);
}

void ConnectableSegment::copy_aux(ConnectableSegment* seg,
                                  Map *map, int num_tiers,
                                  int *tier_index, Chart& chart,
                                  Tier** replacement)
      // Modifies this to have approximately the same connections as
      // seg, and modifies the map appropriately.
{
    int i;

    // Note -- don't need to add any connections not already in the map --
    // they'll be added later.
    // Later Note -- unless, of course, they aren't in the replacement
    // chart either (except as inferiors or superiors, of course)

    ConnectableSegment *sup, *inf;
    Segment *mel;
    Pix match;
    int maplevel;
    int was_replace;

    was_replace = FALSE;

    // Connect in the chart, for replacement.
    match = in_map(this, map, num_tiers, maplevel);
    map[maplevel].next(match);
    if (match) {
      mel = map[maplevel][match]->rule_seg;
```

```
    was_replace = !matches_any_replacement_el(mel, replacement, maplevel);

    if (was_replace) {
      match = map[maplevel][match]→chart_pos;
      inf = (ConnectableSegment *)chart[tier_index[maplevel]][match];
      for (i=0; i<inf→num_superiors; i++)
        add_connection(inf→superior[i], seg, chart, map,num_tiers,tier_index);
      for (i=0; i<inf→num_inferiors; i++)
        add_connection(seg, inf→inferior[i], chart, map,num_tiers,tier_index);
    }
  }

  // Connect based on the map.
  if (num_superiors) {
    match = in_map(this, map, num_tiers, maplevel);
    inf = (ConnectableSegment *)map[maplevel][match]→rule_seg;
  }
  for (i = 0; i<num_superiors; i++) {
    match = in_map(superior[i], map, num_tiers, maplevel);
    if (match) {
      sup = (ConnectableSegment *)map[maplevel][match]→rule_seg;
      sup→connect(inf);
      if (!was_replace) {
        match = map[maplevel][match]→chart_pos;
        sup = (ConnectableSegment *)(chart[tier_index[maplevel]][match]);
        add_connection(sup, seg, chart, map, num_tiers, tier_index);
      }
    } else {
      match=in_replacement_chart(superior[i],replacement, num_tiers, maplevel);
      if (match == NULL) {
        ConnectableSegment* tmp = superior[i];
        superior[i]→disconnect(this);
        sup = (ConnectableSegment *)tmp→copy();
        add_connection(sup, seg, chart, map, num_tiers, tier_index);
        tmp→connect(this);
      }
    }
  }

  if (num_inferiors) {
    match = in_map(this, map, num_tiers, maplevel);
    sup = (ConnectableSegment *)map[maplevel][match]→rule_seg;
  }
  for (i = 0; i<num_inferiors; i++) {
    match = in_map(inferior[i], map, num_tiers, maplevel);
    if (match) {
      inf = (ConnectableSegment *)map[maplevel][match]→rule_seg;
      sup→connect(inf);
      if (!was_replace) {
        match = map[maplevel][match]→chart_pos;
        inf = (ConnectableSegment *)(chart[tier_index[maplevel]][match]);
        add_connection(seg, inf, chart, map, num_tiers, tier_index);
      }
    } else {
      match=in_replacement_chart(inferior[i],replacement, num_tiers, maplevel);
      if (match == NULL) {
        inf = (ConnectableSegment *)inferior[i]→copy();
        add_connection(seg, inf, chart, map, num_tiers, tier_index);
      }
```

```
    }
  }
}

void ConnectableSegment::detach()
      // Detach from all connections prior to deletion of this.
{
  while (num_superiors)
    superior[0]→disconnect(this);
  while (num_inferiors)
    disconnect(inferior[0]);
  free((char *)superior);
  free((char *)inferior);
  superior = NULL;
  inferior = NULL;
}

void Tier::del(Pix& loc)
      // Delete segment at loc, and remove it from the tier.
{
  Segment* seg_to_be_deleted = segments[loc];
  seg_to_be_deleted→detach(); // Make sure nothing else connects to it.
  segments.del(loc);
  delete seg_to_be_deleted;
}

void Tier::metathesize(Pix& map_index, Pix match_loc, Pix curr, int i,
                              Tier **replacement, Map *map)
      // Move segment at map_index to the next position after
      // match_loc, in both the chart and map. Note that the locations
      // mentioned are in the map, which should contain the chart locations as
      // well.
{
  Pix match_index, index, oldidx, find_index, precede_index;
  Segment* seg = segments[map[i][map_index]→chart_pos];

                                          // Update the chart:

  // Remove segment from original position:
  match_index = map[i][map_index]→chart_pos;
  segments.del(match_index);

  // Find the place to put the segment:
  index = curr;
  find_index = match_loc;
  do {
    while (index ≠ find_index) {
      oldidx = index;
      replacement[i]→next(index);
    }
    precede_index = matches_any_map_el((*replacement[i])[oldidx], map, i);
    if (precede_index == NULL)
      find_index = oldidx;
  } while (precede_index == NULL);

  // Put it there:
  index = segments.ins_after(map[i][precede_index]→chart_pos, seg);

  // Update the map:
```

```
  match_data* md = new match_data;
  md->rule_seg = map[i][map_index]->rule_seg->surface_copy();

  if (map[i][map_index]->rule_seg->is_connectable()) {
    ConnectableSegment *copyee, *copy;
    copyee = (ConnectableSegment *)map[i][map_index]->rule_seg;
    copy = (ConnectableSegment *)md->rule_seg;
    int i;
    while (copyee->num_inferiors > 0) {
      copy->connect(copyee->inferior[0]);
      copyee->disconnect(copyee->inferior[0]);
    }
    while (copyee->num_superiors > 0) {
      copyee->superior[0]->connect(copy);
      copyee->superior[0]->disconnect(copyee);
    }
  }

  md->chart_pos = index;
  map[i].del(map_index);
  map[i].ins_after(precede_index, md);
}

int min(int *d, int num)
     // Return the minimum element of d[]
{
  int min = 99999999;
  int i;

  for (i=0; i<num; i++)
    if (d[i] != -1 && d[i]<min)
      min = d[i];
  if (min == 99999999)
    min = -1;
  return min;
}

ConnectableSegment* generalized_eqv(ConnectableSegment* a)
     // Return closest segment matching a that would be found in the
     // chart tree.
{
  Vowel *vow = new Vowel;
  Consonant *cons = new Consonant;
  Tone *t = new Tone(1);
  Phonemic *p = new Phonemic;
  ConnectableSegment *astandin;

  if (a->type_eq(vow) || a->type_eq(cons)) {
    astandin = new X;
    delete vow; delete cons; delete t; delete p;
    return (astandin);
  }
  if (a->type_eq(t)) {
    astandin = new GenericTone;
    delete vow; delete cons; delete t; delete p;
    return (astandin);
  }
  if (a->type_eq(p)) {
    astandin = new GenericPhoneme;
```

```
    delete vow; delete cons; delete t; delete p;
    return (astandin);
  }
  return (a);
}

int contains(SegList* list, Segment* seg)
      // Does list contain something like seg?
{
  Pix p;

  for (p = list->first(); p != NULL; list->next(p))
    if (seg->eqv((*list)[p]))
      return TRUE;
  return FALSE;
}

SegList *ConnectableSegment::inferiors()
      // Return the recursive inferiors of this
{
  SegList *list = new SegList;
  for (int i=0; i<num_inferiors; i++) {
    list->append(inferior[i]);
    list->join(inferior[i]->inferiors());
  }
  return list;
}

SegList *ConnectableSegment::superiors()
      // Return the recursive superiors of this
{
  SegList *list = new SegList;
  for (int i=0; i<num_superiors; i++) {
    list->append(superior[i]);
    list->join(superior[i]->superiors());
  }
  return list;
}

SegList *ConnectableSegment::direct_superiors()
      // Return the direct superiors of this
{
  SegList *list = new SegList;
  for (int i=0; i<num_superiors; i++) {
    list->append(superior[i]);
  }
  return list;
}

int Chart::is_superior(ConnectableSegment* a, ConnectableSegment* b)
{
  if (tree->eqv(a))
    return TRUE;
  else if (tree->eqv(b))
    return FALSE;
  else {
    SegList *list = tree->connects_to(generalized_eqv(a));
    ConnectableSegment* aaa = convert(list, list->first());
    if (contains(aaa->inferiors(), generalized_eqv(b)))
```

```
      return TRUE;
    return FALSE;
  }
}

Segment *fewest_inferiors(SegList* choices)
      // Friend to ConnectableSegment
{
  int min = 99999999;
  Segment* fewest = NULL;

  for (Pix p=choices→first(); p ≠ NULL; choices→next(p))
    if (convert(choices, p)→num_inferiors < min) {
      min = convert(choices, p)→num_inferiors;
      fewest = (*choices)[p];
    }
  return fewest;
}

Pix Tier::find(Segment* seg)
      // Return the location of seg in tier, or NULL if not found.
{
  for (Pix i = segments.first(); i ≠ NULL; segments.next(i))
    if (*segments[i] == *seg)
      return i;
  return NULL;
}

int ConnectableSegment::connects_directly_to(Segment* seg)
      // Does this connect directly to seg?
{
  if (seg→is_connectable()) {
    ConnectableSegment *cs = (ConnectableSegment *)seg;
    int i;
    for (i=0; i<num_inferiors; i++)
      if ((*inferior[i]) == *cs)
        return TRUE;
    for (i=0; i<num_superiors; i++)
      if ((*superior[i]) == *cs)
      return TRUE;
    return FALSE;
  } else
    return FALSE;
}

int exactly_contains(SegList* list, Segment* seg)
      // Tests by ==
      // Is seg in list?
{
  for (Pix p = list→first(); p ≠ NULL; list→next(p))
    if (*seg == *(*list)[p])
      return TRUE;
  return FALSE;
}

void ConnectableSegment::break_connection(ConnectableSegment* seg)
      // Assumes this is superior to seg.
{
  int i;
```

```
  if (connects_directly_to(seg)) {
    disconnect(seg);
    return;
  }

  for (i=0; i<seg→num_superiors; i++)
    if (exactly_contains(seg→superior[i]→superiors(), this))
      seg→superior[i]→disconnect(seg);
}

void ConnectableSegment::spread(Pix spreadpos, Tier& ctier, Chart& chart)
      // spreads along ctier starting from spreadpos, in chart.
{
  Pix i;
  ConnectableSegment *seg;
  int num;
  int is_sup = FALSE;
  Vowel *vow = new Vowel;
  Tone *t = new Tone(1);

  if (contains(inferiors(), ctier[spreadpos])) { // this is superior to csp
    num = num_inferiors;
    is_sup = TRUE;
  } else {
    num = num_superiors;
  }

  if (spreads_right) {
    i = spreadpos;
    ctier.next(i);
    while (i ≠ NULL &&
           (is_sup ? !t→type_eq(ctier[i]) || not_too_many_tones() :
            !vow→type_eq(ctier[i]) || not_too_many_vowels()) &&
           !ctier[i]→connects_to_tier(*tier) && !ctier[i]→is_a_boundary())
      {
        if (ctier[i]→is_connectable()) {
          seg = (ConnectableSegment *)ctier[i];
          if (chart.freely_associate(this, seg))
            if (is_sup)
              connect(seg);
            else
              seg→connect(this);
        }
      ctier.next(i);
    }
  }
  if (spreads_left) {
    i = spreadpos;
    ctier.prev(i);
    while (i ≠ NULL &&
           (is_sup ? !t→type_eq(ctier[i]) || not_too_many_tones() :
            !vow→type_eq(ctier[i]) || not_too_many_vowels()) &&
           !ctier[i]→connects_to_tier(*tier) && !ctier[i]→is_a_boundary())
      {
        if (ctier[i]→is_connectable()) {
          seg = (ConnectableSegment *)ctier[i];
          if (chart.freely_associate(this, seg))
            if (is_sup)
```

```
                connect(seg);
            else
                seg→connect(this);
        }
        ctier.prev(i);
    }
}
delete t;
delete vow;
}

int Rule::adjust_connections(ConnectableSegment* crel,
                             ConnectableSegment* cmel,
                             Map *map, Pix mpos,
                             Chart& chart, int i, int *tier_index)
    // Returns true if made a connection.
    // Adjust chart and map connections of the segment pointed to by cmel
    // such that they match the connections of crel, and apply spreading
    // if necessary.
{
    Tier& tier = chart[tier_index[i]];
    Pix match_loc;
    int loc;
    int made_connection = FALSE;
    ConnectableSegment *sup, *inf;

    if (crel→num_inferiors ≠ cmel→num_inferiors ||
        crel→num_superiors ≠ cmel→num_superiors) {
        int ii, jj;
        for (ii=0; ii<crel→num_inferiors; ii++) {
            int has_that_one = FALSE;
            for (jj=0; jj<cmel→num_inferiors; jj++)
                if (*(crel→inferior[ii]) == *(cmel→inferior[jj]))
                    has_that_one = TRUE;
            if (!has_that_one) {
                match_loc = in_map(crel→inferior[ii], map, num_tiers, loc);
                if (match_loc ≠ NULL) {
                    Tier& tier2 = chart[tier_index[loc]];
                    inf=(ConnectableSegment *)(tier2[map[loc][match_loc]→chart_pos]);
                    sup=(ConnectableSegment *)(tier[map[i][mpos]→chart_pos]);
                    add_connection(sup, inf, chart, map, num_tiers, tier_index, TRUE);
                    inf = (ConnectableSegment *)map[loc][match_loc]→rule_seg;
                    sup = (ConnectableSegment *)map[i][mpos]→rule_seg;
                    sup→connect(inf);
                    made_connection = TRUE;
                } // Otherwise, wait, and it'll all work out...
            }
        } // end for
        for (ii=0; ii<crel→num_superiors; ii++) {
            int has_that_one = FALSE;
            for (jj=0; jj<cmel→num_superiors; jj++)
                if (*(crel→superior[ii]) == *(cmel→superior[jj]))
                    has_that_one = TRUE;
            if (!has_that_one) {
                match_loc = in_map(crel→superior[ii], map, num_tiers, loc);
                if (match_loc ≠ NULL) {
                    Tier& tier2 = chart[tier_index[loc]];
                    inf=(ConnectableSegment *)(tier2[map[loc][match_loc]→chart_pos]);
                    sup=(ConnectableSegment *)(tier[map[i][mpos]→chart_pos]);
```

```
        add_connection(inf, sup, chart, map, num_tiers, tier_index, TRUE);
        sup = (ConnectableSegment *)map[i][mpos]→rule_seg;
        inf = (ConnectableSegment *)map[loc][match_loc]→rule_seg;
        sup→connect(inf);
        made_connection = TRUE;
      } // Otherwise, wait, and it'll all work out...
    }
  } // end for
  for (ii=0; ii<cmel→num_inferiors; ii++) {
    int has_that_one = FALSE;
    for (jj=0; jj<crel→num_inferiors; jj++)
      if (*(cmel→inferior[ii]) == *(crel→inferior[jj]))
        has_that_one = TRUE;
    if (!has_that_one) {
      match_loc = in_replacement_chart(cmel→inferior[ii],
                                       replacement, num_tiers, loc);
      if (match_loc ≠ NULL) {
        match_loc = in_map(cmel→inferior[ii], map, num_tiers, loc);
        Tier& tier2 = chart[tier_index[loc]];
        inf=(ConnectableSegment *)(tier2[map[loc][match_loc]→chart_pos]);
        sup=(ConnectableSegment *)(tier[map[i][mpos]→chart_pos]);
        sup→break_connection(inf);
        inf = (ConnectableSegment *)map[loc][match_loc]→rule_seg;
        sup = (ConnectableSegment *)map[i][mpos]→rule_seg;
        sup→disconnect(inf);
      } // Otherwise, wait, and it'll all work out...
    } // end if
  } // end for
  for (ii=0; ii<cmel→num_superiors; ii++) {
    int has_that_one = FALSE;
    for (jj=0; jj<crel→num_superiors; jj++)
      if (*(cmel→superior[ii]) == *(crel→superior[jj]))
        has_that_one = TRUE;
    if (!has_that_one) {
      match_loc = in_replacement_chart(cmel→superior[ii],
                                       replacement, num_tiers, loc);
      if (match_loc ≠ NULL) {
        match_loc = in_map(cmel→superior[ii], map, num_tiers, loc);
        Tier& tier2 = chart[tier_index[loc]];
        sup=(ConnectableSegment *)(tier2[map[loc][match_loc]→chart_pos]);
        inf=(ConnectableSegment *)(tier[map[i][mpos]→chart_pos]);
        sup→break_connection(inf);
        sup = (ConnectableSegment *)map[loc][match_loc]→rule_seg;
        inf = (ConnectableSegment *)map[i][mpos]→rule_seg;
        sup→disconnect(inf);
      } // Otherwise, wait, and it'll all work out...
    } // end if
  } // end for
} // end if (same number connections?)
else {
  if (crel→spreads()) {
    int ii, num;
    Pix chartpos;
    ConnectableSegment *cseg;
    if (crel→inferior_to_spread_along ≠ NULL)
      for (ii=1; ii≤crel→inferior_to_spread_along[0]; ii++) {
        num = crel→inferior_to_spread_along[ii];
        cseg = (ConnectableSegment *)tier[map[i][mpos]→chart_pos];
        cseg→spreads_right = crel→spreads_right;
```

```
          cseg→spreads_left = crel→spreads_left;
          match_loc = in_map(crel→inferior[num], map, num_tiers, loc);
          chartpos = map[loc][match_loc]→chart_pos;
          cseg→spread(chartpos, chart[tier_index[loc]], chart);
          made_connection = TRUE;
        }
      if (crel→superior_to_spread_along ≠ NULL)
        for (ii=1; ii≤crel→superior_to_spread_along[0]; ii++) {
          num = crel→superior_to_spread_along[ii];
          cseg = (ConnectableSegment *)tier[map[i][mpos]→chart_pos];
          cseg→spreads_right = crel→spreads_right;
          cseg→spreads_left = crel→spreads_left;
          match_loc = in_map(crel→superior[num], map, num_tiers, loc);
          chartpos = map[loc][match_loc]→chart_pos;
          cseg→spread(chartpos, chart[tier_index[loc]], chart);
          made_connection = TRUE;
        }
    }
  }
  return made_connection;
}

void Chart::apply_assoc_convention(Pix con1pos, Tier& tier1, Tier& tier2)
     // needs to be a friend of connectable_segment...
     // Applies the association convention to tiers 1 and 2.
{
  ConnectableSegment *seg = (ConnectableSegment *)(tier1[con1pos]);
  ConnectableSegment *conseg, *inf, *sup;
  Pix con2pos, p, p1, oldp1, q;
  int i;
  ConnectableSegment **segconnection;
  int segnc;
  int superior = FALSE;
  Vowel *vow = new Vowel;
  Tone *t = new Tone(1);

  if (is_tier_superior(tier1, tier2)) {
    segconnection = seg→inferior;
    segnc = seg→num_inferiors;
    superior = TRUE;
  } else {
    segconnection = seg→superior;
    segnc = seg→num_superiors;
  }

  for (i=0; i<segnc; i++)
    if (segconnection[i]→is_in_tier(tier2)) {
      con2pos = tier2.find(segconnection[i]);
      // Associate from right to left:
      p = con1pos; q = con2pos;
      tier2.prev(q);
      oldp1 = p;
      while (p ≠ NULL && !tier1[p]→is_a_boundary() &&
               q ≠ NULL && !tier2[q]→is_a_boundary() &&
               !tier2[q]→connects_directly_to_tier(tier1)) {
        p1 = p;
        tier1.prev(p1);
        while (p1 ≠ NULL && !tier1[p1]→is_a_boundary() &&
                 !tier1[p1]→connects_directly_to_tier(tier2) &&
```

```
            (tier1[p1]→inert() || tier2[q]→inert() ||
             !freely_associate(tier1[p1], tier2[q])))
      tier1.prev(p1);
   if (p1 ≠ NULL && !tier1[p1]→is_a_boundary() &&
         !tier1[p1]→connects_directly_to_tier(tier2)) {
      oldp1 = p1;
      if (superior) {
         sup = (ConnectableSegment *)tier1[p1];
         inf = (ConnectableSegment *)tier2[q];
      } else {
         inf = (ConnectableSegment *)tier1[p1];
         sup = (ConnectableSegment *)tier2[q];
      }
      sup→connect(inf);
      p = p1;
      tier2.prev(q); }
   else // pileup?
      if ((p1 == NULL || tier1[p1]→is_begin()) &&
            (oldp1 ≠ NULL && !tier1[oldp1]→inert() && !tier2[q]→inert() &&
             freely_associate(tier1[oldp1], tier2[q]))) {
         conseg = (ConnectableSegment *)(tier1[oldp1]);
         if (superior) {
            if (!t→type_eq(tier2[q]) || conseg→not_too_many_tones()) {
               conseg→connect((ConnectableSegment *)tier2[q]);
               tier2.prev(q);
            } else
               break;
         } else {
            if (!vow→type_eq(tier2[q]) || conseg→not_too_many_vowels()) {
               ((ConnectableSegment *)tier2[q])→connect(conseg);
               tier2.prev(q);
            } else
               break;
         }
      }
      else
         tier2.prev(q);
}
// Associate from Left to Right
p = con1pos; q = con2pos;
tier2.next(q);
oldp1 = p;
while (p ≠ NULL && !tier1[p]→is_a_boundary() &&
         q ≠ NULL && !tier2[q]→is_a_boundary() &&
         !tier2[q]→connects_directly_to_tier(tier1)) {
   p1 = p;
   tier1.next(p1);
   while (p1 ≠ NULL && !tier1[p1]→is_a_boundary() &&
            !tier1[p1]→connects_directly_to_tier(tier2) &&
            (tier1[p1]→inert() || tier2[q]→inert() ||
             !freely_associate(tier1[p1], tier2[q])))
      tier1.next(p1);
   if (p1 ≠ NULL && !tier1[p1]→is_a_boundary() &&
         !tier1[p1]→connects_directly_to_tier(tier2)) {
      oldp1 = p1;
      if (superior) {
         sup = (ConnectableSegment *)tier1[p1];
         inf = (ConnectableSegment *)tier2[q];
      } else {
```

```
            inf = (ConnectableSegment *)tier1[p1];
            sup = (ConnectableSegment *)tier2[q];
          }
          sup→connect(inf);
          p = p1;
          tier2.next(q); }
        else // pileup?
          if ((p1 == NULL || tier1[p1]→is_end()) &&
              (oldp1 ≠ NULL && !tier1[oldp1]→inert() && !tier2[q]→inert()
               && freely_associate(tier1[oldp1], tier2[q]))) {
            conseg = (ConnectableSegment *)(tier1[oldp1]);
            if (superior) {
              if (!t→type_eq(tier2[q]) || conseg→not_too_many_tones()) {
                conseg→connect((ConnectableSegment *)tier2[q]);
                tier2.next(q);
              } else
                break;
            } else {
              if (!vow→type_eq(tier2[q]) || conseg→not_too_many_vowels()) {
                ((ConnectableSegment *)tier2[q])→connect(conseg);
                tier2.next(q);
              } else
                break;
            }
          }
          else
            tier2.next(q);
      }
    }
  delete vow;
  delete t;
}

// Note that we must guarantee that anything on a given tier A must be
// uniformly superior or inferior to anything on any given tier B.

ConnectableSegment* Chart::tier_in_tree(Tier& tr) {
  if (tree→tier→name_eq(tr))
    return tree;
  SegList* infs = tree→inferiors();
  ConnectableSegment* cs;
  for (Pix p = infs→first(); p ≠ NULL; infs→next(p))
    if ((*infs)[p]→tier→name_eq(tr)) {
      cs = (ConnectableSegment *)(*infs)[p];
      delete infs;
      return cs;
    }
  delete infs;
  return (ConnectableSegment *)0;
}

int Chart::is_tier_superior(Tier& tier1, Tier& tier2)
{
  ConnectableSegment *t1, *t2;
  t1 = tier_in_tree(tier1);
  t2 = tier_in_tree(tier2);
  if (t1 && t2) {
    if (contains(t1→inferiors(), t2))
      return TRUE;
```

```
  }
  return FALSE;
}

void Chart::assoc_convention(int *tier_index, int num_tiers)
      // Finds places to apply the association convention, and applies it.
{
  int i, j;
  Pix p;
  for (i=0; i<num_tiers; i++)
    for (j=i+1; j<num_tiers; j++) {
      Tier& tier1 = *tier[tier_index[i]];
      Tier& tier2 = *tier[tier_index[j]];
      for (p = tier1.first(); p ≠ NULL; tier1.next(p))
        if (tier1[p]→modified && tier1[p]→connects_directly_to_tier(tier2))
          apply_assoc_convention(p, tier1, tier2);
    }
}

void Rule::connect_map(Map* map)
      // Connects the map segments to match the connections in rule.
{
  int i, ii, jj, loc;
  Pix p, q;
  int has_that_one;
  Pix match_loc;
  ConnectableSegment *sup, *inf;

  for (i = 0; i<num_tiers; i++)
    for (p = original[i]→first(), q = map[i].first(); p ≠ NULL && q ≠ NULL;
          original[i]→next(p), map[i].next(q))
      if ((*original[i])[p]→is_connectable()) {
        ConnectableSegment* rseg = (ConnectableSegment *)(*original[i])[p];
        ConnectableSegment* mseg = (ConnectableSegment *)map[i][q]→rule_seg;
        int& rinfs = rseg→num_inferiors;
        int& rsups = rseg→num_superiors;
        int& minfs = mseg→num_inferiors;
        int& msups = mseg→num_superiors;
        if (rinfs > minfs)
          for (ii=0; ii<rinfs; ii++) {
            has_that_one = FALSE;
            for (jj=0; jj<minfs; jj++)
              if (*(rseg→inferior[ii]) == *(mseg→inferior[jj]))
                has_that_one = TRUE;
            if (!has_that_one) {
              match_loc = in_map(rseg→inferior[ii], map, num_tiers, loc);
              if (match_loc ≠ NULL) {
                inf = (ConnectableSegment *)map[loc][match_loc]→rule_seg;
                sup = (ConnectableSegment *)map[i][q]→rule_seg;
                sup→connect(inf);
              }
            }
          }
        if (rsups > msups)
          for (ii=0; ii<rsups; ii++) {
            has_that_one = FALSE;
            for (jj=0; jj<msups; jj++)
              if (*(rseg→superior[ii]) == *(mseg→superior[jj]))
                has_that_one = TRUE;
```

```
          if (!has_that_one) {
            match_loc = in_map(rseg→superior[ii], map, num_tiers, loc);
            if (match_loc ≠ NULL) {
              sup = (ConnectableSegment *)map[i][q]→rule_seg;
              inf = (ConnectableSegment *)map[loc][match_loc]→rule_seg;
              sup→connect(inf);
            }
          }
        }
    }
}

void ConnectableSegment::delete_fully(Map* map, int num_tiers, Chart& chart,
                                      int *tier_idx)
      // Delete a segment and all its inferiors not found in the map.
{
  ConnectableSegment* tmp;
  Pix match;
  int loc;
  while (num_superiors)
    superior[0]→disconnect(this);
  while (num_inferiors) {
    tmp = inferior[0];
    if (tmp→num_superiors < 2) {
      if (!(skeletal_in_map(tmp, map, num_tiers, tier_idx, chart, loc))) {
        tmp→delete_fully(map, num_tiers, chart, tier_idx);
        if (tmp→tier) {
          match = tmp→tier→find(tmp);
          if (match)
            tmp→tier→del(match);
        }
      } else
        disconnect(tmp);
    } else
      disconnect(tmp);
  }
}

void delete_segment(Tier& tier, Map* map, int tier_num, Pix& mpos, int ntiers,
                    Chart& chart, int *tidx)
      // Delete a segment found in the map and chart but not in the replacement
      // chart.
{
  Pix loc = map[tier_num][mpos]→chart_pos;
  if (tier[loc]→is_connectable())
    ((ConnectableSegment *)tier[loc])→delete_fully(map, ntiers, chart, tidx);
  tier.del(loc);
  map[tier_num][mpos]→rule_seg→detach();
  delete map[tier_num][mpos]→rule_seg;
  map[tier_num][mpos]→rule_seg = NULL;
  delete map[tier_num][mpos];
  map[tier_num].insert_at(mpos, NULL);
  map[tier_num].del(mpos);
}

void Rule::application(Chart& chart, Pix* match_index, int *tier_index) {
  // Requires:
  // (1) Each Pix match_index[i] in match_index points to the first
  // segment of a portion of tier chart[tier_index[i]], a portion
```

```
// that matches the tier replacement[i] of the rule of which this is
// a member function. The match must be appropriate to the type of
// this rule (That is, each segment marked for exact matching must
// match exactly, by having the same number of connections to
// tiers mentioned in the rule.) Furthermore, there must be the same
// number of elements in the arrays original, replacement,
// match_index, and tier_index.
// (2) Any non-connectable segment in *original must also be found in
// *replacement. Furthermore, it must be in the same element (tier)
// in *original as in *replacement, and within a given element, the
// *order* of non-connectable segments must be the same between the
// element of *original and the corresponding element in *replacement.
// (3) The arrays original, replacement, match_index, and tier_index must
// be set up such that, for any given i greater than or equal to zero
// and less than num_tiers (a slot in the rule), the tiers referred to
// by original[i], replacement[i], match_index[i], and
// chart[tier_index[i]] all correspond (They must be name_eq,
// etc.)
// Modifies: chart, and conceivably any tier or segment referred to directly
// or indirectly by it.
// Effects: applies rule to chart, and applies the association convention
// afterwards, if possible.
Map *map = new Map[num_tiers];
match_data *curr_el;
Pix curr, curc, oldc, mpos, match_loc, oldmpos;
int i;
int mc = FALSE; // Made connection
int made_connection = FALSE;
FeatureMatrix *fm = new FeatureMatrix;

// Make map.
for (i = 0; i<num_tiers; i++) {
  Tier* map_tier = new Tier;
  map_tier→set_name(original[i]→name);
  map_tier→make_identical_to(original[i]);
  curc = match_index[i];
  for (curr = original[i]→first(); (curr ≠ NULL) && (curc ≠ NULL);
       original[i]→next(curr))
    {
      while ((no_worddivs &&
              chart[tier_index[i]][curc]→is_a_wordboundary())
             || (no_morphdivs &&
                 chart[tier_index[i]][curc]→is_a_morphemeboundary()))
        chart[tier_index[i]].next(curc);
      curr_el = new match_data;
      curr_el→rule_seg = (*original[i])[curr]→surface_copy();
      curr_el→rule_seg→make_identical_to((*original[i])[curr]);
      curr_el→rule_seg→tier = map_tier;
      curr_el→chart_pos = curc;
      map[i].append(curr_el);
      if (curr_el→rule_seg→is_zero()) // Skip zero things.
        do {
          oldc = curc;
          curc = curr_el→rule_seg→zeromatches(curc, chart[tier_index[i]],
                                               *original[i]);
          while ((no_worddivs &&
                  chart[tier_index[i]][curc]→is_a_wordboundary())
                 || (no_morphdivs &&
                     chart[tier_index[i]][curc]→is_a_morphemeboundary()))
```

```
                chart[tier_index[i]].next(curc);
            } while (curc ≠ oldc);
        else
            chart[tier_index[i]].next(curc);
    }
}

connect_map(map);

for (i=0; i<num_tiers; i++) {
    Tier& tier = chart[tier_index[i]];
    curr = replacement[i]→first();
    mpos = map[i].first();
    oldmpos = NULL;
    while (curr ≠ NULL || mpos ≠ NULL) {
        Segment* rel = NULL;
        Segment* mel = NULL;
        if (curr == NULL) { // Delete segment.
            if (map[i][mpos]→chart_pos == match_index[i])
                match_index[i] = 0;
            delete_segment(tier, map, i, mpos, num_tiers, chart, tier_index);
            continue;
        }
        if (mpos == NULL) { // Insert segment.
            rel = (*replacement[i])[curr];
            Segment* seg = rel→surface_copy();
            match_data *md = new match_data;
            md→rule_seg = rel→surface_copy();
            md→rule_seg→make_identical_to(rel);
            md→rule_seg→tier = map[i][oldmpos]→rule_seg→tier;
            md→chart_pos = tier.ins_after(map[i][oldmpos]→chart_pos, seg);
            mpos = map[i].append(md);
            if (rel→is_connectable()) {
                ConnectableSegment* cs = (ConnectableSegment *)rel;
                ConnectableSegment* cseg = (ConnectableSegment *)seg;
                cs→copy_aux(cseg, map, num_tiers, tier_index, chart, replacement);
            }
        }
        rel = (*replacement[i])[curr];
        mel = map[i][mpos]→rule_seg;
        if (*rel ≠ *mel) {
            if (matches_any_map_el(rel, map, i)) {
                match_loc = matches_any_replacement_el(mel, replacement, i);
                if (match_loc ≠ NULL) // Metathesize segment.
                    tier.metathesize(mpos, match_loc, curr, i, replacement, map);
                else { // Delete segment
                    oldmpos = mpos;
                    if (map[i][mpos]→chart_pos == match_index[i])
                        match_index[i] = 0;
                    delete_segment(tier, map, i, mpos, num_tiers, chart, tier_index);
                }
            } else { // Insert segment
                if (!matches_any_replacement_el(mel, replacement, i) &&
                        fm→type_eq(rel)) {
                    FeatureMatrix *chartfm, *relfm;
                    chartfm = (FeatureMatrix *)tier[map[i][mpos]→chart_pos];
                    relfm = (FeatureMatrix *)rel;
                    relfm→copy_features(chartfm);
                    map[i][mpos]→rule_seg→make_identical_to(rel);
```

```
            // ∧ is a cheap hack to make identical without messing with cons.
          } else {
            Segment* seg = rel→surface_copy();
            match_data *md = new match_data;
            md→rule_seg = rel→surface_copy();
            md→rule_seg→make_identical_to(rel);
            md→rule_seg→tier = map[i][mpos]→rule_seg→tier;
            md→chart_pos = tier.insert(map[i][mpos]→chart_pos, seg);
            oldmpos = mpos;
            mpos = map[i].ins_before(mpos, md);
            if (rel→is_connectable()) {
              ConnectableSegment* cs = (ConnectableSegment *)rel;
              ConnectableSegment* cseg = (ConnectableSegment *)seg;
              cs→copy_aux(cseg,map,num_tiers, tier_index, chart, replacement);
            }
          }
        } // end if (matches...)
      } // if (rel ≠ mel)
      else { // Adjust connections:
        if (rel→is_connectable()) {
          ConnectableSegment* crel = (ConnectableSegment *)rel;
          ConnectableSegment* cmel = (ConnectableSegment *)mel;
          mc = adjust_connections(crel, cmel, map, mpos, chart, i, tier_index);
          made_connection = (mc || made_connection);
        } // if (rel.is_connectable())
        replacement[i]→next(curr);
        oldmpos = mpos;
        map[i].next(mpos);
      } // end if (rel ≠ mel)
    } // end while
  } // end for
  delete[] map;
  delete fm;
  // Association convention.
  if (made_connection)
    chart.assoc_convention(tier_index, num_tiers);
}
```

C.3 Matching

```
// matching.cc

#include "tones.h"
extern ConnectableSegment* convert(SegList*, Pix);

int Tier::is_applicable(Tier **applicable_tier, int num_tiers)
     // Is this an applicable tier (is it mentioned in the rule?)
{
  for (int i=0; i<num_tiers; i++)
    if (id_num == applicable_tier[i]→id_num)
      return TRUE;
  return FALSE;
}

int matches_exactly(ConnectableSegment* rseg, ConnectableSegment* seg,
                    Tier** ap_tier, int num)
     // Does seg match rseg exactly?
```

```
{
  int num_applicable_connections =0;
  int num_connections =0;
  SegList* sups = rseg→superiors();
  SegList* infs = rseg→inferiors();
  Pix p;
  for (p = sups→first(); p ≠ NULL; sups→next(p))
    if (!(*sups)[p]→tier || (*sups)[p]→tier→is_applicable(ap_tier, num))
      num_connections++;
  for (p = infs→first(); p ≠ NULL; infs→next(p))
    if (!(*infs)[p]→tier || (*infs)[p]→tier→is_applicable(ap_tier, num))
      num_connections++;
  delete sups;
  delete infs;

  sups = seg→superiors();
  infs = seg→inferiors();
  for (p = sups→first(); p ≠ NULL; sups→next(p))
    if (!(*sups)[p]→tier || (*sups)[p]→tier→is_applicable(ap_tier, num))
      num_applicable_connections++;
  for (p = infs→first(); p ≠ NULL; infs→next(p))
    if (!(*infs)[p]→tier || (*infs)[p]→tier→is_applicable(ap_tier, num))
      num_applicable_connections++;
  delete sups;
  delete infs;
  if (num_applicable_connections ≠ num_connections)
    return FALSE;
  return TRUE;
}

int matches_roughly(ConnectableSegment* rseg, ConnectableSegment* seg,
                    Tier** ap_tier, int num)
     // Does seg match rseg roughly?
{
  int num_applicable_connections = 0;
  int num_connections = 0;
  Pix p;
  SegList* infs = rseg→inferiors();
  for (p = infs→first(); p ≠ NULL; infs→next(p))
    if (!(*infs)[p]→tier || (*infs)[p]→tier→is_applicable(ap_tier, num))
      num_connections++;
  delete infs;

  infs = seg→inferiors();
  for (p = infs→first(); p ≠ NULL; infs→next(p))
    if (!(*infs)[p]→tier || (*infs)[p]→tier→is_applicable(ap_tier, num))
      num_applicable_connections++;
  delete infs;
  if (num_applicable_connections < num_connections)
    return FALSE;
  return TRUE;
}

Pix ConnectableSegment::matches(Pix seg, Tier& tier,
                                Tier **applicable_tier,
                                int num_tiers)
     // Does the segment at location seg match this? If so
     // return the next position. If not, return NULL.
{
```

```
  Segment* segment = tier[seg];
  Pix currpos = seg;

  tier.next(currpos);

  if (segment→is_connectable()) {
    ConnectableSegment* cs = (ConnectableSegment *)segment;
    return ((eq(cs, applicable_tier, num_tiers)) ? currpos : (Pix)(-1));
  } else
    return ((eq(segment, applicable_tier, num_tiers)) ? currpos : (Pix)(-1));
}

int ConnectableSegment::equal(ConnectableSegment* seg, Tier** ap_tier,
                              int num_tiers)
     // Is seg equal to this?
{
  return (eqv(seg) && (!tier || !seg→tier || tier→name_eq(*seg→tier)) &&
          (is_exact || matches_roughly(this, seg, ap_tier, num_tiers)) &&
          (!is_exact || matches_exactly(this, seg, ap_tier, num_tiers)));
}

void sort_list(SegList* sl)
     // Sort list of segments based on tier position.
{
  if (sl→empty())
    return;

  Pix p, q;
  int swapped = TRUE;
  Segment* tmp;

  while (swapped) {
    swapped = FALSE;
    for (p = sl→first(); p ≠ NULL; sl→next(p))
      for (q = p, sl→next(q); q ≠ NULL; sl→next(q))
        if ((*sl)[p]→tier && (*sl)[q]→tier &&
            (*sl)[p]→is_actually_in_tier(*(*sl)[p]→tier) &&
            (*sl)[q]→is_actually_in_tier(*(*sl)[q]→tier) &&
            *(*sl)[p]→tier == *(*sl)[q]→tier &&
            (*sl)[q]→tier→precedes((*sl)[q], (*sl)[p])) {
          tmp = (*sl)[p];
          sl→insert_at(p, (*sl)[q]);
          sl→insert_at(q, tmp);
          swapped = TRUE;
        }
  }
}

int in_order(ConnectableSegment* seg, SegList* choices,
             Tier **ap_tier, int num)
     // Is seg the first thing on its tier within choices?
     // Requires - the segs in choices are connectable.
{
  for (Pix p = choices→first(); p ≠ NULL; choices→next(p)) {
    ConnectableSegment* cseg = (ConnectableSegment *)(*choices)[p];
    if (seg→equal(cseg, ap_tier, num))
      return TRUE;
    if (seg→tier &&
        (!(*choices)[p]→tier ||
```

```
            seg→tier→name_eq(*(*choices)[p]→tier)))
        return FALSE;
  }
  return TRUE;
}

int ConnectableSegment::delete_best_match(SegList* choices, Tier **ap_tier,
                                                          int num) {
  int min = 9999999;
  Pix minloc;

  for (Pix p = choices→first(); p ≠ NULL; choices→next(p))
    if ((*choices)[p]→is_connectable()) {
      ConnectableSegment *cs = (ConnectableSegment *)(*choices)[p];
      if (equal(cs, ap_tier, num) && in_order(cs, choices, ap_tier, num)
            && (cs→num_superiors + cs→num_inferiors < min)) {
        min = cs→num_superiors + cs→num_inferiors;
        minloc = p;
      }
    }

  if (min == 9999999)
    return FALSE;
  else {
    choices→del(minloc);
    return TRUE;
  }
}

int ConnectableSegment::eq(ConnectableSegment* seg, Tier** ap_tier, int numtrs)
{
  if (!equal(seg, ap_tier, numtrs))
    return FALSE;

  SegList* rinflist = inferiors();
  SegList* cinflist = seg→inferiors();
  ConnectableSegment* cs;
  int matches;
  int not_exact = TRUE;
  Pix p, q;

  for (p = rinflist→first(); p ≠ NULL; rinflist→next(p))
    if (convert(rinflist, p)→is_exact) {
      not_exact = FALSE;
      break;
    }

  p = cinflist→first();
  while (p ≠ NULL)
    if (not_exact && !is_exact && (*cinflist)[p]→tier &&
          !(*cinflist)[p]→tier→is_applicable(ap_tier, numtrs))
      cinflist→del(p);
    else {
      matches = FALSE;
      for (q = rinflist→first(); q ≠ NULL; rinflist→next(q))
        if (convert(rinflist, q)→equal(convert(cinflist, p), ap_tier, numtrs))
          {
            matches = TRUE;
            break;
```

```
            }
        if (matches)
            cinflist→next(p);
        else
            cinflist→del(p);
    }

  sort_list(rinflist);
  sort_list(cinflist);

  for (p = rinflist→first(); p ≠ NULL; rinflist→next(p)) {
    cs = convert(rinflist, p);
    if (!cs→delete_best_match(cinflist, ap_tier, numtrs))
        {
            delete rinflist;
            delete cinflist;
            return FALSE;
        }
  }

  delete rinflist;
  delete cinflist;
  return TRUE;
}

Pix C_0::zeromatches(Pix seg, Tier& ctier, Tier& rtier)
{
  Pix c;
  int i, count1=0, count2=0;

  c = rtier.current;
  rtier.next(c);
  while (c ≠ NULL && cons.type_eq(rtier[c])) { // num matching in rule tier.
    count1++;
    rtier.next(c);
  }
  c = seg;
  while (c ≠ NULL && cons.type_eq(ctier[c])) { // num matching in chart tier.
    count2++;
    ctier.next(c);
  }
  if (count2 < count1) // The rule can't possibly match.
    return ((Pix)(-1));
  else {
    for (i = 0; i<count1; i++)
        ctier.prev(c);
    return c;
  }
}

Pix V_0::zeromatches(Pix seg, Tier& ctier, Tier& rtier)
{
  Pix c;
  int i, count1=0, count2=0;

  c = rtier.current;
  rtier.next(c);
  while (c ≠ NULL && vow.type_eq(rtier[c])) { // num matching in rule tier.
    count1++;
```

```
    rtier.next(c);
  }
  c = seg;
  while (c ≠ NULL && vow.type_eq(ctier[c])) { // num matching in chart tier.
    count2++;
    ctier.next(c);
  }
  if (count2 < count1) // The rule can't possibly match.
    return ((Pix)(-1));
  else {
    for (i = 0; i<count1; i++)
      ctier.prev(c);
    return c;
  }
}

Pix X_0::zeromatches(Pix seg, Tier& ctier, Tier& rtier)
{
  Pix c;
  int i, count1=0, count2=0;

  c = rtier.current;
  rtier.next(c);
  while (c ≠ NULL && x.type_eq(rtier[c])) { // num matching in rule tier.
    count1++;
    rtier.next(c);
  }
  c = seg;
  while (c ≠ NULL && x.type_eq(ctier[c])) { // num matching in chart tier.
    count2++;
    ctier.next(c);
  }
  if (count2 < count1) // The rule can't possibly match.
    return ((Pix)(-1));
  else {
    for (i = 0; i<count1; i++)
      ctier.prev(c);
    return c;
  }
}

int ConnectableSegment::connects_directly_to_tier(Tier& ctier) {
  int i;

  for(i=0; i<num_superiors; i++)
    if (superior[i]→is_in_tier(ctier))
      return TRUE;
  for(i=0; i<num_inferiors; i++)
    if (inferior[i]→is_in_tier(ctier))
      return TRUE;
  return FALSE;
}

int Segment::is_in_tier(Tier& intier) {
  return (tier→name_eq(intier));
}

int Segment::is_actually_in_tier(Tier& intier) {
  if (!tier || *tier ≠ intier)
```

```
    return FALSE;
  for (Pix p = intier.first(); p ≠ NULL; intier.next(p))
    if (*intier[p] == *this)
      return TRUE;
  return FALSE;
}

int ConnectableSegment::connects_to_tier(Tier& ctier) {
  SegList *sups = topmost_superiors();
  SegList *infs = new SegList;
  Pix p;

  for (p = sups→first(); p ≠ NULL; sups→next(p)) {
    if ((*sups)[p]→is_in_tier(ctier)) {
      delete sups;
      delete infs;
      return TRUE;
    }
    infs→join(convert(sups, p)→inferiors());
  }

  for (p = infs→first(); p ≠ NULL; infs→next(p))
    if ((*infs)[p]→is_in_tier(ctier)) {
      delete sups;
      delete infs;
      return TRUE;
    }
  delete sups;
  delete infs;
  return FALSE;
}

void Rule::find_closest_usable_rule_segment(int this_tier) {
  Tier& ot = *original[this_tier];

  if (this_tier) // If there are done tiers, get the first unconnected instead.
    while (ot.current ≠ NULL && !unconnected(this_tier))
      ot.next(ot.current);
}

int Rule::unconnected(int this_tier) {
  Tier& ot = *original[this_tier];
  if (ot[ot.current]→is_connectable()) {
    SegList* sups = ((ConnectableSegment *)ot[ot.current])→superiors();
    int unconnected = TRUE;

    for (int i=0; i<this_tier; i++)
      for (Pix p = sups→first(); p ≠ NULL; sups→next(p))
        if ((*sups)[p]→is_in_tier(*original[i])) {
          delete sups;
          return FALSE;
        }

    delete sups;
  }
  return TRUE;
}

Pix Rule::match(Tier& tier, Tier **applicable_tier, int num_tiers,
```

```
                int this_tier)
     // If the rule matches tier at any point including or after tier.current,
     // returns the index (Pix) of the match. Otherwise, returns (Pix)(-1).
     // Does NOT modify tier.current.
{
  Tier& ot = *original[this_tier];
  Pix currpos, oldpos, matchpos;
  int matched;

  currpos = tier.current;
  matched = FALSE;
  matchpos = NULL;
  ot.current = ot.first();
  find_closest_usable_rule_segment(this_tier);
  if (ot.current == NULL) {
    matched = TRUE;
    return matchpos;
  }
  Segment* first_seg = ot[ot.current];

  while (!matched) {
    do {
      oldpos = currpos;
      if (first_seg→is_zero())
        currpos = first_seg→zeromatches(currpos, tier, ot);
      else
        currpos =first_seg→matches(currpos, tier, applicable_tier, num_tiers);

      if (currpos == (Pix)(-1)) {
        matched = FALSE;
        currpos = oldpos;
        tier.next(currpos); }
      else
        matched = TRUE;
    } while (!matched && currpos ≠ NULL && currpos ≠ (Pix)(-1));

    if (!matched)
      return ((Pix)(-1));

    // The first position has matched -- save the position and check the rest.
    matchpos = oldpos;

    while (matched && ot.current ≠ NULL) {
      ot.next(ot.current);
      if (ot.current == NULL)
        break; // The tier matches.
      while ((no_worddivs && tier[currpos]→is_a_wordboundary()) ||
             (no_morphdivs && tier[currpos]→is_a_morphemeboundary()))
        tier.next(currpos);
      if (currpos == NULL) {
        matched = FALSE;
        break;
      }

      Segment* test_seg = ot[ot.current];
      if (test_seg→is_zero())
        do {
          oldpos = currpos;
          currpos = test_seg→zeromatches(currpos, tier, ot);
```

```
            while ((no_worddivs && tier[currpos]→is_a_wordboundary()) ||
                   (no_morphdivs && tier[currpos]→is_a_morphemeboundary()))
              tier.next(currpos);
          } while (currpos ≠ oldpos);
        else
          currpos = test_seg→matches(currpos, tier, applicable_tier, num_tiers);
        if ((currpos == (Pix)(-1)))
          matched = FALSE;
      }

      if (!matched) {
        ot.current = ot.first();
        find_closest_usable_rule_segment(this_tier);
        currpos = matchpos;
        tier.next(currpos);
      }
    }
    return (matchpos);
}
```

C.4 Input/Output

```
// io.cc

#include "StrTable.h"
#include "StrStack.h"

int longest_phoneme;
int eof;
int eophrase;
int duple;

extern Chart chart;
extern StrTable tbl;

int is_valid_ste(StrTableEntry* ste) {
  return(ste→segment && (ste→is_phoneme || !ste→segment→is_connectable()));
}

void determine_longest_phoneme()
{
  longest_phoneme = 0;
  for (Pix p = tbl.first(); p ≠ NULL; tbl.next(p))
    if (is_valid_ste(tbl[p]) && strlen(tbl[p]→name) > longest_phoneme)
      longest_phoneme = strlen(tbl[p]→name);
}

int is_word_div(char ch) {
  return (ch == ' ' || ch == '#');
}

int is_phrase_div(char ch) {
  return (ch == '\n' || ch == '.' || ch == '%');
```

```
}

void error(const char* s1, const char* s2 = "") {
  cerr << s1 << s2 << '\n';
  exit(1);
}

char get_char(istream& infile, StrStack* stk) {
  char c;
  if (c = stk->pop())
    return c;
  else {
    if (!infile.get(c)) {
      eof = TRUE;
      return '\0';
    } else
      return c;
  }
}

void eat_word_divs(istream& infile, StrStack* stk) {
  char ch = get_char(infile, stk);
  while (ch && is_word_div(ch))
    ch = get_char(infile, stk);
  if (infile.eof())
    eof = TRUE;
  else
    stk->push(ch);
}

void eat_line(istream& infile, StrStack* stk) {
  char ch = get_char(infile, stk);
  while (ch && ch != '\n')
    ch = get_char(infile, stk);
  if (infile.eof())
    eof = TRUE;
}

void eat_word_and_phrase_divs(istream& infile, StrStack* stk) {
  char ch = get_char(infile, stk);
  while (ch && (is_word_div(ch) || is_phrase_div(ch))) {
    if (ch == '%')
      eat_line(infile, stk);
    ch = get_char(infile, stk);
  }
  if (infile.eof())
    eof = TRUE;
  else
    stk->push(ch);
}

void StrStack::push(char c)
{
  if ((top - rep + 1) > sz) {
    sz = 2 * sz;
    char* newrep = (char *)malloc(sz);
    strcpy(newrep, rep);
    top = newrep + (top-rep);
    free(rep);
```

```
        rep = newrep;
    }
    *top++ = c;
}

char StrStack::pop()
{
    if (top == rep)
        return '\0';
    else
        return *--top;
}

Segment* read_segment(istream& infile, StrStack* stk) {
    StrTableEntry *seg = NULL;
    StrTableEntry *trial;
    char *reading = (char *)malloc(longest_phoneme);
    int length = 0;
    int goodlength = 0;
    char ch;

    while (length < longest_phoneme) {
        if (!(ch = get_char(infile, stk))) {
            eof = TRUE;
            break;
        }
        if ((is_word_div(ch) || ch == '+' || is_phrase_div(ch)) && length > 0) {
            stk->push(ch);
            break;
        }
        if (is_phrase_div(ch)) {
            eophrase = TRUE;
            eat_word_and_phrase_divs(infile, stk);
            seg = tbl.find("]w");
            break;
        }
        if (is_word_div(ch)) {
            eat_word_divs(infile, stk);
            seg = tbl.find("]w");
            break;
        }
        if (ch == '+' && length == 0) {
            seg = tbl.find("]m");
            duple = TRUE;
            break;
        }
        if (ch == '%' && length == 0) {
            eat_line(infile, stk);
            eophrase = TRUE;
            eat_word_and_phrase_divs(infile, stk);
            seg = tbl.find("]w");
            break;
        }
        reading[length++] = ch;
        reading[length] = '\0';
        if (strcmp(reading, "]w") == 0)
            eat_word_divs(infile, stk);
        if ((trial = tbl.find(reading)) && is_valid_ste(trial)) {
            goodlength = length;
```

```
        seg = trial;
      }
    }
    if (seg && seg == tbl.find("w[")) {
      while (goodlength < length)
        stk→push(reading[--length]);
      free(reading);
      eat_word_divs(infile, stk);
      return (read_segment(infile, stk));
    }
    if (seg && goodlength < length)
      while (goodlength < length)
        stk→push(reading[--length]);
    if (seg) {
      free(reading);
      return seg→segment;
    } else { // Only ignore ONE bad character at a time.
      while (length > 1)
        stk→push(reading[--length]);
      free(reading);
      return (Segment *)0;
    }
}

void ConnectableSegment::safe_detach() {
  while (num_superiors)
    superior[0]→disconnect(this);
  while (num_inferiors)
    disconnect(inferior[0]);
}

void disconnect_tones(ConnectableSegment* cs)
{
  SegList* infs = cs→inferiors();
  GenericTone *tn = new GenericTone;
  for (Pix p = infs→first(); p ≠ NULL; infs→next(p))
    if (tn→type_eq((*infs)[p])) {
      convert(infs, p)→safe_detach();
      break;
    }
  delete tn;
  delete infs;
}

void Chart::add_skeletal_seg(ConnectableSegment* cs)
{
  SegList* infs;
  cs→tier = tier[0];
  tier[0]→append(cs);
  infs = cs→inferiors();
  for (Pix p = infs→first(); p ≠ NULL; infs→next(p))
    for (int i=0; i<num_tiers; i++)
      if (tier[i]→name_eq(*(*infs)[p]→tier)) {
        tier[i]→append((*infs)[p]);
        break;
      }
  delete infs;
  if (no_connect)
    disconnect_tones(cs);
```

```
}

void Chart::read_word(istream& infile, StrStack* stk)
     // friend of ConnectableSegment
{
  WordBegin *wbegin = new WordBegin;
  WordEnd *wend = new WordEnd;
  MorphemeEnd *mend = new MorphemeEnd;
  Segment *seg = NULL;
  int i;
  Pix p;
  for (i = 0; i<num_tiers; i++) {
    p = tier[i]->append(wbegin->copy());
    tier[i]->current = p;
  }
  GenericPhoneme *gp = new GenericPhoneme;
  X *x = new X;
  GenericTone *tn = new GenericTone;
  ConnectableSegment* cs;
  while (!eof && seg == NULL)
    seg = read_segment(infile, stk);
  while (!eof && !wend->type_eq(seg)) {
    if (seg->is_connectable()) {
      if (x->type_eq(seg)) {
        cs = (ConnectableSegment *)seg->copy();
        add_skeletal_seg(cs);
      } else if (tn->type_eq(seg) || gp->type_eq(seg)) {
        cs = (ConnectableSegment *)seg;
        if (cs->num_superiors == 1) {
          cs = (ConnectableSegment *)cs->superior[0]->copy();
          add_skeletal_seg(cs);
        } else if (cs->num_superiors == 0) {
          for (i = 0; i<num_tiers; i++)
            if (tier[i]->name_eq(*cs->tier)) {
              tier[i]->append(cs->copy());
              break;
            }
        } else
          error("Uncaught parse error.");
      } else
        error("Uncaught parse error.");
    } else {
      if (mend->type_eq(seg) && duple) {
        duple = FALSE;
        for (i = 0; i<num_tiers; i++)
          tier[i]->append(seg->copy());
        seg = new MorphemeBegin;
      }
      for (i = 0; i<num_tiers; i++)
        tier[i]->append(seg->copy());
    }

    seg = NULL;
    while (!eof && seg == NULL)
      seg = read_segment(infile, stk);
  }
  for (i=0; i<num_tiers; i++)
    tier[i]->append(wend->copy());
  delete tn;
```

```
  delete gp;
  delete x;
  delete wbegin;
  delete wend;
  delete mend;
}

void Chart::print_and_delete_word()
{
  Pix p;
  WordEnd *we = new WordEnd;
  for (p = tier[0]→first(); p ≠ NULL && !we→type_eq((*tier[0])[p]);
       tier[0]→next(p))
    (*tier[0])[p]→print();
  if (p ≠ NULL)
    (*tier[0])[p]→print();
  if (eophrase) {
    cout ≪ "\n";
    wordend = FALSE;
    eophrase = FALSE;
  }

  cout.flush();

  // Delete the word.
  for (int i = 0; i<num_tiers; i++) {
    p = tier[i]→first();
    while (p ≠ NULL && !we→type_eq((*tier[i])[p]))
      tier[i]→del(p);
    if (p ≠ NULL)
      tier[i]→del(p);
  }
  delete we;
}

void Chart::print_and_delete_phrase()
{
  Pix p;
  for (p = tier[0]→first(); p ≠ NULL; tier[0]→next(p))
    (*tier[0])[p]→print();

  cout ≪ "\n";
  cout.flush();

  // Delete phrase.
  for (int i = 0; i<num_tiers; i++)
    for (p = tier[i]→first(); p ≠ NULL; tier[i]→del(p));
  wordend = FALSE;
  eophrase = FALSE;
}

int ste_matches(StrTableEntry* ste, X* x)
     // Friend of chart & connectableSegment.
{
  Tier** ap_tier = (Tier **)malloc(chart.num_tiers * sizeof(Tier *));
  int i;
  for (i=0; i<chart.num_tiers; i++)
    ap_tier[i] = chart.tier[i];
  ConnectableSegment *cs;
```

```
  if (ste->fullspec)
    if (ste->fullspec->num_superiors == 1)
      cs = ste->fullspec->superior[0];
    else if (ste->fullspec->num_superiors == 0)
      cs = ste->fullspec;
    else
      error("Uncaught parse error.");
  else {
    cs = (ConnectableSegment *)ste->segment;
    if (cs->num_superiors == 1)
      cs = cs->superior[0];
    else if (cs->num_superiors > 1)
      error("Uncaught parse error.");
  }
  x->is_exact = TRUE;
  return (x->eq(cs, ap_tier, chart.num_tiers));
}

void X::print(int pos=0)
{
  char* output = 0;
  Pix p;
  int times_matched = 0;
  for (p = tbl.first(); p != NULL; tbl.next(p))
    if (tbl[p]->is_phoneme && ste_matches(tbl[p], this))
      if (!times_matched++) {
        if (output) free(output);
        output = (char *)malloc(strlen(tbl[p]->name)+1);
        strcpy(output, tbl[p]->name);
      }
      else if (times_matched == 2)
        cout << "(" << output << "/" << tbl[p]->name;
      else
        cout << "/" << tbl[p]->name;
  if (times_matched > 1)
    cout << ")";
  else if (times_matched == 1)
    cout << output;
}

void Feature::print(int pos=0)
{
  char c;
  switch (value) {
  case -1:
    c = '-'; break;
  case 1:
    c = '+'; break;
  case 2:
    c = '@'; break;
  default:
    c = '\0';
  }
  if (modified)
    cerr << "* ";
  cerr << "[";
  if (c != '\0')
    cerr << c;
  cerr << name << "] (";
```

```
  print_id();
  cerr ≪ ")\n";
  print_aux(pos);
}

void FeatureMatrix::print(int pos=0)
{
  if (modified)
    cerr ≪ "* ";
  cerr ≪ "[";
  char c;
  for (int i=0; i<num_features-1; i++) {
    switch (feature[i]→value) {
    case -1:
      c = '-'; break;
    case 1:
      c = '+'; break;
    case 2:
      c = '@'; break;
    default:
      c = '\0';
    }
    if (c ≠ '\0')
      cerr ≪ c;
    cerr ≪ feature[i]→name ≪ ", ";
  }
  if (num_features) {
    switch (feature[num_features-1]→value) {
    case -1:
      c = '-'; break;
    case 1:
      c = '+'; break;
    case 2:
      c = '@'; break;
    default:
      c = '\0';
    }
    if (c ≠ '\0')
      cerr ≪ c;
    cerr ≪ feature[num_features-1]→name ≪ "]\n";
  }
}

void ClassNode::print(int pos=0) {
  if (modified)
    cerr ≪ "* ";
  cerr ≪ name ≪ " (";
  print_id();
  cerr ≪ ")\n";
  print_aux(pos);
}

void Rule::print(int applied)
{
  cerr ≪ "\nApplying " ≪ name ≪ "...  ";
  if (applied)
    cerr ≪ "Applied.\n\n";
  else
    cerr ≪ "No match.\n\n";
```

```
}

void Nspaces(int n) {
  for (int i=0; i<n; i++)
    cerr << " ";
}

void ConnectableSegment::print_aux(int pos=0) {
  for (int i = 0; i<num_inferiors; i++) {
    Nspaces(pos);
    inferior[i]->print(pos+2);
  }
}

void demo_print_tier(Tier& tr) {
  if (demo)
    for (Pix p = tr.current; p != NULL; tr.next(p)) {
      tr[p]->print_aux();
      if (tr[p]->is_connectable())
        getchar();
    }
}

int Chart::empty() {
  for (int i=0; i<num_tiers; i++)
    if (tier[i]->length() > 2)
      return FALSE;
  return TRUE;
}

void Chart::main_loop(istream& infile)
{
  Pix p, q;
  int i, applied;
  eof = FALSE;
  eophrase = FALSE;
  duple = FALSE;
  infile.clear();
  determine_longest_phoneme();
  StrStack *stk = new StrStack;
  if (!sandhi_rules_exist) {
    do {
      read_word(infile, stk);
      if (!empty()) {
        demo_print_tier(chart[0]);
        for (p = rules.first(); p != NULL; rules.next(p)) {
          applied = apply(rules[p], *this);
          if (demo) rules[p].print(applied);
          if (applied) demo_print_tier(chart[0]);
        }
      }
      print_and_delete_word();
    } while (!eof);
  } else {
    while (!eof) {
      while (!eof && !eophrase) read_word(infile, stk);
      if (!empty()) {
        for (i=0; i<num_tiers; i++)
          tier[i]->current = tier[i]->first();
```

```
        demo_print_tier(chart[0]);
        for (p = rules.first(); p ≠ NULL; rules.next(p)) {
          applied = apply(rules[p], *this);
          if (demo) rules[p].print(applied);
          if (applied) demo_print_tier(chart[0]);
        }
      }
      print_and_delete_phrase();
    }
  }
  delete stk;
}
```

Bibliography

Anderson, S. R. 1988. Morphology as a parsing problem. *Linguistics* 26:521–544.

Antworth, E. L. 1990. *PC-KIMMO: A Two-Level Processor for Morphological Analysis.* Dallas, Texas: Summer Institute of Linguistics, Inc. 1–253. Occasional Publications in Academic Computing.

Barton, G. E., R. C. Berwick, and E. S. Ristad. 1987. *Computational Complexity and Natural Language.* Cambridge, MA: MIT Press.

Boisen, S. 1988. Pro-KIMMO: a Prolog implementation of two-level morphology. In K. Wallace (Ed.), *Morphology as a Computational Problem*, no. 7 in Working Papers in Morphology, Los Angeles. Department of Linguistics, University of California.

Chomsky, N. A., and M. Halle. 1968. *The Sound Pattern of English.* New York, NY: Harper and Row.

Clements, G. N. 1985. The geometry of geometrical features. *Phonology Yearbook* 2:223–52.

Clements, G. N. 1987. Phonological feature representation and the description of intrusive stops. In A. Bosch, B. Need, and E. Schiller (Eds.), *23rd Annual Regional Meeting of the Chicago Linguistics Society. Part Two: Parasession on Autosegmental and Metrical Phonology.* Chicago Linguistics Society.

Dalrymple, M., R. Kaplan, L. Karttunen, M. Kay, A. Kornai, K. Koskenniemi, S. Shaio, and M. Wescoat. 1987. DKIMMO/TWOL: a development environment for morphological analysis. Technical report, Xerox Palo Alto Research Center and Center for the Study of Language and Information, Stanford, CA.

Goldsmith, J. A. 1976a. *Autosegmental Phonology.* PhD thesis, Massachusetts Institute of Technology.

Goldsmith, J. A. 1976b. An overview of autosegmental phonology. *Linguistic Analysis* 2:23–68.

Goldsmith, J. A. 1976c. Tone melodies and the autosegment. In R. K. Herbert, (ed.), *Proceedings of the Sixth Conference on African Linguistics*, Ohio State University Working Papers in Linguistics, no. 20: 135–47.

Goldsmith, J. A. 1981. Subsegmentals in Spanish phonology. In *Linguistic Symposium on Romance Languages*, Washington, D.C. Georgetown University Press.

Goldsmith, J. A. 1990. *Autosegmental and Metrical Phonology.* Cambridge, MA: Basil Blackwell, Inc.

Halle, M. 1993. Feature geometry and feature spreading. Unpublished.

Halle, M., and G. N. Clements. 1983. *Problem Book in Phonology.* Cambridge, MA: MIT Press.

Harris, J. W. 1969. *Spanish Phonology.* Cambridge, MA: MIT Press.

Harris, J. W. 1984. Autosegmental phonology, lexical phonology and Spanish nasals. In M. Aronoff and R. Oehrle (Eds.), *Language Sound Structure: Studies in Phonology Presented to Morris Halle by his Teacher and Students.* Cambridge, MA: MIT Press.

Hertz, S. R. 1982. From text to speech with SRS. *Journal of the Acoustical Society of America* 72:1155–1170.

Hertz, S. R. 1990. The Delta programming language: an integrated approach to nonlinear phonology, phonetics, and speech synthesis. In J. Kingston and M. E. Beckman (Eds.), *Between the Grammar and Physics of Speech*, chapter 13, 215–257. New York, NY: Cambridge University Press.

Karlsson, F. 1985. Computational morphosyntax: a report on research 1981–1984. Technical Report 13, University of Helsinki Department of General Linguistics.

Karttunen, L. 1983. KIMMO: a general morphological processor. *Texas Linguistic Forum* 22:217–228.

Kenstowicz, M. 1994. *Phonology in Generative Grammar.* Cambridge, MA: Basil Blackwell, Inc.

Kernighan, B. W., and D. M. Ritchie. 1988. *The C Programming Language.* AT&T Bell Laboratories, second edition.

Keyser, S. J., and K. N. Stevens. 1993. Feature geometry and the vocal tract. Working paper, MIT Department of Linguistics, Cambridge, Massachusetts.

Kisseberth, C. 1984. Digo tonology. In G. N. Clements and J. A. Goldsmith (Eds.), *Autosegmental Studies in Bantu Tone.* Dordrecht, Holland: Foris.

Koskenniemi, K. 1983a. Two-level morphology: A general computational model for word-form recognition and production. Phd thesis, University of Helsinki, Helsinki, Finland.

Koskenniemi, K. 1983b. Two-level morphology for morphological analysis. In *Proc. of the 8th International Joint Conference on Artificial Intelligence*, 683–5.

Koskenniemi, K. 1984. A general computational model for word-form recognition and production. In *COLING STANFORD: International Conference on Computational Linguistics*, 178–181, Stanford, CA. John von Neumann Society for Computing Sciences.

Koskenniemi, K. 1985. A general two-level computational model for word-form recognition and production. In F. Karlsson (Ed.), *Computational Morphosyntax: a Report on Research 1981–1984*, 1–18. University of Helsinki.

Lozano, M. C. 1978. *Stop and Spirant Alternations in Spanish Phonology.* PhD thesis, Indiana University.

McCarthy, J. J. 1975. Formal problems in Semitic phonology and morphology. Phd thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June.

McCarthy, J. J. 1986. OCP effects: Gemination and antigemination. *Linguistic Inquiry* 17:207–263.

Mohanon, K. P. 1983. The structure of the melody. Massachusetts Institute of Technology.

Oflazer, K. 1993. Two-level description of Turkish morphology. Bilkent University, Turkey.

Pulleyblank, D. 1986. *Tone in Lexical Phonology.* Dordrecht: Reidel.

Sagey, E. 1986. *The Representation of Features and Relations in Nonlinear Phonology.* PhD thesis, Massachusetts Institute of Technology.

Stevens, K. N. 1992. Speech synthesis methods: Homage to Dennis Klatt. In G. Bailly, C. Benoit, and T. R. Sawallis (Eds.), *Talking Machines: Theories, Models and Designs*, 3–6. Amsterdam, Holland: Elsevier Science Publishers.

Stroustrup, B. 1991. *The C++ Programming Language.* AT&T Bell Laboratories, second edition.