

A System for Computational Phonology

by

David McAdams Baggett

B.S./B.A. Computer Science and Linguistics
University of Maryland, College Park
(1992)

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science

at the

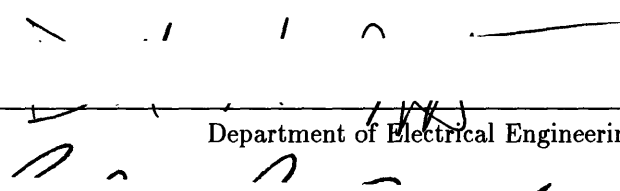
Massachusetts Institute of Technology

January, 1995

Copyright ©1995, by David M. Baggett

The author hereby grants to MIT permission to reproduce and to
distribute copies of this thesis document in whole or in part.

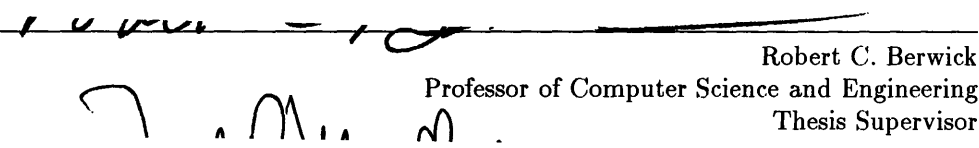
Signature of Author



Department of Electrical Engineering and Computer Science

January 20, 1995

Certified by

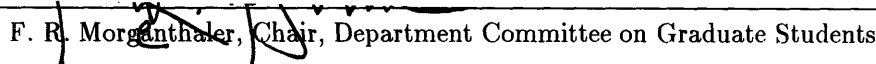


Robert C. Berwick

Professor of Computer Science and Engineering

Thesis Supervisor

Accepted by


F. R. Morgenthau, Chair, Department Committee on Graduate Students

Eng.

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 13 1995

LIBRARIES

Acknowledgments

My advisor Robert Berwick helped in countless ways, from offering an encyclopædic knowledge of several fields to providing general encouragement and specific criticisms. He is a consummate scientist in a field that needs more scientists.

Sharon Inkelas taught me everything I know about phonology. She is an outstanding researcher and an excellent advisor who asked tough questions but never let me lose confidence.

Bill Gasarch and Bill Pugh taught me (almost) everything I know about Computer Science. Both assured me I wasn't stupid and simultaneously showed me how ignorant I was — a valuable experience indeed.

Morris Halle and Michael Kenstowicz were always willing to answer Tough Questions About Phonology, even at a moment's notice. Thanks also to Michael for making drafts of (Kenstowicz 1994) available before publication.

In addition to being an entertaining and considerate officemate, Carl de Marcken is a brilliant researcher who encourages excellence by example. Through our many long conversations I have learned not only many of *The Facts*, but new ways to look at many difficult problems.

Thanks to `gang@ai.mit.edu` and Those Bozos in Columbia for being wonderful friends.

I am so indebted to my parents that I could write another 100 pages about all they have done for me.

And a big L·O·V·E to Catherine Jane, who knows my demons and is not deterred. She is the perfect woman; she is the perfect wife.

This research was funded by the Department of the Army through a National Defense Science and Engineering Graduate Fellowship. Special thanks to Dave Hein for fellowship-related assistance.

Contents

1	Introduction	7
1.1	Generative Grammar and Generative Phonology	7
1.2	Autosegmental Phonology and Feature Geometry	8
1.3	Why this problem is important	9
1.4	Our goals	10
1.5	What makes this problem difficult	11
1.6	Things we are not trying to do	11
1.7	Related Work	12
1.7.1	Kimmo	12
1.7.2	RBT2	12
1.7.3	Delta	13
1.7.4	Bird and Ellison	15
1.7.5	AMAR	15
2	System Description	19
2.1	Lexical Conventions	19
2.1.1	Keywords	19

2.1.2	Tokens	20
2.1.3	#include Directives	20
2.2	Feature Geometry Declaration	20
2.3	Elements	22
2.3.1	Element Basics	22
2.3.2	Elision	24
2.4	Features	24
2.5	Numbering Nodes and Elements	25
2.5.1	Node Updates	25
2.5.2	Node References	27
2.5.3	No Crossing Association Lines Allowed	28
2.6	Phoneme Declarations	28
2.6.1	Append directives	29
2.7	Lexicon Entry Declarations	29
2.8	Rules	30
2.8.1	Match Specification Basics	30
2.8.2	Node Flags	33
2.8.3	Domain Specifications	37
2.8.4	Rule Flags	38
2.8.5	Tier Order Declaration	40
2.8.6	Regular Expression Operators	40
2.8.7	Effects	44

2.8.8	When Effects Cannot Apply	48
2.8.9	Blocks and the CALL Directive	49
2.8.10	The GENERATE Directive	50
3	Examples	51
3.1	The English Plural	51
3.1.1	Theoretical Issues	53
3.1.2	Putting the Rules to Work	54
3.2	Margi Contour Tones	57
3.3	Sudanese Place Assimilation	63
3.4	Japanese Rendaku	67
3.5	Ilokano Reduplication	73
3.6	Register Tone in Bamileke-Dschang	80
4	Conclusions	91
4.1	System Evaluation	91
4.1.1	Coverage	91
4.1.2	Ease of use	92
4.1.3	Flexibility	92
4.1.4	Efficiency	92
4.1.5	Portability	93
4.2	Future Work	93
A	Complete Rulesets for all Examples	94

A.1 Standard Definitions	94
A.2 The English Plural	100
A.3 Margi Contour Tones	102
A.4 Sudanese Place Assimilation	106
A.5 Japanese Rendaku	110
A.6 Ilokano Reduplication	116
A.7 Register Tone in Bamileke-Dschang	121

Chapter 1

Introduction

Almost all current research in phonology is done by hand. Phonologists collect data and analyze a small number of interesting forms with pencil and paper. Not only is this tedious, it is error-prone and depends for its success on the linguist's intuition in picking samples to test. This thesis describes **Pāṇini**¹, a system for automating phonological analyses so that they can be performed on vast data sets and with complete accuracy. Our primary goal for **Pāṇini** is *coverage* — to elegantly support as many real analyses from the phonological literature as possible. To evaluate the system's capabilities, we implement several challenging textbook examples and conclude with an implementation of our own new analysis of some tonal data that have long perplexed the phonological community.

1.1 Generative Grammar and Generative Phonology

Generative grammar is a linguistic theory developed by Noam Chomsky and Morris Halle in the 1950's at MIT. The basic premise of this theory is that the basis for human language capability is a genetic endowment called *Universal Grammar (UG)*. Generative grammarians claim that all languages are based upon UG, and are therefore fundamentally identical — only the lexicon (vocabulary) and a small set of parameters vary from language to language (Chomsky 1986).

The reason this theory is called *generative* grammar is that it is premised on the idea that every utterance is generated by transforming some *underlying representation (UR)* into the correct *surface representation (SR)*. Historically, linguists have assumed that these transformations are ordered sequences of rules.²

Generative phonology is the portion of this overall theory that deals with sound changes — the physical realization of language. Processes that induce the kinds of sound changes phonologists typically study include things like plural formation; compounding; verb inflection; case, number, and gender marking;

¹Named after the 5th century B.C. Sanskrit grammarian.

²Recently there has been a shift away from rules and towards constraints. Proponents of constraint-based approaches claim that constraint-based accounts are easier to learn. We will confine ourselves to rule-based phonology in this thesis.

and even word games like “pig latin”.

Phonologists study these phenomena in an ongoing effort to determine the character of UG with respect to phonology; in other words, to determine just what is “built into” human brains to handle all these sound change processes — how the sounds are represented, what the set operations that can be performed on these representations is, and how a child learns which sequence of operations is correct for each process in his language.

Chomsky and Halle codified the theory of generative phonology in their landmark work *The Sound Pattern of English (SPE)* (Chomsky and Halle 1968). At that time, phonologists viewed the sounds of language (called *segments* or *phonemes*, depending on the role that the sounds play in the language) not as atomic units, but as bundles of binary *features*. Most of these features correspond directly to articulatory gestures; e.g., the feature [voiced], which distinguishes the [-voiced] [t] sound in “tip” from the [+voiced] [d] sound in “dip”, denotes whether or not the vocal cords are stiff or slack when the sound is produced.

Chomsky and Halle combined this conception of linguistically meaningful sounds with a system based on *rewrite rules*. These rules all take the form “rewrite *A* as *B* when it appears in the context *C*”. For example,

$$C \rightarrow [-\text{voiced}] / _ \#$$

specifies that any word-final consonant (the # indicates a word boundary) should be devoiced.

Unfortunately, linguists discovered in the 1970’s that these so-called “SPE-style” rules, although theoretically very powerful, do not allow natural analyses of many phonological phenomena. This realization, which arose out of work on tone languages, prompted the development of *autosegmental phonology*.

1.2 Autosegmental Phonology and Feature Geometry

The tone language phenomena that gave the SPE framework trouble often involved what appeared to be long distance effects, where a change in one part of an utterance affected another part many segments away. Researchers traced this problem to the SPE theory’s treatment of words as *linearly ordered* strings of feature bundles.

The current theory instead arranges the features in a *feature tree*, and views each node in this tree as a *tier* on which elements of that type move about. See Figure 1.1 for a typical view of the feature tree. Figure 1.2 depicts the feature tree for the [d] sound in “dog”.

Imposing a hierarchy on the features changes phonological theory significantly. First, it makes predictions about what combinations of nodes can be manipulated at once. Consider the feature tree in Figure 1.1 again: since the anterior and round features are dominated by different nodes (Labial and Coronal), we expect processes that simultaneously change these two features without also changing all the other

features dominated by the oral place node to be very rare or completely unattested. The data seems to support such predictions; consider, for example, the Sudanese data in §3.3.

Second, a feature tree approach radically alters the notion of *adjacency*. If each node in the feature tree defines its own tier, two segments may be adjacent on a particular tier but far apart on others, because some segments may be unspecified for a particular tier; in other words, they may project no nodes onto the tier. In Figure 1.2, for example, there is no Tongue Root node, because the canonical [d] feature tree does not have a node on the Tongue Root tier. See the Japanese Rendaku data in §3.4 for an analysis that depends on this new interpretation of locality.

Finally, feature tree theory posits links connecting nodes on the tiers together — these are the lines in Figure 1.2. The feature tree describes which nodes can link together; the geometry in Figure 1.1, for example, predicts that nodes of type Tongue Root can only have parents of type Laryngeal, and can have children of types ATR and RTR.

Putting each node on its own tier therefore allows us to view a string of identical nodes on a particular tier as a *single* node with multiple parents. This predicts processes that appear to change long strings of features all at once (in reality by changing the single, multiply-linked feature). John Goldsmith first proposed this idea, called the *Obligatory Contour Principle (OCP)*, to handle tonal phenomena (Goldsmith 1976) — since then it has become a cornerstone of phonological analysis. Giving each feature its own tier potentially extends Goldsmith’s principle to *all* features, not just tonal features.³

1.3 Why this problem is important

Almost all research in phonology is done by hand. Phonologists collect data, select the most interesting forms, and analyze this subset by running the rules that they propose explain the data by hand, checking the results by hand.

Not only is this tedious, it is also inaccurate and error-prone. Phonology is a complex natural phenomenon. Imagine trying to predict the weather, the orbits of the planets, or the behavior of any other complicated system without the aid of machines. Now that technology exists to automate analysis of astronomical and meteorological data, going back to the old hand-checked analyses of thirty years ago is unthinkable. There is no reason that the same kind of automation that has revolutionized so many other scientific endeavors cannot be applied to the problem of phonological analysis.

Given suitable software, we envision a very different approach to phonological research. Whereas now researchers must often duplicate the very time-consuming data collection and transcription tasks of others, on-line databases of utterances in machine-readable form will allow linguists to analyze data from many different languages.

Likewise, where linguists now painstakingly regenerate all their surface forms when they change a single rule in an analysis, the software will be able to do this in a matter of moments. And since computers can perform these tasks many times faster than humans can, such software paves the way for complete

³In practice, it turns out that the OCP does not seem to apply to every tier; research on this topic is ongoing.

phonological descriptions of languages, that can be tested on thousands or even millions of inputs.

Beyond its ability to markedly increase the pace and quality of phonological research, building a computational phonology system requires the author to codify the theory. Automating analyses requires us to take often vague explanations and specify them in complete detail, and to make explicit all the assumptions that practitioners make. In short, it forces us to *formalize* the theory. We feel that this kind of testing is crucial for any complex theory.

Finally, a computational phonology system has direct, practical value as part of a speech generation system. Though we do not develop the idea in this thesis, connecting our system to a speech synthesizer of the kind described by Stevens (1992) could ultimately make machines capable of producing speech indistinguishable from a human's.

1.4 Our goals

Our primary goal is *coverage*. That is, we want to make implementation of as many phonological analyses as possible completely straightforward.

This raises a philosophical point. We want our system not just to support implementations of particular isolated phenomena, but to allow implementation of the *entire phonology* of any given language. If we need only account for a small set of data and a few phonological processes, we are free to make all kinds of ad hoc assumptions that work for the data, but which are not actually true of the language as a whole, and which would prevent proper implementation of other aspects of the language's phonology.

In our account of the English plural in §3.1, for example, we do not need to treat the affricates properly — simply defining them as +continuant atomic segments would be sufficient, and even a system supporting only linear phonology (like RBT2, described in §1.7.2) could handle this sort of analysis. Supporting *some* rules that generate the proper surface forms is not sufficient — we wish to allow direct implementation of the particular analysis that we find in the literature.

A second goal is *ease of use*. Since we hope working linguists will use the system to implement (and test) their analyses on large sets of data, it is important for the notation to be easy for a linguist to understand. In particular, we demand an almost direct correspondence between the syntactic units the program deals with and the linguistic units phonologists talk about. This means that rules that are simple to write down in standard linguistic notation must also be tersely expressible with our system.

We are concerned with *flexibility* as well. The system must not assume too much about the linguistic theory; otherwise it will not be useful once the theory changes. With this in mind, we need to build as few specifics of the theory as possible into our system. This goal is unfortunately at odds with the previous requirement. Consider a general-purpose programming language like C. While it is completely flexible, it is very difficult to use for phonological analyses, because nothing about the theory is built into it — every minute detail of an analysis must be specified in the C program. Our target, then, is to strike a good balance between a system like C that assumes nothing, and a system like AMAR (see below) that assumes too much.

Another important goal is *efficiency*. Since a primary motivation for writing the system is that we wish to be able to test analyses on very large data sets, the program must be able to apply an analysis to an input form in at most a matter of seconds on current hardware.

Finally, *portability* is a concern. We do not want a system that only runs within a particular programming environment or only on certain kinds of hardware.

1.5 What makes this problem difficult

Several things make this problem difficult. First, the theory is still fairly informal. Although basic operations have been formalized, written-out stipulations (in English, not notation) radically change the meanings of many analyses in the literature. (See the Ilokano reduplication analysis in §3.5 for an example of these kinds of textual qualifications to rules.) Consequently, trying to find a rich enough set of primitives we can implement efficiently is difficult.

Furthermore, the theory is still changing rapidly. In the past 25 years phonological theory has been revolutionized twice. A new research direction based upon constraints (Prince and Smolensky 1993) has the potential to radically revise the theory yet again. This rapid progress makes it difficult to get a snapshot of the theory. We suspect, too, that theories are being abandoned before they can be tested adequately, in large part due to the lack of good computer aided phonology systems.

We have also found that implementing a reasonably complete system is considerably more difficult than one might expect. Although the primitive autosegmental operations are fairly easy to implement, the matching procedure that identifies the context in which to apply a rule is quite complex.

1.6 Things we are not trying to do

As we will see in the section on related work, different researchers have built computational phonology systems for rather different reasons. We are not trying to address any of the following issues with this system:

- *Cognitive plausibility*. We do not claim that this system implements what is in people's heads.
- *Learning*. Our system does not infer rules; the user must fully specify analyses.
- *Parsing & speech recognition*. We do not claim that this system is useful for *parsing* — only for generation. We hope that Maxwell's parsing strategy (Maxwell 1994) can be applied to our system, but already recognize some significant difficulties with adapting his ideas to any reasonably complete implementation of autosegmental phonology.

Although these are all interesting problems, they are beyond the scope of this thesis.

1.7 Related Work

Many other researchers have tackled this problem, but the goals of their projects have varied considerably. A few systems, like Kimmo and that of Bird and Ellison, come from an automata framework. In these cases, the designers have set out to force the theory into a very constrained model that can be more easily learned. The Delta system is designed to deal with problems in phonetics as well as phonology. RBT2 is meant for transliteration tasks. AMAR is the only system designed with the goals we have outlined for **Pāṇini**.

1.7.1 Kimmo

The KIMMO system (Koskenniemi 1983) was designed to model morphology, but has been applied to phonological problems. It models language with a finite state transducer — there are no intermediate representations and the “rules” are not ordered.

Kimmo is ideally suited to handling the morphology of languages like Finnish (for which it was built) and Turkish. These languages are purely concatenative — all affixes go onto the ends or beginnings of words. However, many languages have infixation processes, and Kimmo is incapable of handling these. Furthermore, Antworth (1990) and Anderson (1988) point out that Kimmo cannot handle nonlinear representations.

Both of these capabilities are essential in any system intended to support the kinds of rules that phonologists have been writing since the late 1970’s.

1.7.2 RBT2

We developed RBT2 (Rule Based Transliterator II) to handle transliteration between languages. In particular, we were interested in producing IPA (International Phonetic Alphabet) output corresponding to utterances written in English, German, Arabic, and Russian orthography.

RBT2 assumes an SPE-style (Chomsky and Halle 1968) representation, wherein sound segments are viewed as unorganized bundles of features. While this proved sufficient for our task, it is inadequate for many phonological problems, and in particular cannot handle the majority of the analyses in our chapter of examples.

However, RBT2 did have a very natural notation that linguists could adapt to quite easily. We have tried to retain this notation as much as possible in **Pāṇini**. In addition, we have carried RBT2’s regular expression operators over to **Pāṇini** since we found them essential in implementing analyses drawn from the earlier segmental phonology literature.

1.7.3 Delta

Delta (Hertz 1990) is a programming language for dealing with phonetic and phonological phenomena. It is theoretically capable of handling any phonological data since it provides the ability to link in arbitrary C code.⁴

Unfortunately, such power comes at a price — Delta rulesets are very complex and difficult to read, even for a seasoned programmer and linguist. The following Delta code, for example, does what Pāṇini can do in only a few short rules:

```

:: 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

```

⁴Most computer scientists assume (by Church's Thesis) that general-purpose programming languages like C can be used to solve all problems that can be solved by *any* computing device we could build (Hopcroft and Ullman 1979). This says nothing at all about the efficiency or simplicity of the solution, however.

```

:: 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 general, Delta requires considerably more “micromanagement” than any of the other systems we discuss in this section. This results in rules that go on for pages, and a system that is overall very hard to use.

Furthermore, the representation Delta assumes is simply inadequate for contemporary analyses. Hertz gives the following example:

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								

The vertical bars are “sync marks” that both separate nodes from other nodes on the same tier and identify which nodes dominate which others in the tree. Although this looks promising at first — the representation is certainly better than the linear one that RBT2 uses — it has one fatal flaw. The problem is that the representation is only two-dimensional — it does not allow tiers to have multiple parents or children since there are no association lines.

As a result, analyses like the Sudanese place assimilation example in §3.3 are impossible to implement properly — the Delta user must find some ad hoc work-around that relies only upon a 2D representation. So Delta does not meet our goals.

1.7.4 Bird and Ellison

As part of an ongoing project in machine learning of phonology, Bird and Ellison have developed a “phonologist’s workbench” built upon their one-level phonology framework (Bird and Ellison 1992).

They propose that all phonological processes can be implemented with manageably-sized finite automata. If they are correct, the serious learnability problems with present phonological theories will turn out to not be so serious after all — good news indeed.

However, it is not at all obvious that their very constrained model can actually handle all the data. In particular, while they show how to implement some kinds of phonological rules, they have not yet addressed difficult data like the Ilokano reduplication facts we account for in §3.5.

Furthermore, since their research interests are geared more towards machine induction of phonological rules (certainly a noble goal!), they place less emphasis on implementing analyses that appear in the phonological literature. Hence adapting an existing analysis to their notation (in order to test it out on massive amounts of data, for example) is either difficult or impossible, depending on the analysis in question.

1.7.5 AMAR

The only successful system⁵ designed with our goals in mind is AMAR (Albro 1994). This system, like **Pāṇini**, assumes a feature-tree representation and offers a reasonably compact notation for expressing autosegmental rules. In many ways, **Pāṇini** follows naturally from experience with both RBT2 and AMAR.

Within this framework, however, AMAR does have quite a few inadequacies that prevent one from implementing many of our example analyses naturally:

Current theory is hard-wired

Much of the current theory is built into AMAR, which means that system will have trouble supporting new analyses, or analyses designed to test new theories about phonological representations. In particular:

- C and V slots are given special status that cannot be changed.
- The set of boundaries that can be referenced (phrase, word, morpheme) cannot be increased.
- The user is given no control over fundamental operations like the association convention, the OCP, and spreading.
- Feature values are limited to + and –.

⁵As far as we know...

Missing essential features

We have found that AMAR lacks some features necessary for many textbook phonological analyses:

- AMAR lacks “not” links, which are fairly common in the literature. These links specify that a rule should only fire when some node is *not* linked to a node of a particular type (or in some cases, to a single specific node). We crucially rely on this feature in many of our examples.
- There is no way to mark any part of a match specification optional.
- AMAR provides no way to specify the order of precedence for tiers with respect to the position or length of the match.
- It is impossible to implement edge-in association with AMAR, a process that data from Shona (Hewitt and Prince 1989) seem to require.
- Regular expression operators are not supported. We depend on these in several of our analyses.
- There is no way to specify whether or not nodes that have been matched or inserted in a previous cycle can be matched again. Again, this distinction is crucial in many of our analyses.
- Every match specification must list all connections. Using **Pāṇini** domain specifications, one can specify things like “an X slot linked anywhere in the syllable.” AMAR requires the rule author to specify exactly where in the syllable the X attaches (onset, nucleus, or coda).
- AMAR has no equivalent to **Pāṇini**’s initial, medial, and final node flags. Such qualifications are extremely common in the phonological literature.
- Since AMAR cannot copy phonological material, reduplication phenomena like the Ilokano data in §3.5 are nearly impossible to implement.
- All AMAR rules are iterative; there is no way to write a rule that applies exactly once without appealing to boundaries.
- The notation is somewhat cumbersome; connections between nodes must be listed separately, and the system does not fill in intervening tiers like **Pāṇini** does.
- AMAR’s input system is too limited; it does not allow the user to specify a feature tree as input. This has serious repercussions for analyses with complex underlying representations (e.g., the Dschang data in §3.6).

In **Pāṇini** we correct all these shortcomings while maintaining AMAR’s simplicity and ease of use.

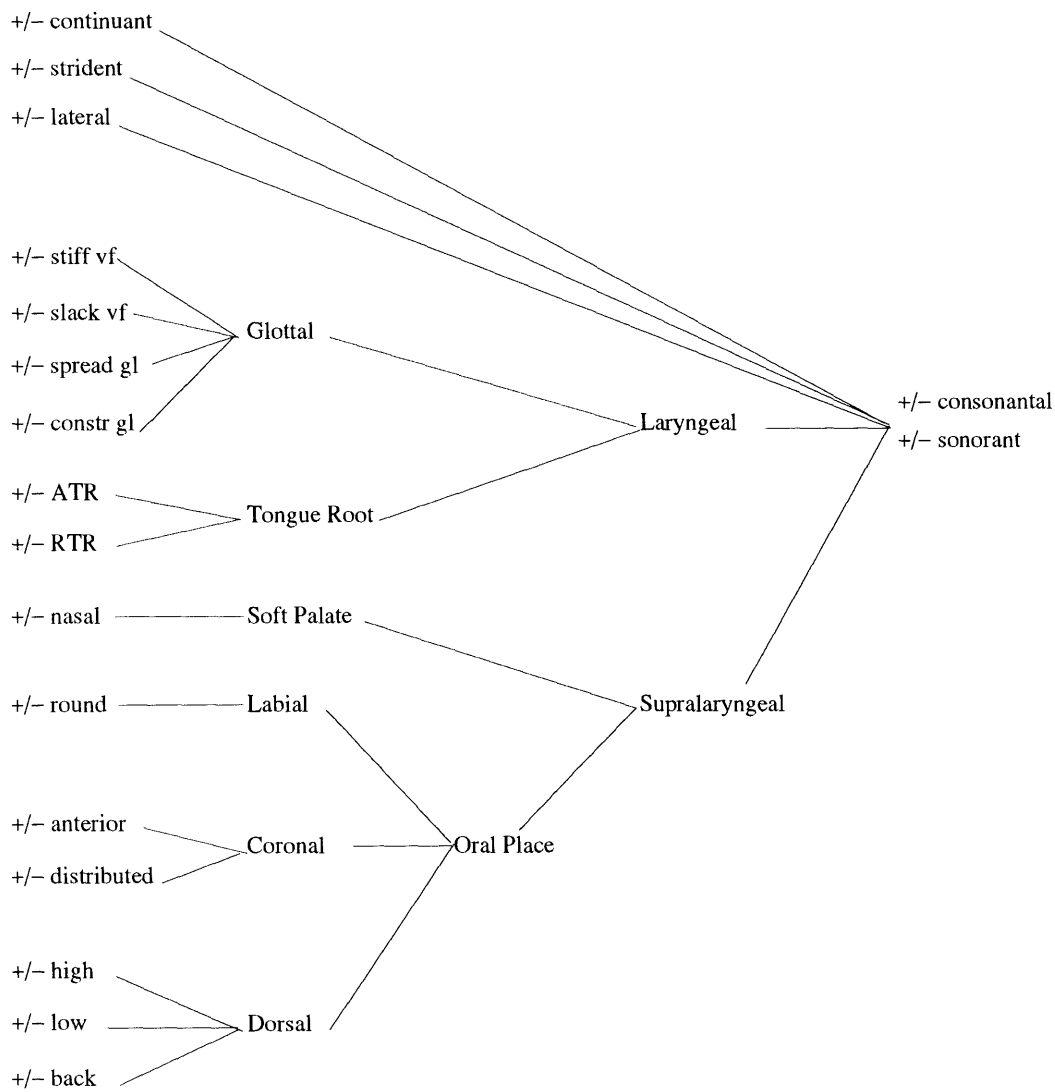


Figure 1.1: Feature Tree

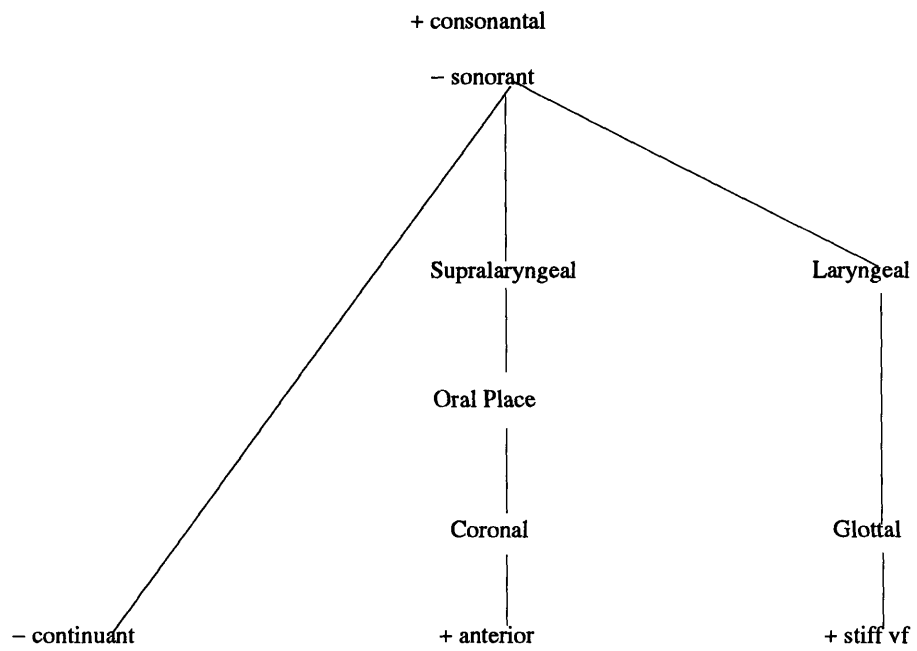


Figure 1.2: Feature Tree for the Phoneme [d]

Chapter 2

System Description

In this chapter we describe the features of **Pāṇini** and show in detail how to use the system.

Pāṇini takes an input file describing the feature geometry, phonemes, and rules for a language process and applies the rules to the list of input forms the user specifies. We will discuss each of these components in detail.

2.1 Lexical Conventions

Like most general-purpose programming languages, **Pāṇini** tokenizes all input files. Whitespace characters like spaces, tabs, and newlines are ignored except where they separate tokens. The input is case-significant except with respect to keywords. In other words, case may distinguish user-assigned names, but not keywords.

The tokens `/*` and `*/` begin and end a comment, respectively. All text inside a comment, including newlines, is ignored, though comments may not be nested.

Pāṇini provides an alternative commenting style as well: it ignores all text from a `//` token until the end of the line. The two kinds of comments can be used interchangeably.

A comment may appear anywhere except within a token or directive.

2.1.1 Keywords

Pāṇini reserves the following keywords:

after append at before block call change childless choice copy
 decrement delete delink dlink edge effect effects erase exact final
 first from generate geometry in increment initial insert inserted
 iterative last leftmost lexicon link linked match matched matches
 medial middle no nonfinal noninitial nop optional order parse
 phoneme rematchable rightmost rule shared stray to trace under
 unlink unlinked warnings within

These tokens (regardless of the case of the letters) may not be used to name user-defined units like phonemes, feature geometry nodes, or lexicon entries.

2.1.2 Tokens

Feature geometry node names are strings of letters; no numbers are allowed. Feature and phoneme names are strings of alphanumerics and underscores, but may not begin with an underscore. Double-quoted strings may contain any characters.

2.1.3 #include Directives

The directive `#include` may appear at the beginning of a line. It must be followed by a double-quoted string, in which case it causes **Pāṇini** to insert the named file verbatim in place of the `#include` directive. For example,

```
#include "std.ph"
```

directs **Pāṇini** to insert the text in the file `std.ph` in place of the directive itself.

Included files may include `#include` directives, to a maximum nesting depth of 16 files.

2.2 Feature Geometry Declaration

In keeping with the current theory, **Pāṇini** requires the user to describe the feature tree that the subsequent rules and data assume. This geometry declaration must be the first thing in every ruleset. An example follows:

```
geometry {
  word -> morpheme,
  morpheme -> sigma,
  sigma -> stress [5 4 3 2 1],
```

```

sigma -> onset,
sigma -> rhyme,
rhyme -> nucleus,
rhyme -> coda,
onset -> X,
nucleus -> X,
coda -> X,
X -> consonantal [- +],
consonantal -> sonorant [- +],
sonorant -> continuant [- +],
sonorant -> strident [- +],
sonorant -> lateral [- +],
sonorant -> laryngeal,
sonorant -> pharyngeal,
laryngeal -> glottal,
laryngeal -> tongue_root,
glottal -> voiced [- +],
glottal -> slack_vf [- +],
glottal -> spread_gl [- +],
glottal -> constr_gl [- +],
tongue_root -> atr [- +],
tongue_root -> rtr [- +],
sonorant -> supralaryngeal,
supralaryngeal -> soft_palette,
supralaryngeal -> oral_place,
soft_palette -> nasal [- +],
oral_place -> labial,
oral_place -> coronal,
oral_place -> dorsal,
labial -> round [- +],
coronal -> anterior [- +],
coronal -> distributed [- +],
dorsal -> high [- +],
dorsal -> low [- +],
dorsal -> back [- +],
X -> T,
T -> register [0 up down],
T -> modal [L M H],
}

```

This is the **Pāṇini** feature geometry we will use for most of our examples, and corresponds closely to the Halle-Sagey model (Halle 1992) shown in Figure 1.1.

Each line in the definition above establishes new nodes or connections in the feature geometry; for example, the first line tells **Pāṇini** that the tree is rooted at the word tier, and that this tier directly dominates the morpheme tier; i.e., that nodes on the word tier take children of type morpheme, and that nodes on the morpheme likewise link to parents of type word.

Note that we list the parent before the child; **Pāṇini** requires the user to declare each parent before any of its children are declared. This means that, for example, interchanging the first two lines of the declaration is illegal — **Pāṇini** needs to know all of morpheme’s parents before it can deal with its

children. Practically speaking, this just means that the feature geometry has to be defined from top to bottom; technically it means that the nodes have to appear in “topological order”.¹

Pāṇini pays no attention to the node names, except insofar as requiring nodes to be uniquely named. In particular, **Pāṇini** assumes nothing about word or syllable structure — it treats nodes named “morpheme” just as it treats nodes named “voiced”. This means that **Pāṇini** is flexible, and that testing out different conceptions of the geometry (moraic versus X-slot syllable structure, for example) is simple — nothing is hard-wired.

Note that square-bracketed lists follow some nodes; these declare feature values. Any node may have an associated list of feature values; most will take + and - if they take features at all. However, features can be given arbitrary names, and users may assign as many features to a node as desired. Note, however, that no feature can have the same name as any node.

A few syntactic notes: The keyword “geometry” and the opening and closing brace are required. Node declarations must contain exactly two node names, separated by the → sequence. A comma must separate consecutive declarations, and may optionally follow the last declaration before the closing brace.

Henceforth we will refer to the declared feature geometry as the “defining geometry” for a ruleset to distinguish it from the geometry of a particular input segment.

2.3 Elements

Now we need some way to create instances of the defining geometry; i.e., some way to write our equivalent of [d] in segmental parlance. Using the same syntax we used for the geometry declaration would be cumbersome indeed; instead **Pāṇini** allows a linearized and abbreviated syntax. We call an instance of the defining geometry declared this way an *element*.

An element can be as large as an entire word (in which case it will be rooted at a word node), or as small as a single feature geometry node.²

2.3.1 Element Basics

Roughly speaking, element declarations are simply lists of node names enclosed in square brackets. For example,

```
[word morpheme sigma]
```

¹We will revisit this notion when we discuss finding the leftmost, longest match in §2.8.1. For now the practical definition should suffice.

²Such singleton nodes, called “floating” units in current linguistic parlance, play a crucial role in many tonal phenomena, as we will see in the analysis of Margi tone in § sec:margi.

declares an element consisting of a word node with a single child of type morpheme, itself with a single child of type sigma. We could give this element an onset, nucleus, and coda by adding these nodes onto our declaration:

```
[word morpheme sigma onset nucleus coda]
```

Note that both these declarations instantiate elements with no segmental material; i.e., no voicing node, no nasal node, etc. To put an X slot in the nucleus, we'd declare the element like so:

```
[word morpheme sigma onset nucleus X coda]
```

This illustrates a key point: we must order the nodes from top to bottom, just as we did with the geometry declaration. In particular,

```
[word morpheme sigma onset X nucleus coda]
```

puts our X slot in the onset instead of the coda, and

```
[morpheme sigma onset nucleus X coda word]
```

declares a bizarre element consisting of two root nodes: one of type morpheme, containing the sigma, onset, nucleus, X, and coda; and the other a bare word node.

Specifically, each node links up with the first previous node that can take that kind of child. **Pāṇini** searches backwards through the element declaration to find the attachment point. For example, to connect the nucleus node in

```
[sigma onset X nucleus coda]
```

Pāṇini looks back through the list, skipping over the X since, according to the defining geometry, nodes of type X cannot take children of type nucleus. Looking back, **Pāṇini** finds the sigma node next, and makes the nucleus node its child.

Technically speaking, nodes must be ordered according to an *inorder traversal* of the target element's feature geometry.³

³An inorder traversal is one that applies the following rule recursively: visit the parent first, then visit the children in order from left to right.

2.3.2 Elision

Having to list the morpheme node in

```
[word morpheme sigma onset X nucleus coda]
```

is a bit tedious, since our defining geometry specifies that only nodes of type morpheme can possibly link above to words and below to sigmas. Fortunately, **Pāṇini** permits nodes it can infer from the defining geometry to be omitted. This makes

```
[word onset X nucleus coda]
```

equivalent to the previous declaration: since nodes of type onset, nucleus, and coda can only link to nodes of type sigma, **Pāṇini** fills in the sigma automatically. By the same token, **Pāṇini** infers the existence of the morpheme node.

2.4 Features

Feature values can be assigned in several ways. The most common method applies only to the features + and –; these can be prepended to any node in an element list:

```
[+voiced -constr_g1]
```

Other features must follow the node they modify, separated from the node name by a colon:⁴

```
[modal:H register:down]
[stress:4]
```

Finally, if a feature only applies to one type of node, the feature name may be used in place of the node name. In our defining geometry, for example, the only nodes that take the feature H are modal nodes, so we can write:

```
[H]
```

instead of

```
[modal:H]
```

⁴This syntax can be used with + and – as well.

2.5 Numbering Nodes and Elements

In order to declare nodes with multiple parents, to reference nodes in other elements, and to declare several nodes of the same type within a single element, we need to number nodes. Any nodes in an element can be numbered, but all nodes of a given type within an element must be given different numbers. Numbering a node involves following its name and optional feature specification with # and an integral number, as in the following examples:

```
[X#423 T#12 modal:H#1 +voiced#0]
```

We will call these element-internal numbers *local ID's*.

Entire elements can be numbered as well:

```
[X T H]#1
```

These are called *element* or *global ID's*. This allows us to refer to nodes in other elements by specifying both local and global ID's, as follows:

```
[onset X#2:1] [X#1]#2
```

Here we've linked the onset in the first (non-numbered) element to the X slot with local ID 1 in element 2. We will see why this is useful when we discuss the matcher in §2.8.1.

2.5.1 Node Updates

How **Pāṇini** interprets a node in an element list depends on what precedes the node in the element declaration. The first node of a particular type in an element always declares a *new* node in the instantiated geometry; this is the case for all the non-numbered nodes in the preceding examples.

However, **Pāṇini** treats a node with no local ID that comes after a node of the same type in the same element as an *update* to the earlier node. E.g.,

```
[X +voiced -voiced]
```

is equivalent to

```
[X -voiced]
```

since **Pāṇini** replaces the earlier node with the second. Note that this can only affect the features, and if the features match the second node declaration has no effect; i.e.,

```
[X -voiced -voiced]
```

is equivalent to

```
[X -voiced]
```

However, if the second declaration has no features, any features in the earlier declaration are removed;

```
[X -voiced voiced]
```

is equivalent to

```
[X voiced]
```

In general, the update declaration completely replaces the earlier declaration.

An obvious question now, is “what if we want to have two of the same kind of node in an element?” This is what local ID’s are for;

```
[X +voiced#1 -voiced#2]
```

declares an element with two voiced nodes.⁵

Updates provide an easy way to instantiate an element that is very similar to an element we have already declared. For example, given that we have defined a phoneme [d]⁶, we can write

```
[d -voiced]
```

to get a [t] element by changing the [d]’s voiced node from + to –.

Pāṇini places no restrictions on how many nodes we can declare; if we really needed a thousand voiced nodes, we could declare them (giving each its own unique ID) and **Pāṇini** would happily instantiate the gigantic element.

⁵Note that both will be linked to the single Glottal node that **Pāṇini** infers must be present.

⁶We describe how to define phonemes in §2.6.

2.5.2 Node References

We now have an element declaration syntax with which we can define any feature tree. Technically speaking, however, not all phonological units are trees. We need to be able to declare arbitrary *directed acyclic graphs (DAGs)*⁷.

Whereas every node in a tree has at most one parent, DAG nodes can have multiple parents. This is one instance where we need *node references*. A node reference is a node with a local and (optionally) global ID specified. When **Pāṇini** sees such a node, it attaches the referenced node in the element right where it would have attached a new node of the same type given a non-numbered node. For example,

```
[X T#1 H#1 T#2 H#1]
```

declares an element with an X slot linked to two different T node children, both of which are linked to the same high tone. The first **H#1** prompts **Pāṇini** to create a new modal H node; the second H node, since it has the same local ID, simply references the earlier node.

We can also reference nodes in different elements, like so:

```
[X T H#1]#0 [X T H#0:1]
```

Here we have defined two elements. The first has an X slot with a tonal root node (T) child linked to a high tone with local ID 1 (**H#1**). The second element also has an X slot and tonal root node, but its tonal root node is linked to element 0's high tone.

Such extra-element, or *global* references are only useful in match specifications, where consecutive elements are considered to be consecutive in the input stream. We describe match specifications in detail in §2.8.1.

Note that all of a node's children must be specified locally in that node's element. This means that

```
[X T#2:1 H] [X T#1]#2
```

is not legal — instead we must write

```
[X T#2:1] [X T#1 H]#2
```

⁷Directed because we need to pay attention to parent/child relationships, and acyclic because we prohibit any node from having a path back to itself.

2.5.3 No Crossing Association Lines Allowed

A basic restriction current phonological theory places on representations is that association lines (the lines connecting nodes together) may not cross. The motivation for this is that we want moving left to right in a representation to correspond to moving forward in time (in keeping with the fact that linguistic utterances are produced serially and not in parallel). If association lines cross, the order of the phonological units is indeterminate. Consider the following element:

```
[X T#1 H#1 L#2 T#2 H#1 L#2]
```

The lines from T#2 to H#1 and from T#1 to L#2 cross, and hence these tonal nodes are both before and after each other — a violation. Contrast this with

```
[X T#1 H#1 T#2 H#1 L#2]
```

where one of the offending lines has been removed; now we have a legal element.

Pāṇini does not strictly prohibit crossing association lines, but the matcher will not always operate properly on elements with crossing association lines. Consequently, **Pāṇini** warns the user any time it determines that association lines cross during a derivation.

2.6 Phoneme Declarations

Every language has a *phonemic inventory* — a subset of all possible sounds that the language manipulates with phonological rules. Since our inputs will use these sounds heavily, we would like to be able to give them symbolic names.

Pāṇini allows us to define as many phonemes as we want, with declarations like the following:

```
phoneme t [ X +consonantal -sonorant -continuant +anterior -voiced ]
```

Phoneme declarations consist of the **phoneme** keyword, a unique phoneme name (which must not conflict with any node or feature name), and an element declaration. Nodes can be numbered, but only with local ID's — global ID's are not allowed, and the element's global ID will be ignored.

Once we have defined a phoneme, we can use it in element declarations in place of typing the element's node list verbatim:

```
[t +voiced]
```

corresponds to

```
[ X +consonantal -sonorant -continuant +anterior -voiced +voiced ]
```

which is equivalent to

```
[ X +consonantal -sonorant -continuant +anterior +voiced ]
```

Phoneme names can appear in other phoneme declarations as well; e.g.,

```
phoneme d [t +voiced]
```

says that a voiced [t] is a [d]. In general, phoneme names can be used anywhere node names can be used, but note that phoneme names cannot be assigned numbers; hence all phoneme names declare new material and never reference existing material.

2.6.1 Append directives

Once a phoneme has been declared, we can modify it with an **append** directive; this tells **Pāṇini** to tack the material listed in the append directive onto the end of the named phoneme. For example,

```
phoneme d [ X +consonantal -sonorant -continuant +anterior ]
append d [ +voiced ]
```

directs **Pāṇini** to add +voiced to the end of phoneme [d]’s declaration. Note that this will not change any phonemes or lexicon entries defined in terms of [d]; in other words, **Pāṇini** does *not* re-evaluate all the phoneme definitions after an **append**.

We generally use **append** to update phonemes defined in some standard include file that we wish to base a whole set of rulesets on.

2.7 Lexicon Entry Declarations

Lexicon declarations are just like phoneme declarations, but are intended to deal with the next higher level of phonological grouping — morphemes and words.

The declaration syntax is quite similar to the phoneme declaration syntax:

```
lexicon "dog" [word sigma onset d nucleus aw coda g]
```

enters a word named “dog” into the lexicon with a single syllable and three phonemes (assuming that phonemes [d], [ɔ], and [g] have been declared).

Like phoneme names, lexicon names can appear anywhere node names can. However, lexicon names must be enclosed in double-quotes.⁸

Again, **Pāṇini** imposes no limit on the number of lexicon entries.

2.8 Rules

Once we have declared the feature geometry and all the phonemes, we can specify the rules we want **Pāṇini** to apply, in the order we want it to apply them. A **Pāṇini** consists of two parts: 1) a match specification describing all contexts in which the rule is to apply, and 2) a list of actions to be taken if **Pāṇini** finds a match. We call these actions *effects*.

The rule syntax is:

```
rule "name":
  <rule flags>
  match: <match specification>
  effects: <list of effects>
;
```

where “name” is an arbitrary double-quoted name for the rule, <rule flags> is an optional list of rule flags, described in §2.8.4, <match specification> is a match specification, described in §2.8.1, and <list of effects> is a list of effects, described in §2.8.7.

The keywords **effect** and **matches** can be used in place of **effects** and **match**, respectively.

2.8.1 Match Specification Basics

We generally want a rule to apply only to certain inputs, or to certain parts an input string. And when we do get a match, we want our effects to operate on particular matched nodes. The match specification syntax allows us to tell **Pāṇini** what input sequence to match, and what elements our effects will refer to. An example match specification follows.

```
match: [X H]
```

⁸As a consequence, they can also have spaces in them, unlike phoneme names.

This directs **Pāṇini** to apply the rule if the input contains an X slot linked to a high tone. The square-bracketed list is an element, as described in §2.3. In fact, the match specification is simply a list of elements. E.g.,

```
match: [X H] [X L]
```

will match an X slot linked to a high tone followed by another X slot linked to a low tone.

Here is where global ID's come into play.

```
match: [X H#1]#0 [X H#0:1]
```

will match an X slot linked to a high tone followed by another X slot linked to the *same* high tone.

Adjacency

Consecutive elements will match only consecutive input nodes. For example,

```
match: [X H] [X L]
```

specifies that:

- no X node may intervene between the two X slots,
- no modal node may intervene between the H and L,
- no T node may intervene between the two elements.

Note the last restriction in particular, recalling that **Pāṇini** infers intervening nodes in elements. Since modal nodes do not link directly to X slots — tonal root nodes (T) must intervene — these elements really reference material on the T tier as well, and hence prohibit any tonal root nodes from intervening between the two elements.

This notion of adjacency applies over strings of elements. For example,

```
match: [X +voiced] [X T H] [X -voiced]
```

stipulates not only that the three X slots are adjacent (with no X slots intervening), but also that the two voiced nodes are adjacent. Hence this match specification will *not* match input corresponding to

[X +voiced] [X T H voiced] [X -voiced]

because the middle element projects a node onto the voiced tier, and this voiced node is not mentioned in the match specification.

Superset Matching

By default a match element will match input elements that have all the nodes and links established by the match element *and any other nodes and links*.⁹

This means that

match: [X T]

will match all of the following input elements, as well as many others:

[X T]
 [X T H]
 [X T +voiced]
 [X +consonantal -sonorant -continuant +anterior +voiced T H]
 [X T#1 H#1 T#2 H#2]

The **exact** node flag forces an exact, rather than a superset match — see §2.8.2.

The Leftmost Match

A match specification will often match an input string in many different places — e.g., [X H] [X H] will match the string [X H] [X H] [X H] at two different offsets.

Since we want the matcher’s behavior to be completely predictable, **Pāṇini** finds the *leftmost* match by default. (The **rightmost** rule flag, discussed in §2.8.4, overrides this behavior.)

Given our nonlinear representation, “leftmost” could be interpreted many ways. The method **Pāṇini** uses to decide which of two possible matches is the leftmost match is as follows. **Pāṇini** starts at the highest node in the defining geometry and compares the leftmost nodes on this tier in each of the candidate matches. If one match is leftmost with respect to this tier, **Pāṇini** considers it the leftmost overall; otherwise it does the same check on the next lower tier. If some candidate match has no node at all on a particular tier, that tier is skipped entirely.

⁹This is, of course, subject to the adjacency requirements mentioned in the previous section.

An obvious question is “what do we do for tiers with multiple children?” The answer is that the results are not predictable by default — the matcher is only guaranteed to enforce the rule that

- If tier A dominates tier B in the defining geometry, tier A will be checked before tier B.

This rule establishes a so-called *topological ordering* on the defining geometry.

When it is necessary, we can specify the exact order that tiers will be considered — see §2.8.5.

The Longest Match

In the event that candidate matches are equally leftmost according to the definition above, **Pāṇini** favors the longest match. Here again, “longest” is too vague for our purposes; in fact, **Pāṇini** compares the length of each match string on successive tiers, just as it compares ordering on successive tiers to find the leftmost match.

The length of a sequence of nodes on a particular tier is just the number of nodes on that tier.

2.8.2 Node Flags

Phonological rules often refer to initial, medial, and final units; for example, syllable-final consonants are devoiced in German, and a word-initial [p] is aspirated in English.

We can place these restrictions on nodes with *node flags*. Node flags directly precede the node they modify; the order of the node flags is not significant.

Node flags actually modify the *link* between the following node and the preceding parent in the element. This means that the *initial* flag in

```
match: [onset X#1 nucleus initial X#1]
```

applies to the link between the X slot and the nucleus, not the onset. In other words, this element will match an onset and nucleus both linked to the same X slot, with the additional requirement that the X slot be the leftmost X slot linked to the nucleus.¹⁰

With the exception of *unlinked* and *~*, node flags only apply to match elements, not input elements. (They are simply ignored in input elements.)

¹⁰In this case, the *initial* flag is probably redundant, since in any standard analysis an X slot in the onset is going to be the first X slot in the nucleus if it's linked to the nucleus at all.

Position flags

The **initial** node flag specifies that the node must be the leftmost child; likewise, **final** requires the node to be the rightmost child. The **medial** flag permits only nodes that are neither initial nor final; similarly, **noninitial** and **nonfinal** work as expected.

Note that we can write **first** instead of **initial**, **last** instead of **final**, and **middle** instead of **medial**.

Linkage flags

A match node may be marked **unlinked**, in which case it will only match a node with no parents. Similarly, the **childless** flag tells the matcher to consider only nodes with no children.

The **unlinked** flag can be applied to input nodes as well; in this case the flag specifies that the node is not linked to its parent, but is still within the domain defined by the parent. We typically use this mechanism to put floating nodes in lexicon entries; see the Margi analysis in §3.2 for an example of this.

The ! flag

The ! node flag indicates that no node of the given type can be linked to the parent node. E.g.,

```
match: [X !T]
```

specifies that the X slot cannot be linked to any tonal root node. Features are significant;

```
match: [X !+voiced]
```

indicates that the X slot can be linked to a voiced node, but only if the node does not have the + feature.

Local and global ID's change the interpretation of the ! flag;

```
match: [X !voiced#2:1]
```

specifies that the X slot must not be linked to the voiced node with local ID 1 in element 2. It may be linked to *other* voiced nodes, however.

The ~ flag

The ~ flag is only useful in phoneme, lexicon, and input elements; it is ignored in match specifications. Unlike other node flags, this flag cannot be combined with any other node flags; if it is set all other node flags are ignored.

The ~ flag undefines a previously declared node. This is mainly useful when dealing with included files. For example, we keep our standard defining geometry and phoneme definitions the file `std.ph` for convenience, and include this file in our rulesets with `#include` directives. We can then make minor changes to the phoneme definitions using ~ and the `append` directive (described in §2.6.1), like so:

```
append phoneme i [ ~voiced ]
```

This example removes the voiced node from the [i] phoneme, or “underspecifies [i] for voicing” in linguistic terms.

Note that this flag may not be applied to nodes with global ID’s.

The optional flag

The `optional` flag indicates that a node need not appear in the input, but that if it does it is to be matched so that an effect can be applied to it. See the analysis of Japanese Rendaku in §3.4 for a crucial use of this flag.

The exact flag

The `exact` flag stipulates that the node may not have any parents or children not explicitly mentioned in the match specification.

```
match: [X T H]
```

will match [X T H L] and [X#1 T#1 H X#2 T#1] while

```
match: [X exact T H]
```

will match neither of these.¹¹

¹¹The first element fails because the T node has an extra child; the second fails because it has an extra parent.

The shared flag

By default, the matcher requires consecutive match nodes to match consecutive input nodes. The *shared* flag relaxes this restriction, and allows a node to match an input node that has already been matched by the preceding match node. For example,

```
match: [X H] [X H]
```

will match [X H] [X H] but not [X H#1 H#2] — the second input string fails to match because it has only one X slot. However,

```
match: [X H] [shared X H]
```

will match both these input strings, because the X can be shared with the previous match node of type X.

Given a choice between a match that uses sharing and one that does not, the matcher prefers the match that uses sharing. This means that our example will choose [X H#1 H#2] over [X H] [X H].

The rematchable flag

As we will see in §2.8.4, rules may be *iterative*, meaning that they apply again and again to the same input string until they fail to match. Unless we tell it otherwise, the matcher will not consider input nodes that have been matched in a previous iteration of the same rule on the same input string. The **rematchable** flag tells the matcher to allow the flagged node to be matched again and again if possible. To understand the difference, consider

```
match: [X H]
```

This will match a sequence of [X H] elements, which is usually what we want. However, it will fail to match both high tones in the element [X H#1 H#2] because the matcher will refuse to match the same X slot twice. In cases where we need the other behavior, we can mark the X as **rematchable**:

```
match: [rematchable X H]
```

Note, however, that the matcher will *never* match exactly the same set of input nodes twice; at least one node must differ, even if all nodes are marked **rematchable**. This prevents the matcher from getting stuck in endless loops while searching for a match.

Nevertheless, it is fairly easy to send the matcher into an infinite loop with the `rematchable` keyword, and **Pāṇini** will not detect this condition — it will blindly run forever. So the `rematchable` node flag must be used with care.

The `match matched` rule flag marks all nodes `rematchable`. We discuss this in §2.8.4.

2.8.3 Domain Specifications

Phonological rules often apply only within certain domains. The Japanese Rendaku analysis in §3.4, for example, looks at voicing within a particular morpheme and not beyond it. We can restrict matches to particular domains in **Pāṇini** with *domain specifications*.

To give any match node a domain specification, we follow the node whose domain we wish to limit by `linked within` and a node reference. For example,

```
match: [sigma X linked within sigma]
```

stipulates that the X slot must be part of the referenced syllable. This eliminates the need to specify the exact geometry of the elements to be matched; instead of specifying that the X slot must be linked to the onset, nucleus, or coda, we simply require it to be linked somewhere within the syllable.

The node reference can be global as well;

```
match: [sigma X linked within sigma#1]#0 [X linked within sigma#0:1]
```

will only match an X slot that is shared by two syllables.

A single node can have multiple domain specifications; for example,

```
match: [word sigma X linked within sigma X linked within word#2:1] [word#1]#2
```

will match an X slot that is linked within a syllable in one word and also linked in the following word.

The `linked` keyword is optional; if it is omitted the node does not have to be linked within the dominator — it merely has to be within the dominator’s scope. We establish the scope of unlinked nodes in input elements with the `unlinked` node flag; this flag indicates that the node is not linked to its parent, but is within the domain of the parent and of every node the parent is linked to.¹²

Note that domain-specified nodes can take node flags. In this case, the flag modifies the relationship with the dominator in the obvious way:

¹²At present, there is no direct way to tell **Pāṇini** to assign an unlinked node to another domain. A workaround is to link the node and then immediately unlink it; this will establish the ephemerally linked parent as the new dominator.

match: [word sigma initial X linked within sigma]

will match the first X linked anywhere within a syllable.

2.8.4 Rule Flags

Whereas node flags provide very fine control over the matcher's behavior, rule flags make broader adjustments that affect every node in the match specification.

The iterative flag

Unless we tell it otherwise, the matcher will apply a rule exactly once to the input form — this means that we can only modify the leftmost or rightmost match in the input string.

The **iterative** rule flag specifies that **Pāṇini** is to keep applying the rule to the input until it fails to find a match. Each iteration, or *cycle* will apply to the leftmost (or rightmost) match, with the stipulation that nodes that have been matched in a previous cycle cannot be matched again. This rule produces the behavior we generally want — to search over the string, changing all portions of the input that match.

The rightmost flag

The matcher finds the leftmost match unless we set the **rightmost** flag, in which case it finds the rightmost match. We can combine this flag with **iterative** to search over the input from right to left.

While it may seem at first like **rightmost iterative** and **iterative** will produce the same end result, there are many analyses in the literature that depend on right-to-left searching. In fact, data from languages like Hausa (Newman 1986) suggests that this may parameterize a proposed Universal Grammar operation called the *Universal Association Convention*.

The edge in flag

Data from Shona (Hewitt and Prince 1989) and Bambara (Rialland and Badjimé 1989) argue for *edge-in* matching, where each cycle alternates between looking for the leftmost and rightmost match.

The **edge in** rule flag enables this behavior. If the rule is also marked **rightmost**, the first match will be a rightmost match, the second will be a leftmost match, and so on. If the **rightmost** flag is omitted, the first match will be a leftmost match, the second will be a rightmost, and so on. In other words, the rightmost/leftmost distinction determines which side of the input the iteration will begin with.

The `edge in` flag has no effect unless it is paired with the `iterative` flag.

The `match matched` and `match inserted` flags

As we saw in the discussion of the `rematchable` node flag in §2.8.2, we sometimes need to permit the matcher to match an input node that has already been matched in a previous cycle. An example of this is the implementation of the OCP in the Margi analysis in §3.2. The `match matched` rule flag applies the `rematchable` node flag to every node in the match specification.

Similarly, the `match inserted` flag permits the matcher to consider input nodes that have been inserted in a previous cycle.

Note that even if both these flags are on, the matcher will never match exactly the same set of input nodes twice; it always requires some input node to change — in other words, some tier must be advanced.

The `no warnings` flag

When we discuss effects in §2.8.7, we will see that some effects will fail to apply because the nodes they apply to fail to match input nodes. When this happens, **Pāṇini** will print a warning. The `no warnings` flag prevents **Pāṇini** from print these warnings for a given rule; this is useful when we design a rule in which we know that not all of the effect will ever apply. We will see how this can happen when we deal with regular expression operators in §2.8.6.

The `trace` flag

Since **Pāṇini** rules can be quite complex, it is often useful to see exactly what the matcher is matching at each stage in the derivation. The `trace` flag enables code to print the result of each operation **Pāṇini** performs.

The `optional` flag

Many phonological processes apply optionally or only in certain cases; e.g., the English syllable liason process that changes [ði:z·aɪ·r·old·ɛgz] (‘These are old eggs’) into [ði:·zaɪ·rɒl·dɛgz] only occurs during rapid speech (Giegerich 1992).

The `optional` flag marks rules that are not mandatory. **Pāṇini** currently ignores this flag.

2.8.5 Tier Order Declaration

In the discussion of the leftmost and longest match in §2.8.1 and §2.8.1, we saw that the matcher compares candidate matches tier by tier, starting at the top of the defining geometry.

We occasionally need to be more precise about the order the matcher is to visit the tiers. To do this, we can simply list the tiers in the rule flags section. For example, the declaration

```
rule "example":
  X T H
  match: ...
;
```

will prompt **Pāṇini** to compare candidate matches first with respect to the X tier, then with respect to the tonal root node (T) tier, and finally on the modal tier. It is important to realize that **Pāṇini** will not consider *any other tiers* — it will only look at the ones we list if we list any at all.

Notice the use of H in the example above. As in element declarations, we can substitute a feature name for a tier name if the feature only associates with a single tier.

Although the order of the tiers in the tier order declaration is significant, other rule flags may precede, intervene in, or follow a tier node declaration. For example, the following declarations equivalent:

```
rule "example 1":
  iterative
  X T H
  rightmost
  match: ...
;
rule "example 2":
  X iterative T rightmost H
  match: ...
;
rule "example 3":
  X T H
  iterative
  rightmost
  match: ...
;
```

2.8.6 Regular Expression Operators

Thus far we have considered only fixed length matches. Our match specifications describe exactly what nodes must be present and absent, and no nodes may intervene between consecutive matched nodes.

Nowadays, most phonologists would argue that this is sufficient to handle the attested phenomena; indeed reference works on phonology rarely even discuss the machinery we will introduce in this section, except in very limited form (Kenstowicz 1994) (Goldsmith 1990) (Durand 1990).

The mechanisms here are not part of the phonological theory proper, but but they make many rules much simpler to state, and they allow us to accommodate aspects of the theory that are not yet codified or fully understood.

Our implementation of McCarthy and Prince’s Ilokano Reduplication analysis in §3.5 draws upon these operators in order to deal with some unformalized stipulations that the researchers place on the derivations.

The ? operator

We have already seen that the `optional` node flag denotes an optional node; the ? operator analogously marks an entire element as optional. A match specification like

```
match: [X T H] [X]? [X T H]
```

will match input matching either

```
match: [X T H] [X] [X T H]
```

or

```
match: [X T H] [X T H]
```

In other words, the middle [X] may be either present or absent, but if it is present, it will be matched, since **Pāṇini** always finds the *longest* possible match.

The * and + operators

The * operator tells **Pāṇini** to match zero or more occurrences of the modified element.¹³

The specification

```
match: [X]*
```

¹³In phonological analyses a zero subscript usually denotes this; e.g., C₀ means “a string of zero or more C slots.” Our notation provides a superset of this capability.

will match a sequence of zero or more X slots, while

```
match: [X T]*
```

will match a sequence of X slots, each linked to its own tonal root node.

The + is exactly the same except that it will match one or more elements; i.e., it acts like * but does not allow the material to be omitted entirely.

The * and + operators can be used in conjunction with node flags like **shared** and **rematchable** as well.

These operators are often used in iterative rules; however, care must be taken to put some fixed-length material in the match specification — a rule like

```
rule "runs forever":
  iterative
  match: [X]*
  effects: ...
;
```

will send the matcher into an infinite loop, because it allows the matcher to match nothing (zero X slots) over and over.

Upper and lower bounds

Contemporary phonological analyses almost never appeal to counting; rules like “delete the tone on the third vowel” are now regarded as ad-hoc. From a theoretical standpoint, users should avoid these operators in new analyses. However, older analyses did sometimes appeal to counting, and in the interest of supporting such accounts **Pāṇini** provides a way to put upper and lower bounds on the number of times an element can be matched. The syntax is as follows:

```
match: [X]{1,3}
```

This will match one, two, or three X slots; no more and no fewer.

Omitting the second number in the pair removes the upper bound. Hence $[X]\{0\}$ is equivalent to $[X]^*$, and $[X]\{1\}$ is equivalent to $[X]^+$.

The | operator

Pāṇini interprets the | operator as an “or”. For example,

```
match: [X H] | [X L]
```

will match either an X slot linked to a high tone, or an X slot linked to a low tone.

It may seem like the | is simply a shorthand for writing two different rules, but this is not the case. Compare the first rule below with the second two rules:

```
// single rule approach
rule "example 1":
  iterative
  match: [X H] | [X L]
;
// two-rule approach
rule "example 2a":
  iterative
  match: [X H]
;
rule "example 2b":
  iterative
  match: [X L]
;
```

The second two rules together will apply to all the high toned X slots and then — after all the high toned X slots have been dealt with — all the low-toned X slots. In contrast, the first rule will deal with high and low toned X-slots in the order they appear, from left to right. This difference is often crucial; see, for example, the Ilokano Reduplication analysis in §3.5.

Parentheses

In match specifications with many regular expression operators, it may not be obvious which operators apply to which elements. Furthermore, we often wish to group elements together and apply operators to the entire group.

Pāṇini support arbitrary parenthesization of elements; parentheses enclose groups that are to be treated as units with respect to operators. For example,

```
match: ([X H] | [X L])*
```

will match a sequence of zero or more units, where each unit is either an X slot linked to a high tone, or an X slot linked to a low tone.

Extraneous parentheses are perfectly legal — the following two examples are equivalent:

```
match: [X H]
match: (((([X H])))
```

Of course, the parentheses must be balanced or **Pāṇini** will report an error.

2.8.7 Effects

We have seen how to describe the context in which a rule is to apply. Now we explore the range of operations we can perform on match input nodes. We call these operations *effects*.

Effects operate on referenced nodes. Nodes with both global and local ID's are resolved as before, but for notational convenience the interpretation of an effect node reference with a single ID is different. Whereas in match specifications **Pāṇini** assumes that the single ID is the *local* ID, **Pāṇini** treats single ID's in effect reference nodes as *global* ID's. This reflects the fact that we usually want to refer to nodes by element number in effects since we do not put element in effects lists, only nodes.

This means that

```
delete X#2:1
```

refers to the X slot with local ID 1 in element 2, whereas

```
delete X#2
```

refers to the *only* X slot in element 2. If the referenced X slot were not the only X slot in element 2, we would have to specify a local ID to disambiguate the reference.

Furthermore, non-numbered nodes are still treated as node references in effects. This means that

```
delete X
```

refers to the only X slot in the match specification. If there is more than one X slot in the match specification, **Pāṇini** will report that it cannot properly resolve the reference.

LINK and DELINK

The **LINK** and **DELINK** effects affect the connections between input nodes — **LINK** creates a new connection, while **DELINK** removes an existing connection.

LINK expects two node references, separated by the **to** keyword. **DELINK** syntax is the same, except that the keyword is **from** instead of **to**. Examples follow:

```
match: [X childless T]#1 [unlinked H]#2
effect: link H#2 to T#1
```

```
match: [X T H]#1 [X T H]#2
effect: delink H#1 from T#1
       delink H#2 from T#2
```

When running a **LINK** effect, **Pāṇini** will notify the user if link already exists. Likewise, **Pāṇini** will alert the user if the nodes referenced by a **DELINK** effect are not connected. These warnings can be suppressed with the **no warnings** rule flag.¹⁴

CHANGE, INCREMENT, and DECREMENT

These effects change feature values. **CHANGE** takes a node reference and a feature name, and changes the referenced node's feature to the named value. E.g.,

```
match: [X H]
effect: change H to L
```

changes the high tone to a low tone.¹⁵ **Pāṇini** will warn the user if the change has no effect; i.e., if the referenced node already has the target feature.

INCREMENT changes the referenced node to the next feature in the defining node's feature list. In other words, **Pāṇini** looks back at the feature geometry declaration (see §2.2) and finds the feature that follows the current feature value in the node's declared feature list. If the current feature value is the last one in the list, **Pāṇini** assigns the referenced node the first feature value in the list; i.e., the **INCREMENT** effect wraps around at the end of the list.

DECREMENT is analogous; it finds the previous feature and assigns the value to the referenced node. Here is an example:

```
match: [X H]
effect: decrement H
```

Note that all these operations completely replace the referenced node's prior feature value. There is currently no way to tell **Pāṇini** to *add* features to a node; every node has exactly one feature value at a time.

¹⁴Although we will not explicitly state it in the rest of the section, note that *all* effect warnings are disabled by the **no warnings** rule flag.

¹⁵Note that this also illustrates that feature values can be substituted for node names, as usual.

DELETE

The **DELETE** effect removes an entire subtree¹⁶ from the matched input. Specifically, it deletes the referenced node *and every node connected to the referenced node by a path of child links*.

Furthermore, **DELETE** breaks any connections between deleted nodes and nodes that survive the deletion operation. (This just ensures that the remaining nodes won't have links that go nowhere.)

Consider the following rule:

```
rule "example":
  match: [syllable onset initial X]
  effect: delete X
;
```

This matches the first X slot in the onset of a syllable and deletes it, taking the entire subtree rooted at the X-slot with it. This would typically delete an entire phoneme.

STRAY ERASE

This effect does the same thing as **DELETE**, except that it will only perform the deletion if the referenced node is unlinked (i.e., has no parents). This is provided to make stray erasure rules, which are very common, easier to write.

INSERT

This effect directs **Pāṇini** to insert an element into the input stream. The **INSERT** keyword must be followed by an element declaration describing the element to be inserted. The syntax is identical to the standard element declaration syntax described in §2.3, except that references to nodes in other elements are not permitted.

In addition to the element description, we need to give **Pāṇini** information about where to put the element. There are three relevant keywords: **BEFORE**, **AFTER**, and **UNDER**. Each must be followed by a node reference.

Every **INSERT** effect must have at least one placement specification. Furthermore, if both a **BEFORE** and an **AFTER** specification are provided, only the **BEFORE** specification will be used.

The meanings of **BEFORE** and **AFTER** are “immediately before” and “immediately after”, respectively. An **UNDER** placement specification names the intended parent of the inserted material. If this is omitted

¹⁶Technically, “successor graph.”

and the input node the material is to be inserted before or after has multiple parents, the result of the **INSERT** effect is not predictable. **Pāṇini** prints a warning in this case.

The rule

```
rule "example":
  match: [word initial sigma]
  effect: insert [sigma onset d nucleus aw coda g] before sigma
;
```

inserts a syllable linked to [dɔg] before the referenced syllable (sigma). This rule assumes that the referenced sigma node will have a single parent — this is the parent the inserted material will be linked to.

Every once in a while, we need to refer to inserted material in a later effect. To accommodate this, **Pāṇini** allows inserted elements to be given new global ID's. These ID's must differ from all global ID's in the match specification and other **INSERT** effects. This allows us to write

```
rule "example":
  match: [word final sigma final X within sigma voiced]#1
  effect: insert [d glottal ~voiced]#2 after X#1
         link voiced#1 to glottal#2
;
```

to insert a [d] suffix (unmarked for voicing) and assimilate voicing with the previous segment in a single rule.

COPY

COPY works much like **INSERT**; the only difference is that instead of specifying the material to be inserted with an element declaration, we provide a reference node whose subtree is to be cloned. For example,

```
rule "example":
  match: [word initial sigma]
  effect: copy sigma before sigma
;
```

makes an exact copy of the entire referenced syllable and links it to the referenced syllable's parent immediately before the syllable.

We generally only need this effect to handle *reduplication* phenomena. See the Ilokano analysis in §3.5 for an example of a reduplication process.

One difference between `COPY` and `INSERT` is that there is no way to assign a new global ID to copied material as we can with inserted material.

`NOP`

Every rule must have at least one effect; the effects list cannot be empty. Sometimes when developing a complex rule, we wish to test only the rule's match specification (usually with the `trace` rule flag set). In these cases we can put a `NOP` effect, which takes no arguments, in the effect list as a placeholder. The `NOP` effect does nothing.

2.8.8 When Effects Cannot Apply

Effects can fail to apply (and generate a warning message in the process) for two reasons:

- A node referenced in the effect corresponds to a match node that did not match an input node.
- A node referenced in the effect corresponds to a match node for which the matching input node was deleted.

An effect that fails to apply for either of these reasons does no harm. In fact, exploiting this property allows us to write simpler rules in many cases. Consider the following rule from our account of Japanese Rendaku in §3.4:

```
rule "compounding":
  match: [word +compound morpheme#1
         morpheme#2 initial glottal within morpheme#2
         optional voiced]
  effects:
    // Delete the voiced node if there already is one.
    delete voiced
    insert [+voiced] under glottal
;
```

Note that the voiced node is optional. The first effect will delete this node if it is present in the input; otherwise the `DELETE` effect will have no effect at all.

This rule¹⁷ from the Ilokano analysis in section illustrates another way to use this aspect of effects:

```
rule "anchor template":
```

¹⁷Slightly abridged.


```

match: ([X childless V]#1 [unlinked -consonantal]#2) |
       ([X childless C]#3 [unlinked +consonantal]#4)
effect:
  link consonantal#2 to V#1
  link consonantal#4 to C#3
;

```

In this case, which effect that actually operates depends upon which half of the `|` gets matched. Both effects will never apply in a single cycle. This trick allows us to deal with both possible cases in the same rule. It is essential for some iterative rules, as the Ilokano implementation also shows.

2.8.9 Blocks and the CALL Directive

Some rules, such as the OCP (implemented the Margi analysis in §3.2) need to be applied multiple times in a single derivation. Typically these rules fix up illicit configurations, and there is reason to believe (Kenstowicz 1994, page 528) that in reality they apply continuously — in other words that they apply after every intermediate stage of a derivation.

While **Pāṇini** does not provide any way to define continuous rules, it does offer a syntactic construct that makes it easy to insert a rule at many points in a ruleset without having to retype it each time. This construct is the **CALL** directive.

The **CALL** directive takes a single argument — the name of a rule block. A rule block contains one or more rules, and is declared like so:

```

block "block name":
  <rule-declaration>
  ...
  <rule-declaration>
;

```

where "**block-name**" is any double-quoted string to name the block and **<rule-declaration>** is a rule declaration as described in §2.8. Block names must be unique.

The following example shows how to define and call a block:

```

block "example block":
  rule "test":
    iterative
    match: [X H]
    effect: delete H
;

```

```
// do other things here  
  
CALL "example-block"
```

Note that the block declaration is itself executed; in the example above, the “test” rule will be run twice — once where the block is declared, and once when the block is CALLED.

2.8.10 The GENERATE Directive

Perhaps the most important directive **Pāṇini** recognizes is the **GENERATE** directive. This directive prompts **Pāṇini** to run the specified input string through the rules (defined earlier in the file) in order.

The directive takes a double-quoted string naming the input form and an element declaration describing the input form to be run through the ruleset. Typically the inputs will reference phonemes and lexicon entries, although this is not required.

An example **GENERATE** directive follows:

```
generate "dogs" ["dog" +plural]
```

This example assumes that a “dog” lexicon entry has been defined, and that the defining geometry has a plural node that takes feature +.

Chapter 3

Examples

In this chapter we implement several analyses from the literature. The first two examples — the English plural and Margi contour tones — are tutorial in nature. The remaining examples were selected for the challenges they present any computational phonology system; we know of no other system that can handle these examples elegantly.

We close with our own analysis of a fairly large and complex set of tonal data from Bamileke-Dschang that other researchers have not analyzed satisfactorily. We propose a change to the phonological theory to handle the data, and implement our analysis to test the new theory.

3.1 The English Plural

In this section we implement an analysis of the English plural that takes advantage of our feature geometry representation and suggests a particular interpretation of the coronal affricates [ʃ] and [tʃ].

Consider the following data:

noun	plural	gloss
kæb	kæbz	'cabs'
kʌf	kʌfs	'cuffs'
kæt	kæts	'cats'
dɔg	dɔgz	'dogs'
bʌs	bʌsɪz	'buses'
bɹʌʃ	bɹʌʃɪz	'brushes'
ʃʌʃ	ʃʌʃɪz	'judges'
tʃɹtʃ	tʃɹtʃɪz	'churches'

It is clear why the plural suffix surfaces as [s] versus [z] — the suffix is unmarked for voicing and assimilates the voicing of the final segment.

The first portion of the ruleset, then, includes the standard feature geometry and phoneme definitions from the file `std.ph`, defines the lexicon entries listed in the table above, and declares the rule that inserts the plural suffix:

```

/*
 * English Plural
 */
#include "std.ph"

//
// Lexicon entries
//
lexicon "cab" [
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 ae
      coda #1 b
]
...
lexicon "church" [
  morpheme
    sigma#1
      onset#1 ch
      nucleus#1 r
      coda #1 ch
]

rule "insert plural suffix":
  match: [word +plural final X within word]#1
  effect: insert [z ~voiced] after X#1
;

```

This first rule matches any plural word¹ with an X slot; in particular, it will match the *final* X slot. Since the X node has a domain specification, it does not have to be attached to the word in any specific way; there need only be some path from the word node to the X slot.² Practically speaking, the domain specification frees us from having to list out all the places the X slot could attach — the onset, nucleus, or coda of the syllable. (Consider the plural [az] ‘ahs’, as in “oohs and ahs;” the final X slot in the base noun is in the nucleus.)

¹We handle syntactic features like plural, progressive, etc. by giving them nodes that attach directly to `word` in the defining feature geometry.

²Though we use `within` in this rule, we should technically qualify this as `linked within` to prevent matching any floating X slots in the word’s domain. Since we know that we will never have floating X slots in our data, however, we can safely use the shorter form.

The **INSERT** effect appends the plural suffix, which we declare as a [z] with its voiced node removed. Ultimately, we need to harmonize voicing so that the suffix picks up the final X slot's voicing. But notice what happens in forms like 'churches', where [i] is inserted — in these cases the [z ~ voiced] does *not* pick up the final X slot's voicing; instead, it gets its voicing from the epenthetic [i].³ So we need to insert the [i] before the voicing assimilation:

```
rule "insert vowel":
  match: [word +plural X#1 within word
         +continuant#1 coronal#1
         final X#2 within word
         +continuant#2 coronal#2 ]#1

  effect: insert [i_bar] before X#1:2
;
```

This rule matches the final two X slots in the word (the last of which will be the previously inserted [z ~ voiced] segment), but only if both segments are non-obstruents produced with the coronal articulator ([s], [z], [t], [d], and, as we will explain shortly, [č] and [j]). Given a match, the **INSERT** effect will insert an [i] between the two segments.

The final rule carries out the voicing assimilation:

```
rule "assimilate voicing":
  match: [word +plural X#1 within word glottal#1 voiced#1
         final X#2 within word glottal#2 ]#1

  effect: link voiced#1:1 to glottal#1:2
;
```

Here again we match the final two X slots in the word, only this time we focus on the glottal articulator. Since the inserted [z ~ voiced] has no voiced node, we simply link the previous node's voiced node to the final bare glottal node — this makes the [z ~ voiced] share voicing with the previous segment.

Note that if the preceding segment is unspecified for voicing (an example might be [h]), the effect will not apply and the suffix will be left without a voicing specification. To handle such cases, researchers often posit *feature-filling* rules that assign default values to unmarked features. We do not address this problem in this example ruleset, but we could add such rules using the standard **Pāṇini** rule syntax.

3.1.1 Theoretical Issues

One of the reasons this process is interesting theoretically is that it provides useful evidence regarding the structure of affricates like [č] and [j]. [Ewen 1982] argues that affricates should be analyzed not as

³Our standard phoneme definitions file marks all vowels +voiced, which is usually what we want. We will see in §3.4 that this simple approach does not work for Japanese.

single segments, as the [č] notation suggests, but as sequences like [tš].

We can cast this in terms of feature geometry by proposing that affricates contain two continuant nodes linked to the same X slot, where the first is marked [-continuant] and the second is marked [+continuant].⁴

This proposal predicts that rules that deal with the left side of an affricate will see the segment as -continuant, while rules that look at the right side of the affricate will see a +continuant feature — a supposition supported by data from Zoque (Sagey 1986) (left side) and the present analysis of the English plural (right side).

3.1.2 Putting the Rules to Work

At the end of the ruleset we generate our test words:

```
generate "cabs" [word +plural "cab"]
generate "cuff" [word +plural "cuff"]
generate "cats" [word +plural "cat"]
generate "dogs" [word +plural "dog"]
generate "bus" [word +plural "bus"]
generate "brushes" [word +plural "brush"]
generate "judges" [word +plural "judge"]
generate "churches" [word +plural "church"]
```

Here is the output for 'churches':

```
generating "churches"...
applying "insert plural suffix": rule fired
applying "insert vowel": rule fired
applying "assimilate voicing": rule fired
```

Final output:

```
word(1) [matched c1]
  plural(1) [matched c1] [ + ]
  morpheme(1)
    sigma #1(1)
      onset #1(1)
        X(1)
          consonantal(1) [ + ]
          sonorant(1) [ - ]
            continuant #1(1) [ - ]
            continuant #2(2) [ + ]
            supralaryngeal(1)
              oral_place(1)
```

⁴This corresponds to the fact that [t] is -continuant and [š] is +continuant.

```

coronal(1)
  anterior(1) [ - ]
laryngeal(1)
  glottal(1)
    voiced(1) [ - ]
rhyme(1)
  nucleus #1(1)
    X(2)
      consonantal(2) [ + ]
      sonorant(2) [ + ]
      supralaryngeal(2)
        oral_place(2)
          coronal(2)
            anterior(2) [ + ]
        laryngeal(2)
          glottal(2)
            voiced(2) [ + ]
    coda #1(1)
      X(3)
        consonantal(3) [ + ]
        sonorant(3) [ - ]
        continuant #1(3) [ - ]
        continuant #2(4) [ + ]
        supralaryngeal(3)
          oral_place(3)
            coronal(3)
              anterior(3) [ - ]
          laryngeal(3)
            glottal(3)
              voiced(3) [ - ]
      X(4) [matched c1]
        consonantal(4) [matched c1] [ - ]
        sonorant(4) [matched c1] [ + ]
        supralaryngeal(4)
          oral_place(4)
            dorsal(1)
              high(1) [ + ]
              low(1) [ - ]
            labial(1)
              round(1) [ - ]
          laryngeal(4) [matched c1]
            tongue_root(1)
              atr(1) [ - ]
            glottal(4) [matched c1]
              voiced(4) [matched c1] [changed c1] [ + ]
      X(5) [matched c1]
        consonantal(5) [matched c1] [ + ]
        sonorant(5) [matched c1] [ - ]
        continuant(5) [ + ]
        supralaryngeal(5)
          oral_place(5)
            coronal(4)
              anterior(4) [ + ]

```

```
strident(1) [ + ]
laryngeal(5) [matched c1]
  glottal(5) [matched c1] [changed c1]
    voiced(4) [matched c1] [changed c1] [ + ]
```

Scrutinizing the output reveals a problem: the [iz] suffix is crammed into the coda of the final syllable — an invalid syllabification. Again, we could write rules to resyllabify the output but have skipped this here for the sake of simplicity.⁵

⁵More complex resyllabification problems arise in other languages; we comment on this thorny issue in §4.1.1.

3.2 Margi Contour Tones

[Hoffman 1963] describes describes the two types of verbal suffixes in Margi, a Chadic language spoken in Nigeria. The following data show the tone alternations these suffixes induce:⁶

verb	type 1: bá	gloss	verb	type 2: na	gloss
cú	cíbá	'tell'	sá	sáná	'lead astray'
ghà	ghàbá	'reach'	dlà	dlànà	'overthrow'
fì	fìbá	'make swell'	bdlū	bdlènà	'forge'

Note how the tones on the suffixes change, and how the contour tones on the stems disappear. We will look at what causes these changes.

Our ruleset will implement an analysis by Kenstowicz (1994); this analysis makes use of most of the standard autosegmental devices, and is thus a good test for our system.

As usual, we include the standard definitions file `std.ph` and define our lexicon entries. However, we need to add tonal root nodes to those phonemes that can take tones (the so-called tone-bearing units). In Margi the vowels alone take tones, so the following code suffices:

```
//
// Add tonal root node to vowels.
// This makes them tone-bearing units (TBU's).
//
append phoneme i      [ T ]
append phoneme u      [ T ]
append phoneme o      [ T ]
append phoneme schwa  [ T ]
append phoneme a      [ T ]
```

Kenstowicz posits that the type one suffixes like [bá] have high tones, while the type two suffixes like [na] have no tone at all. A key point is that no tones are linked initially; they are connected according to the *Universal Association Convention*. This rule proceeds from left to right, connecting unlinked tones to bare tonal root nodes. A **Pāṇini** declaration for this rule follows.

```
rule "association convention":
  iterative

  match: [X T !modal]#1 [unlinked modal]#2
  effect: link modal#2 to T#1
;
```

⁶Transcription note: we transcribe tones with accent marks — à indicates a low-toned [a], á indicates a high-toned [a], and ä denotes a low-high contour tone.

Note the use of the ! node flag to indicate a tonal root node with no modal (base) tone linked to it. Also note that despite the fact that the two elements appear to be ordered, neither will precede or follow the other since they match nodes on different tiers. (There is no tier that both elements project a match node onto.) In cases like this, the textual ordering of the elements makes no difference.

If this rule is to be useful, our inputs must start out with unlinked tones. An example lexicon entry from this ruleset shows how to do this:

```
lexicon "gha" [ // 'reach'
  morpheme
    sigma#1
      onset#1 g h
      nucleus#1 a
      unlinked L
]
```

The `unlinked` flag puts the node within the domain of the nucleus, but does not link it to the nucleus. Furthermore, **Pāṇini** will not fill in the intermediate nodes (like T) from the nucleus to the tonal node as it will for linked children.

The previous rule accounts for the surface forms of many of the verbs, but one wonders what happens with verbs like *fī* that have two tones. We answer this question with the following docking rule:

```
rule "docking":
  iterative
  rightmost

  match: [X T] [unlinked modal]
  effect: link modal to T
;
```

This rule matches any tone that remains unlinked after the association convention has applied and “docks” it onto the final tonal node.

These rules are sufficient to handle many derivations. However, what happens in a form like [dlàna], where the verbal suffix has no tone? Given that [dlà + na] → [dlànà], we might posit a default rule assigning a low tone to those tone-bearing units that end the derivation toneless. However, this gives the wrong result for [sá + na]:

sa(H) + na	UR with floating H tone
sána	association convention
—	docking (inappl.)
sánà	default L tone
*sánà	surface form

The solution to this problem is a common one in autosegmental analyses — we employ a rule that spreads a tone onto all the free tonal nodes. In **Pāṇini** notation, the rule that works for Margi is:

```
rule "spreading":
  iterative
  match matched

  match: [T modal]#1 [T !modal]#2
  effect: link modal#1 to T#2
;
```

This rule marches across the input from left to right looking for linked tonal root nodes adjacent to bare tonal root nodes. When it finds such a match, it links the tone to the bare tonal root node. Hence it spreads the tone until it runs into an association line.

Note the **match matched** flag — we need this so that the input corresponding to the second element can be matched again in the next cycle (as the match for the first element).

This rule not only generates [sáná] correctly, but also handles disyllabic verbs like [ndábyá] that have only a single tone underlyingly.

The verbs we have looked at so far have fallen into three classes, each class corresponding to a row in our tables above. Each class has its own tonal pattern: H, L, or LH. However, Hoffman reports a fourth stem class, that he calls the “changing verb” class. These verbs surface with a low tone when unaffixed, but get a high tone when suffixed with a type one suffix like [bá] or [ŋgéri]:

verb	affixed	gloss
hù	hóbá	‘take’
fà	fáŋgéri	‘take many’

The second form, in particular, seems bizarre indeed — we begin with a low-toned stem and get a surface form with all high tones.

Pulleyblank (1986) suggests that such stems are inherently toneless — that their lexicon entries have bare tonal nodes, and hence that they pick up the tones around them. The data above supports this proposal. Consider the derivation of [fáŋgéri]. If the stem is toneless, the association convention will link the high tone contributed by the suffix to the first tonal root node in the stem. The spreading rule will then spread this high tone to all the other tonal nodes.

The only thing left to account for is the low tone on these stems when they appear in isolation. To handle this, Pulleyblank proposes a defaulting rule that fills empty tonal root nodes with low tones. We express this simple rule as follows:

```
rule "default tone":
```

```

iterative

match: [T !modal]
effect: insert [L] under T
;

```

Finally, while it is not strictly necessary in explaining this data, we follow our default tone insertion with an instance of the *Obligatory Contour Principle (OCP)*, a general constraint first proposed by Goldsmith (1976):

```

rule "OCP":
  iterative
  match matched
  no warnings

  match: ([T H]#1 [unlinked H]* [shared T H]#2) |
         ([T L]#3 [unlinked H]* [shared T L]#4)
  effect:
    // case 1: successive high tones:
    delink H#2 from T#2
    link H#1 to T#2

    // case 2: successive low tones:
    delink L#4 from T#4
    link L#3 to T#4
;

```

The OCP stipulates that adjacent nodes on the tonal tier must differ; i.e. that a sequence like HHH or LL is invalid. What appears to be a string of like tones on the surface, then, is a single tonal node linked to multiple parents. Our rule enforces this representation by changing strings of like tones, a tone at a time, into a single tone. This rule makes a good example because it requires us to use some of the more rarely used **Pāṇini** machinery.

First of all, we use the `|` regular expression operator described in §2.8.6 so that we match the tones in the order they appear. Though we could write this as two rules — one for low tones, and one for high tones — the `|` provides a way to keep both cases in the same rule so we only have to declare the rule flags once.

Likewise, we exploit the fact that **Pāṇini** will not execute an effect if a node referenced in the effect failed to match an input node. Only two effects will ever apply when this rule fires.

We also put the `shared` node flag to good use here. Without it, the rule will not match a single tonal root node linked to two like tones — a clear OCP violation. The `shared` flag allows the second T match node to match the same input node as the first T match node.

The `match matched` rule flag is also crucial. If we omit it, the rule will not match the same tonal root node twice, which means that a sequence like `[T H] [T H#1 H#2]` will get reduced to `[T H#1]#1 [T H#1:1 H#2]` instead of `[T H#1]#1 [T H#1:1]`.

Finally, note the use of the `*` operator. This allows any number of floating tones to intervene between the two offending tones. It makes our OCP rule ignore floating tones entirely.

Our last rule in this account implements *stray erasure*. This is another very common sort of autosegmental rule — it simply deletes any unlinked material. This reflects our intuition that unlinked phonological units are not pronounced (Kenstowicz 1994, page 285–6).

```
rule "stray erasure":
  iterative

  match: [unlinked modal]
  effect: stray erase modal
;
```

Technically speaking, we do not need the `unlinked` flag in the match specification — the `stray erase` effect will delete only unlinked input nodes.

The output for `[fɪbá]` follows.

```
generating "fiba 'make swell'"...
applying "association convention": rule fired
applying "docking": rule fired
applying "spreading": rule did not fire
applying "default tone": rule did not fire
applying "OCP": rule fired
applying "stray erasure": rule fired
```

Final output:

```
word(1)
  morpheme(1)
    sigma #1(1)
      onset #1(1)
        X(1)
          consonantal(1) [ + ]
            sonorant(1) [ - ]
              continuant(1) [ + ]
                supralaryngeal(1)
                  oral_place(1)
                    labial(1)
                      laryngeal(1)
                        glottal(1)
                          voiced(1) [ - ]
rhyme(1)
```

```

nucleus #1(1)
  X(2)
    consonantal(2) [ - ]
    sonorant(2) [ + ]
    supralaryngeal(2)
      oral_place(2)
        dorsal(1)
          high(1) [ + ]
          low(1) [ - ]
          back(1) [ - ]
        labial(2)
          round(1) [ - ]
      laryngeal(2)
        tongue_root(1)
          atr(1) [ + ]
        glottal(2)
          voiced(2) [ + ]
    T(1)
      modal #1(1) [ L ]
morpheme(2)
  sigma #1(2)
    onset #1(2)
      X(3)
        consonantal(3) [ + ]
        sonorant(3) [ - ]
        continuant(2) [ - ]
        supralaryngeal(3)
          oral_place(3)
            labial(3)
          laryngeal(3)
            glottal(3)
              voiced(3) [ + ]
      rhyme(2)
        nucleus #1(2)
          X(4)
            consonantal(4) [ - ]
            sonorant(4) [ + ]
            supralaryngeal(4)
              oral_place(4)
                dorsal(2)
                  high(2) [ - ]
                  low(2) [ + ]
                  back(2) [ + ]
                labial(4)
                  round(2) [ - ]
            laryngeal(4)
              tongue_root(2)
                atr(2) [ + ]
              glottal(4)
                voiced(4) [ + ]
          T(2)
            modal #2(2) [ H ]

```

3.3 Sudanese Place Assimilation

In this section we will look at some data from Sudanese Arabic that argues for giving place of articulation full status in the phonology as a separate tier. While the Halle-Sagey model (Halle 1992) does give place of articulation its own oral place tier, the motivation for this comes primarily from phonetics and physiology, not phonology. The Sudanese data we analyze here suggest that phonological rules can manipulate all the features in oral place subtree at once, and hence that oral place is indeed a phonological entity.

Hamid (1984) describes a process whereby the coronal nasal [n] assimilates the following consonant's oral place (point of articulation). Thus [n] becomes labial [m] before [b], coronal [n] before [z], velar [ŋ] before [k], labiodental [ɱ] before [f], and palatalized [ɲ] before [j]:

perfect	imperfect	gloss
nabaḥ	ya-mbaḥ	'bark'
nafad	ya-ɱfad	'save'
nazal	ya-nzil	'descend'
nasaf	ya-nsif	'demolish'
našar	ya-ɲšur	'spread'
naĵaḥ	ya-ɲĵaḥ	'succeed'
nakar	ya-ŋkur	'deny'
naxar	ya-ŋxar	'puncture'
nagal	ya-ŋgul	'transfer'
naḥar	ya-nḥar	'slaughter'
niʕis	ya-nʕas	'fall asleep'
nahab	ya-nhab	'rob'

Kenstowicz (1994) points out that we can explain all this data by positing a single rule that, once the first vowel in the stem is deleted, spreads oral place node of the stem's second X slot to the first X slot, and then deletes the first X slot's oral place subtree. The following **Pāṇini** code implements this rule:

```
rule "assimilate oral place":
  match: [word +imperfect
          initial X#1 within word
          supralaryngeal#1 oral_place#1
          X#2 within word
          oral_place#2 ]#1

  effect: delete oral_place#1:1
          link oral_place#1:2 to supralaryngeal#1:1
;
```

Handling the other parts of these derivations (deleting the first vowel and adding the prefix [ya]) is fairly straightforward. We do the first of these operations before the assimilation and the second after the assimilation.


```

        low(1) [inserted c1] [ + ]
        back(1) [inserted c1] [ + ]
        labial(1) [inserted c1]
        round(1) [inserted c1] [ - ]
    laryngeal(2) [inserted c1]
        tongue_root(1) [inserted c1]
        atr(1) [inserted c1] [ + ]
        glottal(2) [inserted c1]
        voiced(2) [inserted c1] [ + ]
sigma #1(2) [matched c1]
    onset #1(2)
        X(3)
            consonantal(3) [ + ]
            sonorant(3) [ + ]
            continuant(2) [ - ]
            supralaryngeal(2)
                soft_palette(1)
                    nasal(1) [ + ]
            oral_place(2)
                labial(2)
            laryngeal(3)
                glottal(3)
                voiced(3) [ + ]
    rhyme(2)
        nucleus #1(2)
sigma #2(3)
    onset #2(3)
        X(4)
            consonantal(4) [ + ]
            sonorant(4) [ - ]
            continuant(3) [ - ]
            supralaryngeal(3)
                oral_place(2)
                    labial(2)
            laryngeal(4)
                glottal(4)
                voiced(4) [ + ]
    rhyme(3)
        nucleus #2(3)
            X(5)
                consonantal(5) [ - ]
                sonorant(5) [ + ]
                supralaryngeal(4)
                    oral_place(3)
                        dorsal(2)
                            high(2) [ - ]
                            low(2) [ + ]
                            back(2) [ + ]
                    labial(3)
                        round(2) [ - ]
                laryngeal(5)
                    tongue_root(2)
                    atr(2) [ + ]

```

```
          glottal(5)
            voiced(5) [ + ]
coda #2(1)
  X(6)
    consonantal(6) [ - ]
      sonorant(6) [ + ]
        laryngeal(6)
          glottal(6)
            spread_gl(1) [ + ]
              voiced(6) [ - ]
            pharyngeal(1)
```

3.4 Japanese Rendaku

Now we turn to some data from Japanese compounding (Itô and Mester 1986). Notice that in each of the forms in the top half of the table the initial consonant in the second noun gets voiced. The challenge is to explain why the voicing does not occur in the forms in the bottom half of the table.

stem ₁	+	stem ₂	→	compound
eda 'branch'	+	ke 'hair'	→	edage 'split hair'
unari 'moan'	+	koe 'voice'	→	unarigoe 'groan'
mizu 'water'	+	seme 'torture'	→	mizuzeme 'water torture'
ori 'fold'	+	kami 'paper'	→	origami 'origami paper'
neko 'cat'	+	šita 'tongue'	→	nekošita 'aversion to hot food'
ko 'child'	+	tanuki 'raccoon'	→	kodanuki 'baby raccoon'
kita 'north'	+	kaze 'wind'	→	kitakaze (*kitagaze) 'freezing north wind'
širo 'white'	+	tabi 'tabi'	→	širotabi (*širodabi) 'white tabi'
taikutsu 'time'	+	šinogi 'avoiding'	→	taikutsušinogi (*taikutsušinogi) 'time killer'
onna 'woman'	+	kotoba 'words'	→	onnakotoba (*onnagotoba) 'feminine speech'
doku 'poison'	+	tokage 'lizard'	→	dokutokage (*dokudokage) 'Gila monster'

Ito and Mester point out that in the forms where the voicing fails to apply, there is a voiced obstruent in the second stem. While we could write a **Pāṇini** rule to implement exactly this intuitive notion by using the * operator to skip [–voiced] nodes intervening between the first consonant and the voiced obstruent, such a rule is theoretically unappealing because it relies on nonlocal context. Since the vast majority⁷ of phonological processes can be explained by rules that use only local context (i.e., directly adjacent nodes), and since a locality requirement on all rules makes for a much simpler theory⁸, we would greatly prefer an explanation of these facts that requires only local context.

Ito and Mester propose an *underspecification* analysis. In other words, some phonemes are *underspecified* and therefore project no nodes at all onto certain tiers (much as consonants — since they are not tone-bearing units — fail to project nodes onto the tonal root node tier in Margi; see §3.2). Ito and Mester

⁷Evidence suggests all, in fact. (Kenstowicz 1994)

⁸Simpler in the sense that the range of phenomena that can be described is more limited. Linguists claim that simpler theories are better, appealing not only to Occam's Razor, but to learnability arguments as well — if the range of possible rules is smaller, learning a particular rule is easier. (Kenstowicz 1994, page 153) Furthermore, if the context is unbounded, rules within theory are unlearnable since the probability of getting a positive example goes to zero as the size of the context goes to infinity.

argue that only the voiced obstruents are specified for voicing in Japanese; i.e. that all other phonemes simply leave the voiced tier empty. They posit a late rule that fills in the gaps with default values so the phonemes get pronounced properly.

Given this proposal, a compound like *ori* + *kami* undergoes the voicing because the voicing tier is empty following the initial [k], since the sonorant [m] is underspecified for voicing and hence projects no node onto this tier.

In contrast, *kita* + *kaze* fails to undergo the voicing process, because the obstruent [z] is marked for voicing. Compare this to *ko* + *tanuki*, where the voicing rule does apply; here the second stem contains an obstruent, but the obstruent is unvoiced [k], and hence projects no voicing node. This derivation is particularly interesting because it shows that the intervening nasal [n], like its labial counterpart [m], has no voicing node.

The **Pāṇini** rules that implement this analysis should be straightforward to readers who have already gone through the previous analyses, so we will include them here without much comment. However, note the use of **append** and the `~` node flag to implement the underspecification, and the crucial use of the domain specification in the first two rules to limit the context to nodes within the second morpheme.⁹

The implementation first applies the voicing operation, then *undoes* it produces the illegal configuration of two voiced nodes in the same morpheme.¹⁰

```
#include "std.ph"

//
// First we remove all the vowels' voicing specifications.
//
append phoneme i      [ ~voiced ]
append phoneme I      [ ~voiced ]
append phoneme e      [ ~voiced ]
append phoneme eh     [ ~voiced ]
append phoneme ae     [ ~voiced ]
append phoneme u      [ ~voiced ]
append phoneme U      [ ~voiced ]
append phoneme o      [ ~voiced ]
append phoneme schwa  [ ~voiced ]
append phoneme a      [ ~voiced ]
append phoneme aw     [ ~voiced ]

//
// Now remove voicing specifications from all but the
// voiced obstruents.
//
```

⁹This argues that forms like *edage* still surface when they precede other morphemes containing voiced obstruents, a claim borne out by the data (Kenstowicz 1994, page 511).

¹⁰This is a general prohibition in Japanese called Lyman's Law; it applies in many other derivations as well, and is thus motivated by data beyond what we have shown here. (Kenstowicz 1994, page 162)

```

append phoneme p      [ ~voiced ]
append phoneme k      [ ~voiced ]
append phoneme t      [ ~voiced ]
append phoneme f      [ ~voiced ]
append phoneme s      [ ~voiced ]
append phoneme sh     [ ~voiced ]
append phoneme ts     [ ~voiced ]
append phoneme ch     [ ~voiced ]

//
// Nasals are unspecified for voicing as well.
//
append phoneme m      [ ~voiced ]
append phoneme n      [ ~voiced ]

// (lexicon entries omitted for brevity's sake)

rule "compounding":
  no warnings

  match [word +compound morpheme#1
        morpheme#2
        initial glottal within morpheme#2
        optional voiced]

  effects:
    //
    // Delete the voiced node if there already is one.
    // This rule will not apply if no voiced node got matched.
    //
    delete voiced
    insert [+voiced] under glottal
    change compound to -
;

rule "lyman's law":
  iterative

  match: [morpheme +voiced#1 within morpheme
        +voiced#2 within morpheme]#1
  effect: delete voiced #1:1
;

rule "feature-fill voicing (sonorants)":
  iterative

  match: [+sonorant glottal !voiced]
  effect: insert [+voiced] under glottal
;

```

```

rule "feature-fill voicing (obstruents)":
  iterative

  match: [-sonorant glottal !voiced]
  effect: insert [-voiced] under glottal
;

```

The success of analyses like these argues strongly for a feature geometry view of the segment. Without this representation we must appeal to more complex rule formalisms that support long-distance operations and contexts (Kenstowicz 1994, page 163).

Finally, notice that we have completely ignored the *value* of the voicing feature in this account — our rules depend only on the presence or absence of voicing nodes, regardless of these nodes' +/– feature values. This interpretation of the voiced feature as *monovalent* makes many theoretical predictions. Some of these are examined in (Itô and Mester 1986) and (Lombardi 1991).

A sample run on *kita + kaze* follows.

```

generating "freezing north wind (kitakaze)"...
applying "compounding": rule fired
applying "lyman's law": rule fired
applying "feature-fill voicing (sonorants)": rule fired
applying "feature-fill voicing (obstruents)": rule fired

```

Final output:

```

word(1)
  compound(1) [ - ]
  morpheme(1)
    sigma #1(1)
      onset #1(1)
        X(1)
          consonantal(1) [ + ]
            sonorant(1) [matched c1] [ - ]
              continuant(1) [ - ]
                supralaryngeal(1)
                  oral_place(1)
                    dorsal(1)
                  laryngeal(1) [matched c1]
                    glottal(1) [matched c1]
                      voiced(1) [inserted c1] [ - ]
          rhyme(1)
            nucleus #1(1)
              X(2)
                consonantal(2) [ - ]
                  sonorant(2) [ + ]
                    supralaryngeal(2)
                      oral_place(2)
                        dorsal(2)
                          high(1) [ + ]

```

```

        low(1) [ - ]
        back(1) [ - ]
        labial(1)
        round(1) [ - ]
    laryngeal(2)
        tongue_root(1)
        atr(1) [ + ]
        glottal(2)
        voiced(2) [ + ]
sigma #2(2)
    onset #2(2)
        X(3)
            consonantal(3) [ + ]
            sonorant(3) [matched c2] [ - ]
            continuant(2) [ - ]
            supralaryngeal(3)
                oral_place(3)
                coronal(1)
                anterior(1) [ + ]
            laryngeal(3) [matched c2]
            glottal(3) [matched c2]
            voiced(3) [inserted c2] [ - ]
    rhyme(2)
        nucleus #2(2)
            X(4)
                consonantal(4) [ - ]
                sonorant(4) [ + ]
                supralaryngeal(4)
                    oral_place(4)
                    dorsal(3)
                        high(2) [ - ]
                        low(2) [ + ]
                        back(2) [ + ]
                    labial(2)
                    round(2) [ - ]
                laryngeal(4)
                    tongue_root(2)
                    atr(2) [ + ]
                    glottal(4)
                    voiced(4) [ + ]
morpheme(2)
    sigma #1(3)
        onset #1(3)
            X(5)
                consonantal(5) [ + ]
                sonorant(5) [matched c3] [ - ]
                continuant(3) [ - ]
                supralaryngeal(5)
                    oral_place(5)
                    dorsal(4)
                laryngeal(5) [matched c3]
                glottal(5) [matched c3]
                voiced(5) [inserted c3] [ - ]

```

```

rhyme(3)
  nucleus #1(3)
    X(6)
      consonantal(6) [ - ]
      sonorant(6) [ + ]
      supralaryngeal(6)
        oral_place(6)
          dorsal(5)
            high(3) [ - ]
            low(3) [ + ]
            back(3) [ + ]
          labial(3)
            round(3) [ - ]
        laryngeal(6)
          tongue_root(3)
            atr(3) [ + ]
          glottal(6)
            voiced(6) [ + ]
sigma #2(4)
  onset #2(4)
    X(7)
      consonantal(7) [ + ]
      sonorant(7) [ - ]
      continuant(4) [ + ]
      supralaryngeal(7)
        oral_place(7)
          coronal(2)
            anterior(2) [ + ]
        strident(1) [ + ]
      laryngeal(7)
        glottal(7)
          voiced(7) [ + ]
rhyme(4)
  nucleus #2(4)
    X(8)
      consonantal(8) [ - ]
      sonorant(8) [ + ]
      supralaryngeal(8)
        oral_place(8)
          dorsal(6)
            high(4) [ - ]
            low(4) [ - ]
            back(4) [ - ]
          labial(4)
            round(4) [ - ]
        laryngeal(8)
          tongue_root(4)
            atr(4) [ + ]
          glottal(8)
            voiced(8) [ + ]

```


3.5 Ilokano Reduplication

McCarthy and Prince (1986) report the following data for a reduplication process in Ilokano, a Philippine language.

root	progressive	gloss
basa	ag-bas-basa	'read'
adal	ag-ad-adal	'study'
da-it	ag-da-dait	'sew'
takder	ag-tak-takder	'standing'
trabaho	ag-trab-trabaho	'work'

This kind of phenomenon, where some portion of the base form appears to be copied, is called *reduplication*.

Initially it is tempting to explain the data by positing a rule that copies the entire first syllable. Unfortunately, this makes false predictions like [*ba-basa] and [*a-adal].

The correct analysis, proposed by Marantz (1982) and further developed by McCarthy and Prince, involves *template filling*, a process that plays a major role in Semitic languages (McCarthy 1975).

Specifically, these researchers suggest that the reduplication rule makes a copy of the entire word, and then fits as much of the material as possible into a CCVC template. Not coincidentally, this pattern is the maximal syllable in Ilokano. Our first three rules, then, copy the word, remove the C and V slots from the copy (thereby turning all the segments into floating units), and insert the template.¹¹

```
#include "cv.h"
// Stage 1: Duplicate the entire word.

rule "reduplication stage 1 (duplicate word)":
  match: [word +progressive morpheme]
  effect: copy morpheme under word before morpheme
;

// Stage 2: Delink all consonantal nodes in copied material.

rule "reduplication stage 2 (delink consonantal nodes)":
  iterative
  match matched
  no warnings
```

¹¹For this ruleset we include the file `cv.h` instead of the usual `std.h`. This is necessary because `std.h` does not distinguish between C and V slots; it only deals with X slots, whereas `cv.h` assigns X two child tiers — C for consonants and V for vowels. Using `std.h` simplifies rulesets for many languages because it makes a match element declaration like `[X consonantal]` unambiguous; contrast this with the CV case, where the consonantal node could link to either a C or a V.

```

match: [word +progressive initial morpheme
        X within morpheme C consonantal]#1 |
        [word +progressive initial morpheme
        X within morpheme V consonantal]#2
effect: delink consonantal#1 from C
        delink consonantal#2 from V
;

// Stage 3: Delete the duplicated morpheme node and all its inferiors.
//         Since we have delinked all the consonantal nodes, this won't
//         affect the segmental material.

rule "reduplication stage 3 (delete morpheme)":
  match: [word +progressive initial morpheme]
  effect: delete morpheme
;

// Stage 4: Insert the reduplication template.

rule "insert template":
  match: [word +progressive initial morpheme]
  effect:
    insert [
      morpheme
      sigma
      onset      X#1 C#1
                X#2 C#2
      nucleus    X#3 V#3
      coda       X#4 C#4
    ] before morpheme
;

```

Kenstowicz (1994) points out that this kind of analysis is only expressible given a model that puts skeletal and segmental material on different tiers. He goes on to discuss some unformalized stipulations on the feature-filling procedure:

Marantz notes certain technical details in getting the association to work correctly. First, it must be “phoneme-driven” in the sense that segments are matched one by one to the template. Mapping in the opposite direction (from the skeletal template to the segmental tier) creates immediate problems: given the sequence [bas], mapping C_1 to [b] and C_2 to [s] prevents matching V to [a] without crossing an association line.

Second, the mapping must restrict itself to a continuous portion of the segmental tier. It may skip positions in the template when the latter is not expanded fully; for example, in mapping [ad], we must skip the first two C-slots, and for [dait], we must skip the second C-slot. But why can't the final C-slot match with a phoneme farther down the line, skipping the [i] and generating the impossible *[dat-da-it]? The requirement that only a contiguous region of the phonemic string can be mapped blocks this unwanted outcome... (p. 624)

This kind of textual (as opposed to notational) rule qualification is quite common in the literature, so any system intended to support real-world analyses must be powerful enough to cope with such stipulations. Fortunately, we can put a few of **Pāṇini**'s more powerful features to use to implement the template-filling operation properly. The first is tier ordering.

To make the filling operation phoneme-driven, we have **Pāṇini** compare the relative positions of candidate matches first on the consonantal tier.¹² Specifically, we find the leftmost bare C/V node and unlinked matching consonantal node, giving preference to those matches with earlier consonantal nodes. This always matches the leftmost consonantal node to the leftmost C/V slot it can be paired with.¹³

In order to implement the second stipulation Kenstowicz mentions, we have to break up the rule into two pieces. The first finds an anchor point — the leftmost template slot that matches the first consonantal node. The second part then fills in the rest of the template, enforcing the requirement that the filled material be contiguous. In **Pāṇini** parlance, this just means that we can only link a consonantal node to a C/V slot if the consonantal node directly follows a consonantal node that has already been linked. Since the anchor node is the only one that does not need to obey this stipulation, it gets its own rule.

```
rule "anchor template":
  no warnings
  consonantal
  match: ([X childless V]#1 [unlinked -consonantal]#2) |
         ([X childless C]#3 [unlinked +consonantal]#4)
  effect:
    // Vowel case:
    link consonantal#2 to V#1
    // Consonant case:
    link consonantal#4 to C#3
;
rule "fill template":
  iterative
  match matched
  no warnings
  match: ([X consonantal linked within X]
         [X]*
         [X childless V]#1 [unlinked -consonantal]#2)
         | // or
         ([X consonantal linked within X]
         [X]*
         [X childless C]#3 [unlinked +consonantal]#4)
  effect:
    // Vowel case:
    link consonantal#2 to V#1
    // Consonant case:
```

¹²By default it would look at the C and V tiers first, which would give us a template-driven mapping.

¹³Only +consonantal nodes can link to C slots; only -consonantal nodes can link to V slots.

```

link consonantal#4 to C#3
;

```

As in the implementation of the OCP in Margi (§3.2), we use the | operator to fill in the slots in the order they appear, rather than matching all the [-consonantal] nodes before any of the [+consonantal] nodes. In this case, however, rewriting the rule as two rules will break it; the use of | is crucial.

The [X]* expressions handle another technical detail. Recall from §2.8.1 that consecutive nodes in match specifications will match only consecutive input nodes. Without the [X]* expressions, **Pāṇini** will only fill adjacent consonantal nodes into adjacent C/V slots. This results in the incorrect *[b-basa], where the [b] links to the first C slot and the [a] cannot link to the second C slot.

Finally, we need to insert the prefix [ag-] and erase stray material. Since these rules are not very interesting, we will skip the discussion of them. See §A.6 for the complete ruleset text.

A sample run on [da-it] follows.

```

generating "agdadait"...
applying "reduplication stage 1 (duplicate word)": rule fired
applying "reduplication stage 2 (delink consonantal nodes)": rule fired
applying "reduplication stage 3 (delete morpheme)": rule fired
applying "insert template": rule fired
applying "anchor template": rule fired
applying "fill template": rule fired
applying "add prefix ag-": rule fired
applying "stay erasure 1": rule fired
applying "stray erasure 2": rule fired
applying "stray erasure 3": rule fired
applying "stray erasure 4": rule fired

```

Final output:

```

word(1)
  progressive(1) [ - ]
  morpheme(1)
    sigma(1)
      rhyme(1)
        nucleus(1)
          X(1)
            V(1)
              consonantal(1) [ - ]
                sonorant(1) [ + ]
                  supralaryngeal(1)
                    oral_place(1)
                      dorsal(1)
                        high(1) [ - ]
                          low(1) [ + ]
                            back(1) [ + ]
                              labial(1)
                                round(1) [ - ]
                                  laryngeal(1)

```

```

                                tongue_root(1)
                                atr(1) [ + ]
                                glottal(1)
                                voiced(1) [ + ]
coda(1)
  X(2)
    C(1)
      consonantal(2) [ + ]
      sonorant(2) [ - ]
      continuant(1) [ - ]
      supralaryngeal(2)
      oral_place(2)
      dorsal(2)
      laryngeal(2)
      glottal(2)
      voiced(2) [ + ]
sigma(2)
  onset(1)
    X #1(3)
      C #1(2)
        consonantal(3) [ + ]
        sonorant(3) [ - ]
        continuant(2) [ - ]
        supralaryngeal(3)
        oral_place(3)
        coronal(1)
        anterior(1) [ + ]
        laryngeal(3)
        glottal(3)
        voiced(3) [ + ]
rhyme(2)
  nucleus(2)
    X #3(4)
      V #3(2)
        consonantal(4) [ - ]
        sonorant(4) [ + ]
        supralaryngeal(4)
        oral_place(4)
        dorsal(3)
          high(2) [ - ]
          low(2) [ + ]
          back(2) [ + ]
        labial(2)
          round(2) [ - ]
        laryngeal(4)
          tongue_root(2)
          atr(2) [ + ]
          glottal(4)
          voiced(4) [ + ]
morpheme(2)
  sigma #1(3)
    onset #1(2)
      X(5)

```

```

C(3)
  consonantal(5) [ + ]
  sonorant(5) [ - ]
    continuant(3) [ - ]
    supralaryngeal(5)
      oral_place(5)
        coronal(2)
          anterior(2) [ + ]
    laryngeal(5)
      glottal(5)
        voiced(5) [ + ]
rhyme(3)
  nucleus #1(3)
    X(6)
      V(3)
        consonantal(6) [ - ]
        sonorant(6) [ + ]
          supralaryngeal(6)
            oral_place(6)
              dorsal(4)
                high(3) [ - ]
                low(3) [ + ]
                back(3) [ + ]
              labial(3)
                round(3) [ - ]
          laryngeal(6)
            tongue_root(3)
              atr(3) [ + ]
            glottal(6)
              voiced(6) [ + ]
sigma #2(4)
  rhyme(4)
    nucleus #2(4)
      X(7)
        V(4)
          consonantal(7) [ - ]
          sonorant(7) [ + ]
            supralaryngeal(7)
              oral_place(7)
                dorsal(5)
                  high(4) [ + ]
                  low(4) [ - ]
                  back(4) [ - ]
                labial(4)
                  round(4) [ - ]
            laryngeal(7)
              tongue_root(4)
                atr(4) [ + ]
              glottal(7)
                voiced(7) [ + ]
coda #2(2)
  X(8)
    C(4)

```

```
consonantal(8) [ + ]
  sonorant(8) [ - ]
    continuant(4) [ - ]
      supralaryngeal(8)
        oral_place(8)
          coronal(3)
            anterior(3) [ + ]
      laryngeal(8)
        glottal(8)
          voiced(8) [ - ]
```

3.6 Register Tone in Bamileke-Dschang

We have seen that **Pāṇini** can handle a wide range of textbook examples. In this section we will explore a “real world” phonological analysis with a large data set that phonologists have not yet adequately explained.¹⁴ The data is from Bamileke-Dschang compounding (Hyman 1985) (Pullyblank 1982), and is shown in tables 3.1 and 3.2.¹⁵

What is fascinating about this data is the behavior of the downsteps — they seem to appear out of nowhere and move around randomly.

Both Pulleyblank and Hyman have attempted to analyze the Dschang downstep data and have run into problems that require tonal metathesis and a variety of ad hoc fixup rules. Both these researchers based their analyses on the assumption that the downsteps correspond to low tones on a second “register” tier — a tier that behaves exactly like the primary “modal” tier.¹⁶ Here we propose a slightly different view of the register tier, and use **Pāṇini** to test this new theory on the Dschang data.

We propose that nodes on the register tier behave differently than their modal tier counterparts. In particular, since register shifts are *relative* to the previous register setting, whereas modal changes are *absolute*, we believe that it is in fact counterintuitive for the register tier to obey the OCP like the modal tier.

We find evidence supporting this intuition in Babungo, a Grassfields Bantu language spoken in Cameroon. Schaub (1985) reports the following surface forms resulting from $N_1 + N_2$ compounding in Babungo:

compound	gloss
bi ši	‘the goat of the nobleman’
tɔ [!] ši	‘the head of the nobleman’
tɔ ^{!!} bi	‘the head of the goat’

The double downstep in the third example argues directly against the OCP on the register tier.¹⁷

Removing the OCP constraint and giving register nodes full status in the theory allows a simpler and more natural analysis of the Dschang data using standard autosegmental rules.

First we establish which phonemes are tone-bearing units. Dschang appears to treat only syllable heads

¹⁴In fact, this data has proven so difficult that Bird (1993) has done a machine-aided analysis of the data to check Hyman’s transcriptions.

¹⁵Transcription notes: Tones are marked as in the Margi example (§3.2). Extra tone marks at the ends of morphemes indicate whether the tone rises, falls, or stays level. We will not deal with this aspect of the data here. A raised exclamation point indicates a tonal downstep — a shift in tonal register that lowers all the following tones until the end of the morpheme. The capital V in, e.g. entry three in table 3.2, is a vowel slot that assimilates the features of the neighboring vowels. We will deal only with tonal phenomena in this analysis, so we will not explain this assimilation process or any other segmental changes here.

¹⁶This terminology follows (Snider 1990).

¹⁷Recall that the OCP prohibits two consecutive nodes with the same feature value — this means that [down] [down] is not allowed.

as tone-bearing units (Hyman 1985), and we can see from the data that nasals and glottal stops can act as tone-bearing units when necessary. The following **Pāṇini** code implements these aspects of Dschang:

```
//
// Add tonal root node to vowels.
// This makes them tone-bearing units (TBU's).
//
append phoneme i      [ T ]
append phoneme u      [ T ]
append phoneme o      [ T ]
append phoneme schwa [ T ]
append phoneme a      [ T ]
append phoneme aw     [ T ]
append phoneme gstop [ T ]
append phoneme n      [ T ]
append phoneme ny     [ T ]
append phoneme ng     [ T ]

...

rule "TBU cleanup":
  iterative
  match matched
  no warnings

  match: [ onset T ]#1 | [ coda T ]#2
  effect: delete T#1
         delete T#2
;
```

We posit several continuous rules — rules that apply after every rule in the derivation. These rules simply repair any illegal configurations that may arise.¹⁸ Since there is no direct way to tell **Pāṇini** that a rule is continuous, we declare our rules as a block and then **CALL** the block (§2.8.9) after each rule.

```
block "continuous": // rules that apply continuously
  rule "contour reduction":
    iterative

    match: [ T L#1 H#2 down#1 ]
    effect: delink L from T
  ;

  rule "down deletion":
    iterative
```

¹⁸As we can see from the data, the sequence H¹L is prohibited; we reduce it to HL. Likewise, LH contours cannot take downsteps, and a single TBU cannot be double-downstepped.

```

    match: [ H ]#1 [ T L down ]#2
    effect: delete down
;

rule "double downstep reduction":
  iterative
  match matched

  match: [ T down#1 down#2 ]
  effect: delete down#1
;
rule "floating L deletion":
  iterative

  match: [unlinked L]
  effect: stray erase L
;
;

```

Our first main rule explains the downsteps in the lower half of table 3.1. We claim that the [mə] prefix has a linked downstep underlyingly; the following “downstep hopping” rule moves the downstep to the right.

```

rule "downstep hopping":
  iterative
  match matched

  match: [ morpheme T#1 within morpheme L#1 down#1
          T#2 within morpheme !register ]#1
  effect: link down#1 to T#2
          delink down#1 from T#1
;

```

The rule stipulates that a downstep hops off a TBU linked to a low tone and onto the following TBU if that TBU does not already have a linked downstep. This rule explains why the downsteps appear on the final TBU’s in forms like table 3.1 #17.¹⁹ Note that a downstep will not hop off a morpheme — here is another case where the ability to restrict rule application to certain domains is crucial.

The next rule explains why the [á] in entries 25 through 32 of both tables gets a high tone, and why high tones in the second stems in these forms are so common.

```

rule "H spreading right":

```

¹⁹It also explains why entry 19 in the same table keeps its downstep in the penultimate TBU — we claim the stem [məmbhʉ] is linked to two downsteps underlyingly, and hence the second downstep prevents the first from moving off of [mə].

```

iterative
match matched

match: [ T H ]#1 [ T L !register ]#2
effect: link H#1 to T#2
       delink L#2 from T#2
       stray erase L#2
;

```

The rule spreads a high tone onto a following low-toned TBU if the TBU has no downstep. Since the nouns [àláʔ] ‘country’ and [àsáj] ‘tail’ have high tones linked to their final TBU’s, this rule spreads the high tone onto the associative marker [á] and the final stem. Unlike the downstep hopping rule, this rule crucially applies across morpheme boundaries.

An additional spreading process applies in several compounds where the final stem is either [məmbhə] or [mətsəŋ]. We have written this as two separate **Pāṇini** rules for simplicity’s sake; we could have used the | regular expression operator to combine both cases into a single match pattern, but the present approach seems more readable.

```

rule "H spreading left 1":
  match: [ T L ]#1
        [ T L ]#2
        [ word final T within word H down ] #3

  effect: link H#3 to T#2
         delink L#2 from T#2
         link down#3 to T#2
         delink down#3 from T#3
;
rule "H spreading left 2":
  match: [ T H ]#1
        [ T L ]#2
        [ word final T within word H down ] #3

  effect: link H#3 to T#2
         delink L#2 from T#2
;

```

Note that the rule is not iterative. It states that a word-final TBU with a high tone and a downstep will spread to the previous TBU, taking the downstep with it. A stipulation, however, is that the rule will not produce the illegal H¹L configuration; it simply fails to move the downstep in such cases. (This is the case that the second rule above handles.)

Our final rule describes what happens to floating downsteps. This is important because we posit an underlying unlinked downstep in the stem [ɲɲi] — this explains why [səŋ] is downstepped in entry 16 of table 3.2. It also correctly predicts entries 5 and 6 in this table, if we assume that the associative vowel

[a] is first deleted.²⁰

```
rule "down docking":
  rightmost

  match: [ T unlinked down within T ]#1 [ T ]#2
  effect: link down to T#2
;
```

The rule specifies that a downstep links the to the following tone-bearing unit. Note that this rule does not work properly if the domain T is not specified; without it, the downstep will just attach to the rightmost TBU in the word!

There are a few other rules, but these handle details like stray erasure. For a complete account, including the underlying representations of all the stems, see the ruleset text in §A.7.

Our analysis correctly predicts 61 of the 64 forms listed in the two tables. Two of the forms, #5 and #6 in table 3.2, work properly if we assume the missing associative marker [a] is deleted before the downstep docking rule applies. (Or perhaps the docking rule is continuous.)

Form 8 in table 3.2 is perplexing. We predict the resulting form, except that the associative marker gets a high tone where we predict a low tone. Since this does not happen to the [ndzà] stem anywhere else, we suspect this is a marked case.

A sample run on entry 26 in table 3.1 follows.

```
generating "country of roosters"...
applying "TBU cleanup": rule fired
applying "contour reduction": rule did not fire
applying "down deletion": rule did not fire
applying "double downstep reduction": rule did not fire
applying "floating L deletion": rule did not fire
applying "down hopping": rule fired
applying "contour reduction": rule did not fire
applying "down deletion": rule did not fire
applying "double downstep reduction": rule did not fire
applying "floating L deletion": rule did not fire
applying "H spreading right": rule fired
applying "contour reduction": rule did not fire
applying "down deletion": rule fired
applying "double downstep reduction": rule did not fire
applying "floating L deletion": rule did not fire
applying "H spreading left 1": rule did not fire
applying "H spreading left 2": rule did not fire
applying "contour reduction": rule did not fire
applying "down deletion": rule did not fire
```

²⁰We assume there is some explanation for this data using segmental rather than tonal data.

applying "double downstep reduction": rule did not fire
 applying "floating L deletion": rule did not fire
 applying "down docking": rule did not fire
 applying "contour reduction": rule did not fire
 applying "down deletion": rule did not fire
 applying "double downstep reduction": rule did not fire
 applying "floating L deletion": rule did not fire

Final output:

```

word(1)
  morpheme(1)
    sigma #1(1)
      rhyme(1)
        nucleus #1(1)
          X(1)
            consonantal(1) [ - ]
            sonorant(1) [ + ]
            supralaryngeal(1)
              oral_place(1)
                dorsal(1)
                  high(1) [ - ]
                  low(1) [ + ]
                  back(1) [ + ]
            labial(1)
              round(1) [ - ]
            laryngeal(1)
              tongue_root(1)
                atr(1) [ + ]
            glottal(1)
              voiced(1) [ + ]
          T(1)
            modal #1(1) [ L ]
        sigma #2(2)
          onset #2(1)
            X(2)
              consonantal(2) [ + ]
              sonorant(2) [ + ]
              lateral(1) [ + ]
              supralaryngeal(2)
                oral_place(2)
                  coronal(1)
                    anterior(1) [ + ]
              laryngeal(2)
                glottal(2)
                  voiced(2) [ + ]
          rhyme(2)
            nucleus #2(2)
              X(3)
                consonantal(3) [ - ]
                sonorant(3) [ + ]
                supralaryngeal(3)
                  oral_place(3)
                    dorsal(2)
  
```

```

        high(2) [ - ]
        low(2) [ + ]
        back(2) [ + ]
        labial(2)
        round(2) [ - ]
    laryngeal(3)
        tongue_root(2)
        atr(2) [ + ]
        glottal(3)
        voiced(3) [ + ]
    T(2)
        modal #2(2) [ H ]
        register(1) [ down ]
    coda #2(1)
        X(4)
            consonantal(4) [ - ]
            sonorant(4) [ + ]
            laryngeal(4)
            glottal(4)
            constr_gl(1) [ + ]
            voiced(4) [ - ]
    morpheme(2)
        sigma #1(3)
        rhyme(3)
            nucleus #1(3)
                X(5)
                    consonantal(5) [ - ]
                    sonorant(5) [ + ]
                T(3)
                    modal #2(2) [ H ]
    morpheme(3)
        sigma #1(4)
        onset #1(2)
            X(6)
                consonantal(6) [ + ]
                sonorant(6) [ + ]
                continuant(1) [ - ]
                supralaryngeal(4)
                    soft_palette(1)
                        nasal(1) [ + ]
                    oral_place(4)
                        labial(3)
                laryngeal(5)
                    glottal(5)
                    voiced(5) [ + ]
        rhyme(4)
            nucleus #1(4)
                X(7)
                    consonantal(7) [ - ]
                    sonorant(7) [ + ]
                    supralaryngeal(5)
                        oral_place(5)
                        dorsal(3)

```

```

        high(3) [ - ]
        low(3) [ - ]
        back(3) [ + ]
        labial(4)
        round(3) [ - ]
    laryngeal(6)
        tongue_root(3)
        atr(3) [ - ]
        glottal(6)
        voiced(6) [ + ]
    T(4)
        modal #2(2) [ H ]
    coda #1(2)
    X(8)
        consonantal(8) [ + ]
        sonorant(8) [ + ]
        continuant(2) [ - ]
        supralaryngeal(6)
        oral_place(6)
        coronal(2)
        anterior(2) [ - ]
        dorsal(4)
        soft_palette(2)
        nasal(2) [ + ]
        laryngeal(7)
        glottal(7)
        voiced(7) [ + ]
    sigma #2(5)
        onset #2(3)
        X(9)
            consonantal(9) [ + ]
            sonorant(9) [ - ]
            continuant(3) [ - ]
            supralaryngeal(7)
            oral_place(7)
            dorsal(5)
            laryngeal(8)
            glottal(8)
            voiced(8) [ - ]
    rhyme(5)
        nucleus #2(5)
        X(10)
            consonantal(10) [ - ]
            sonorant(10) [ + ]
            supralaryngeal(8)
            oral_place(8)
            dorsal(6)
            high(4) [ - ]
            low(4) [ - ]
            back(4) [ + ]
            labial(5)
            round(4) [ + ]
            laryngeal(9)

```

```

        tongue_root(4)
          atr(4) [ + ]
        glottal(9)
          voiced(9) [ + ]
      T(5)
        modal #1(3) [ L ]
coda #2(3)
  X(11)
    consonantal(11) [ - ]
    sonorant(11) [ + ]
    supralaryngeal(9)
      oral_place(9)
        dorsal(7)
          high(5) [ - ]
          low(5) [ + ]
          back(5) [ + ]
        labial(6)
          round(5) [ + ]
    laryngeal(10)
      tongue_root(5)
        atr(5) [ + ]
      glottal(10)
        voiced(10) [ + ]
  X(12)
    consonantal(12) [ - ]
    sonorant(12) [ + ]
    laryngeal(11)
      glottal(11)
        constr_gl(2) [ + ]
        voiced(11) [ - ]

```


1.	èfɔ̃	+	è	+	mèndzɔ̃	→	èfɔ̃ mèndzɔ̃	‘chief of leopards’
2.	èfɔ̃	+	è	+	mèŋkùɔ̃ʔ°	→	èfɔ̃ mèŋkùɔ̃ʔ°	‘chief of roosters’
3.	èfɔ̃	+	è	+	mèmbhú`	→	èfɔ̃ mèm¹bhú	‘chief of dogs’
4.	èfɔ̃	+	è	+	mètsɔ̃ŋ	→	èfɔ̃ mètsɔ̃ŋ	‘chief of thieves’
5.	̀ndzà´	+	è	+	mèndzɔ̃	→	̀ndzà¹à mèndzɔ̃	‘axe of leopards’
6.	̀ndzà´	+	è	+	mèŋkùɔ̃ʔ°	→	̀ndzà¹à mèŋkùɔ̃ʔ°	‘axe of roosters’
7.	̀ndzà´	+	è	+	mèmbhú`	→	̀ndzà¹à mèm¹bhú	‘axe of dogs’
8.	̀ndzà´	+	è	+	mètsɔ̃ŋ	→	̀ndzà¹à mètsɔ̃ŋ	‘axe of thieves’
9.	̀ndɔ̃ŋ`	+	è	+	mèndzɔ̃	→	̀n¹dɔ̃ŋ mèndzɔ̃	‘horn of leopards’
10.	̀ndɔ̃ŋ`	+	è	+	mèŋkùɔ̃ʔ°	→	̀n¹dɔ̃ŋ mèŋkùɔ̃ʔ°	‘horn of roosters’
11.	̀ndɔ̃ŋ`	+	è	+	mèmbhú`	→	̀n¹dɔ̃ŋ mèm¹bhú	‘horn of dogs’
12.	̀ndɔ̃ŋ`	+	è	+	mètsɔ̃ŋ	→	̀n¹dɔ̃ŋ mètsɔ̃ŋ	‘horn of thieves’
13.	̀jɪní	+	è	+	mèndzɔ̃	→	̀jɪnɪ mèndzɔ̃	‘machete of leopards’
14.	̀jɪní	+	è	+	mèŋkùɔ̃ʔ°	→	̀jɪnɪ mèŋkùɔ̃ʔ°	‘machete of roosters’
15.	̀jɪní	+	è	+	mèmbhú`	→	̀jɪnɪ mèm¹bhú	‘machete of dogs’
16.	̀jɪní	+	è	+	mètsɔ̃ŋ	→	̀jɪnɪ mètsɔ̃ŋ	‘machete of thieves’
17.	̀àzɔ̃b	+	á	+	mèndzɔ̃	→	̀àzɔ̃b ɔ̃ mèn¹dzɔ̃	‘song of leopards’
18.	̀àzɔ̃b	+	á	+	mèŋkùɔ̃ʔ°	→	̀àzɔ̃b ɔ̃ mèŋ¹kùɔ̃ʔ°	‘song of roosters’
19.	̀àzɔ̃b	+	á	+	mèmbhú`	→	̀àzɔ̃b ɔ̃ ¹mómbhú	‘song of dogs’
20.	̀àzɔ̃b	+	á	+	mètsɔ̃ŋ	→	̀àzɔ̃b ɔ̃ ¹mótsɔ̃ŋ	‘song of thieves’
21.	̀àlèŋ´	+	á	+	mèndzɔ̃	→	̀àlèŋ ɔ̃ mèn¹dzɔ̃	‘stool of leopards’
22.	̀àlèŋ´	+	á	+	mèŋkùɔ̃ʔ°	→	̀àlèŋ ɔ̃ mèŋ¹kùɔ̃ʔ°	‘stool of roosters’
23.	̀àlèŋ´	+	á	+	mèmbhú`	→	̀àlèŋ ɔ̃ ¹mómbhú	‘stool of dogs’
24.	̀àlèŋ´	+	á	+	mètsɔ̃ŋ	→	̀àlèŋ ɔ̃ ¹mótsɔ̃ŋ	‘stool of thieves’
25.	̀àláʔ`	+	á	+	mèndzɔ̃	→	̀à¹láʔ á méndzɔ̃	‘country of leopards’
26.	̀àláʔ`	+	á	+	mèŋkùɔ̃ʔ°	→	̀à¹láʔ á méŋkùɔ̃ʔ°	‘country of roosters’
27.	̀àláʔ`	+	á	+	mèmbhú`	→	̀à¹láʔ á móm¹bhú	‘country of dogs’
28.	̀àláʔ`	+	á	+	mètsɔ̃ŋ	→	̀à¹láʔ á mó¹tsɔ̃ŋ	‘country of thieves’
29.	̀àsáŋ	+	á	+	mèndzɔ̃	→	̀àsáŋ á méndzɔ̃	‘tail of leopards’
30.	̀àsáŋ	+	á	+	mèŋkùɔ̃ʔ°	→	̀àsáŋ á méŋkùɔ̃ʔ°	‘tail of roosters’
31.	̀àsáŋ	+	á	+	mèmbhú`	→	̀àsáŋ á mém¹bhú	‘tail of dogs’
32.	̀àsáŋ	+	á	+	mètsɔ̃ŋ	→	̀àsáŋ á mó¹tsɔ̃ŋ	‘tail of thieves’

Table 3.1: 32 tone combinations of bisyllabic nouns in $N_1 + N_2$ associative constructions

1.	èfō	+	è	+	nà	→	èfō nà	‘chief of animal’
2.	èfō	+	è	+	kàŋ´	→	èfō kàŋ°	‘chief of squirrel’
3.	èfō	+	è	+	Ṽmó`	→	èfō ´mó	‘chief of child’
4.	èfō	+	è	+	sóŋ	→	èfō sóŋ	‘chief of bird’
5.	̀ndzà´	+	è	+	nà	→	̀ndzà ´nà	‘axe of animal’
6.	̀ndzà´	+	è	+	kàŋ´	→	̀ndzà ´kàŋ°	‘axe of squirrel’
7.	̀ndzà´	+	è	+	Ṽmó`	→	̀ndzà´à ´mó	‘axe of child’
8.	̀ndzà´	+	è	+	sóŋ	→	̀ndzà´á sóŋ	‘axe of bird’
9.	̀ndóŋ`	+	è	+	nà	→	̀n´dòŋ nà	‘horn of animal’
10.	̀ndóŋ`	+	è	+	kàŋ´	→	̀n´dòŋ kàŋ°	‘horn of squirrel’
11.	̀ndóŋ`	+	è	+	Ṽmó`	→	̀n´dòŋ ´mó	‘horn of child’
12.	̀ndóŋ`	+	è	+	sóŋ	→	̀n´dòŋ sóŋ	‘horn of bird’
13.	̀j̀nɪ´	+	è	+	nà	→	̀j̀nɪ nà	‘machete of animal’
14.	̀j̀nɪ´	+	è	+	kàŋ´	→	̀j̀nɪ kàŋ°	‘machete of squirrel’
15.	̀j̀nɪ´	+	è	+	Ṽmó`	→	̀j̀nɪ ´mó	‘machete of child’
16.	̀j̀nɪ´	+	è	+	sóŋ	→	̀j̀nɪ ´sóŋ	‘machete of bird’
17.	̀àzòb	+	á	+	nà	→	̀àzòb ò nà	‘song of animal’
18.	̀àzòb	+	á	+	kàŋ´	→	̀àzòb ò kàŋ°	‘song of squirrel’
19.	̀àzòb	+	á	+	Ṽmó`	→	̀àzòb ´ó mó	‘song of child’
20.	̀àzòb	+	á	+	sóŋ	→	̀àzòb ò sóŋ	‘song of bird’
21.	̀àlèŋ´	+	á	+	nà	→	̀àlèŋ è nà	‘stool of animal’
22.	̀àlèŋ´	+	á	+	kàŋ´	→	̀àlèŋ è kàŋ°	‘stool of squirrel’
23.	̀àlèŋ´	+	á	+	Ṽmó`	→	̀àlèŋ ´é mó	‘stool of child’
24.	̀àlèŋ´	+	á	+	sóŋ	→	̀àlèŋ è sóŋ	‘stool of bird’
25.	̀àlá?`	+	á	+	nà	→	̀à´lá? á nà	‘country of animal’
26.	̀àlá?`	+	á	+	kàŋ´	→	̀à´lá? á kàŋ°	‘country of squirrel’
27.	̀àlá?`	+	á	+	Ṽmó`	→	̀à´lá? á ´mó	‘country of child’
28.	̀àlá?`	+	á	+	sóŋ	→	̀à´lá? á sóŋ	‘country of bird’
29.	̀àsáŋ	+	á	+	nà	→	̀àsáŋ á nà	‘tail of animal’
30.	̀àsáŋ	+	á	+	kàŋ´	→	̀àsáŋ á kàŋ°	‘tail of squirrel’
31.	̀àsáŋ	+	á	+	Ṽmó`	→	̀àsáŋ á ´mó	‘tail of child’
32.	̀àsáŋ	+	á	+	sóŋ	→	̀àsáŋ á sóŋ	‘tail of bird’

Table 3.2: 32 tone combinations of associative bisyllabic N_1 + monosyllabic N_2

Chapter 4

Conclusions

In this chapter we evaluate the system relative to the goals set forth in chapter 1. We then discuss possible future work.

4.1 System Evaluation

Recall that we began the project with five goals. We address each in turn.

4.1.1 Coverage

We feel that the most important contribution the system makes is its broad coverage of the range of analyses that phonologists write. We picked several examples that we felt would be quite difficult to implement *as analyzed by researchers*. We have no doubt that more limited systems could handle much of the data, but not with the same analysis the linguists employed. **Pāṇini** does not require the phonologist to significantly alter an analysis for the sake of implementation — we feel that this makes it a good tool.

However, there are some important phenomena that **Pāṇini** cannot handle. The major one is stress. Since there are many different theories, most of which are totally unlike theories governing the rest of phonology, we felt that stress was better left for a later incarnation of the system.

Pāṇini also cannot handle complex syllabification problems. Such phenomena are not yet fully understood, but it seems clear that phonologists need some new mechanisms to explain data from Berber (Dell and Tangi 1991) (Dell and Elmedlaoui 1985). Involved resyllabification analyses like Harris (1990) proposes for Spanish are also more difficult to implement than we would like.

Finally, **Pāṇini** probably needs an effect that specifies that a particular floating element is to be moved

into another node’s domain. For example, it may prove crucial to be able to express rules like “move the floating high tone into the next word”. Currently, the only way to change a floating element’s domain is to link it to a new parent and then immediately delink it. This solution is not only odious but insufficient, since this only works for domains that the floating node can link *directly* to.

4.1.2 Ease of use

We feel that we have met this goal given the confines of ASCII-only input. The notation we developed for **Pāṇini** is straightforward, though we suspect that the node numbering may take some getting used to. To make the system much easier to use will require “upgrading” to a windowing environment that allows input and match elements to be built up on the screen interactively, using exactly the same kinds of diagrams that phonologists use (suitably extended for the extra degree of precision we require of our rules).

4.1.3 Flexibility

We are surprised at how little we had to hard-code into **Pāṇini**. **Pāṇini** will handle any feature tree (an arbitrary DAG), any number of features, elements of any size, arbitrarily large rule sets, etc. Available memory is the only factor limiting the major components of the system. In this regard we feel that the system is very successful.

However, since we did have to assume a rule-based approach to phonology, the system is incompatible with current research in constraint-based approaches. In addition, because **Pāṇini** assumes a multi-dimensional feature DAG model, any analysis that assumes a radically different representation will not be implementable with **Pāṇini**. (We know of no such alternatives with any significant support in the phonological community, however.)

4.1.4 Efficiency

We are less pleased with the performance of the system. Although it is within our stated speed bounds (at most several seconds per form), we only achieve these results on fairly fast machines.¹

We suspect that system performance could be significantly improved with better memory management. **Pāṇini** tends to allocate huge numbers of pointers to small chunks of memory — things like lists of child and parent nodes that are small but which we do not wish to place any hard-coded size bound on. Worse, dynamically resize these arrays. We believe that some simple changes to the system could greatly reduce the amount of time the program spends doing memory management.

More worrisome is the theoretically exponential time bound on the matcher. We used a dynamic programming approach to ensure polynomial time performance for fixed length match patterns, but were

¹150Mhz Silicon Graphics Indigo workstations.

forced to resort to an exponential backtracking algorithm for regular expression matching.² This means that excessive use of the * and + operators can cause the matcher to “go away and never come back”. In practice, however, we have found no analyses that require heavy use of these operators, and, as we have already noted, the theory predicts that these operators should not be necessary at all. We mainly use them to implement unusual stipulations on rule application or to solve technical problems implementing trickier rules like the OCP.

4.1.5 Portability

Pāṇini is written in YACC (Johnson 1986), Lex (Lesk 1986), and C (Kernighan and Ritchie 1988). We found YACC surprisingly easy to use — it made specification and subsequent modification of the input file format simple.

These are all standard Unix tools that require no special run-time environment. We have taken care not to rely on specific integer or pointer sizes, structure layout details, or any other machine-dependent behavior. As a consequence, our code compiles and runs without modification on a variety of systems.³

Pāṇini assumes no special workstation capabilities. It should run without major changes on PC’s and Macintoshes as well.

4.2 Future Work

There are many possible directions to take from here. One possibility is to explore the new constraint-based approaches to phonology. These theories change the way surface representations are generated, but assume the same feature tree representation that we have already implemented in **Pāṇini**.

Another idea is to explore applying Maxwell’s (1994) underspecification technique for segmental phonological parsing to autosegmental phonology, using **Pāṇini** as a test bed.

Finally, we hope to improve **Pāṇini**, so that it will be able to handle stress and syllabification properly. We also intend to look into making the program faster, with the goal of achieving real-time generation of surface representations that could be fed to an articulator-based speech synthesis system.

²We represent regular expressions with nondeterministic finite automata. In theory these could be converted into equivalent deterministic automata that we could match against in polynomial time. However, because our match specifications deal with multiple tiers, the alphabet size is too large to permit this given memory limitations.

³We have extensively tested the Sparc and SGI versions. We expect that the code will run on any Unix system.

Appendix A

Complete Rulesets for all Examples

This appendix lists the complete text of each ruleset, including the definitions of the lexicon entries. Each example is explained in a corresponding section in chapter 3.

A.1 Standard Definitions

The following two files define the feature geometry and phonemes used in our examples. The first, `std.ph`, defines an X-slot model; the second, `cv.ph`, defines a CV model.

```
/*
 * std.ph
 *
 * Standard feature geometry and phoneme definitions
 */
geometry {
    //
    // Modified Halle-Sagey Articulator Model with X-slot model syllables.
    //
    word -> morpheme,
    morpheme -> sigma,
    sigma -> stress [5 4 3 2 1],
    sigma -> onset,
    sigma -> rhyme,
    rhyme -> nucleus,
    rhyme -> coda,
    onset -> X,
    nucleus -> X,
    coda -> X,
    X -> consonantal [- +],
    consonantal -> sonorant [- +],
    sonorant -> continuant [- +],
    sonorant -> strident [- +],
    sonorant -> lateral [- +],
```

```

sonorant -> laryngeal,
sonorant -> pharyngeal,           // no subtree here by default
laryngeal -> glottal,
laryngeal -> tongue_root,
glottal -> voiced [- +],
glottal -> slack_vf [- +],
glottal -> spread_gl [- +],
glottal -> constr_gl [- +],
tongue_root -> atr [- +],
tongue_root -> rtr [- +],
sonorant -> supralaryngeal,
supralaryngeal -> soft_palette,
supralaryngeal -> oral_place,
soft_palette -> nasal [- +],
oral_place -> labial,
oral_place -> coronal,
oral_place -> dorsal,
labial -> round [- +],
coronal -> anterior [- +],
coronal -> distributed [- +],
dorsal -> high [- +],
dorsal -> low [- +],
dorsal -> back [- +],

// Tonal features
X -> T,
T -> register [0 up down],
T -> modal [L M H],

// Pseudo-features used to drive rules
word -> plural [- +],
word -> compound [- +],
word -> perfect [- +],
word -> imperfect [- +],
word -> progressive [- +],
}

//
// Base forms
//
phoneme C      [ X +consonantal ]
phoneme V      [ X -consonantal +sonorant dorsal -high -low -back -round -atr +voiced]

//
// consonants
//
phoneme p      [ C -sonorant -continuant labial -voiced ]
phoneme b      [ p +voiced ]
phoneme k      [ C -sonorant -continuant dorsal -voiced ]
phoneme g      [ k +voiced ]
phoneme x      [ k +continuant ]
phoneme t      [ C -sonorant -continuant +anterior -voiced ]
phoneme d      [ t +voiced ]
phoneme f      [ C -sonorant +continuant labial -voiced ]
phoneme v      [ f +voiced ]
phoneme theta  [ C -sonorant +continuant +anterior -strident -voiced ]
phoneme eht    [ theta +voiced ]
phoneme dh     [ eth ]
phoneme s      [ C -sonorant +continuant +anterior +strident -voiced ]
phoneme z      [ s +voiced ]
phoneme sh     [ C -sonorant +continuant +strident +distributed -voiced ]

```

```

phoneme zh      [ sh +voiced ]

// affricates
phoneme ts      [ C -sonorant -continuant#1 +continuant#2 +anterior -voiced ]
phoneme dz      [ ts +voiced ]
phoneme ch      [ ts -anterior ]
phoneme dj      [ ch +voiced ]

// liquids
phoneme l       [ C +sonorant +lateral +anterior +voiced ]
phoneme r       [ C +sonorant +anterior +voiced ]

// nasals
phoneme m       [ C +sonorant -continuant +nasal labial +voiced ]
phoneme n       [ C +sonorant -continuant +anterior +nasal +voiced ]
phoneme ng      [ C +sonorant -continuant -anterior +nasal dorsal +voiced ]
phoneme ny      [ C +sonorant -continuant +anterior +nasal +voiced +distributed ]

// glides etc.
phoneme y       [ X -consonantal +sonorant +continuant +voiced ]
phoneme w       [ y +round ]
phoneme h       [ X -consonantal +sonorant +spread_gl -voiced ]
phoneme gstop   [ X -consonantal +sonorant +constr_gl -voiced ]
phoneme gs      [ gstop ]

//
// vowels
//
phoneme i       [ V +high +atr ]
phoneme I       [ V +high ]
phoneme i_bar   [ I ~back ]
phoneme e       [ V +atr ]
phoneme eh      [ V ]
phoneme epsilon [ eh ]
phoneme ae      [ V +low ]
phoneme u       [ V +back +atr +round ]
phoneme u_bar   [ u ~back ]
phoneme U       [ V +high +back +round ]
phoneme o       [ V +back +atr +round ]
phoneme schwa   [ V +back ]
phoneme a       [ V +low +back +atr ]
phoneme aw      [ V +low +back +atr +round ]

//
// examples of common modifications
//
phoneme p_h     [ p +spread_gl ]           // aspirated p
phoneme p_g     [ p +constr_gl ]          // glottalized p
phoneme k_w     [ k +round ]              // labial k

```



```

/*
 * cv.ph
 *
 * CV feature geometry
 */
geometry {
  //
  // Modified Halle-Sagey Articulator Model with CV model syllables.
  //
  word -> morpheme,
  morpheme -> sigma,
  sigma -> stress [5 4 3 2 1],
  sigma -> onset,
  sigma -> rhyme,
  rhyme -> nucleus,
  rhyme -> coda,
  onset -> X,
  nucleus -> X,
  coda -> X,
  X -> V,
  X -> C,
  V -> consonantal [- +],
  C -> consonantal [- +],
  consonantal -> sonorant [- +],
  sonorant -> continuant [- +],
  sonorant -> strident [- +],
  sonorant -> lateral [- +],
  sonorant -> laryngeal,
  sonorant -> pharyngeal,           // no subtree here by default
  laryngeal -> glottal,
  laryngeal -> tongue_root,
  glottal -> voiced [- +],
  glottal -> slack_vf [- +],
  glottal -> spread_gl [- +],
  glottal -> constr_gl [- +],
  tongue_root -> atr [- +],
  tongue_root -> rtr [- +],
  sonorant -> supralaryngeal,
  supralaryngeal -> soft_palette,
  supralaryngeal -> oral_place,
  soft_palette -> nasal [- +],
  oral_place -> labial,
  oral_place -> coronal,
  oral_place -> dorsal,
  labial -> round [- +],
  coronal -> anterior [- +],
  coronal -> distributed [- +],
  dorsal -> high [- +],
  dorsal -> low [- +],
  dorsal -> back [- +],

  // Tonal features
  X -> T,
  T -> register [0 up down],
  T -> modal [L M H],

  // Pseudo-features used to drive rules
  word -> plural [- +],
  word -> compound [- +],
  word -> perfect [- +],
  word -> imperfect [- +],

```

```

        word -> progressive [- +],
    }

//
// Base forms
//
phoneme Cons    [ X C +consonantal ]
phoneme Vowel  [ X V -consonantal +sonorant dorsal -high -low -back -round -atr +voiced]

//
// consonants
//
phoneme p      [ Cons -sonorant -continuant labial -voiced ]
phoneme b      [ p +voiced ]
phoneme k      [ Cons -sonorant -continuant dorsal -voiced ]
phoneme g      [ k +voiced ]
phoneme x      [ k +continuant ]
phoneme t      [ Cons -sonorant -continuant +anterior -voiced ]
phoneme d      [ t +voiced ]
phoneme f      [ Cons -sonorant +continuant labial -voiced ]
phoneme v      [ f +voiced ]
phoneme theta  [ Cons -sonorant +continuant +anterior -strident -voiced ]
phoneme eth    [ theta +voiced ]
phoneme dh     [ thorn ]
phoneme s      [ Cons -sonorant +continuant +anterior +strident -voiced ]
phoneme z      [ s +voiced ]
phoneme sh     [ Cons -sonorant +continuant +strident +distributed -voiced ]
phoneme zh     [ sh +voiced ]

// affricates
phoneme ts     [ Cons -sonorant -continuant#1 +continuant#2 +anterior -voiced ]
phoneme dz     [ ts +voiced ]
phoneme ch     [ ts -anterior ]
phoneme dj     [ ch +voiced ]

// liquids
phoneme l      [ Cons +sonorant +lateral +anterior +voiced ]
phoneme r      [ Cons +sonorant +anterior +voiced ]

// nasals
phoneme m      [ Cons +sonorant -continuant +nasal labial +voiced ]
phoneme n      [ Cons +sonorant -continuant +anterior +nasal +voiced ]
phoneme ng     [ Cons +sonorant -continuant -anterior +nasal dorsal +voiced ]

// glides etc.
phoneme y      [ X C -consonantal +sonorant +continuant +voiced ]
phoneme w      [ y +round ]
phoneme h      [ X C -consonantal +sonorant +spread_gl -voiced ]
phoneme gstop  [ X C -consonantal +sonorant +constr_gl -voiced ]
phoneme gs     [ gstop ]

//
// vowels
//
phoneme i      [ Vowel +high +atr ]
phoneme I      [ Vowel +high ]
phoneme i_bar  [ I ~back ]
phoneme e      [ Vowel +atr ]
phoneme eh     [ Vowel ]
phoneme epsilon [ eh ]
phoneme ae     [ Vowel +low ]

```

```
phoneme u      [ Vowel +back +atr +round ]
phoneme u_bar  [ u ~back ]
phoneme U      [ Vowel +high +back +round ]
phoneme o      [ Vowel +back +atr +round ]
phoneme schwa  [ Vowel +back ]
phoneme a      [ Vowel +low +back +atr ]
phoneme aw     [ Vowel +low +back +atr +round ]

//
// examples of common modifications
//
phoneme p_h    [ p +spread_gl ]      // aspirated p
phoneme p_g    [ p +constr_gl ]     // glottalized p
phoneme k_w    [ k +round ]         // labial k
```

A.2 The English Plural

```

/*
 * English Plural
 */
#include "std.ph"

/*
 * [Kenstowicz 1994, p. 500]
 *
 * The distribution of the English plural represents a rule sensitive to
 * the right side of an affricate: [i-z] becomes [z] (and then [s] by
 * voicing assimilation) unless deletion would create successive
 * [+continuant] segments sharing the Coronal articulator. Deletion is
 * thus possible in cab-[z], cuff-[s], and cat-[s] but not in buss-[iz]
 * and brush-[iz]. The fact that deletion is blocked after the coronal
 * affricates [ch] and [dj] now makes sense: they terminate in [+continuant]
 * and hence would give rise to successive [+continuant] specifications:
 * judg-[iz], church-[iz].
 */

//
// Lexicon entries
//
lexicon "cab" [
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 ae
      coda #1 b
]
lexicon "cuff" [
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 schwa
      coda #1 f
]
lexicon "cat" [
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 ae
      coda #1 t
]
lexicon "dog" [
  morpheme
    sigma#1
      onset#1 d
      nucleus#1 aw
      coda #1 g
]
lexicon "bus" [
  morpheme
    sigma#1
      onset#1 b
      nucleus#1 schwa
      coda #1 s
]
lexicon "brush" [

```

```

morpheme
  sigma#1
    onset#1  b r
    nucleus#1 schwa
    coda #1  sh
]
lexicon "judge" [
  morpheme
    sigma#1
      onset#1  dj
      nucleus#1 schwa
      coda #1  dj
]
lexicon "church" [
  morpheme
    sigma#1
      onset#1  ch
      nucleus#1 r
      coda #1  ch
]

//
// Rules
//
block "plural":
  rule "insert plural suffix":
    match: [word +plural final X within word]#1
    effect: insert [z ~voiced] after X#1
    ;

  // This epenthesis is probably a language-wide fixup process, not
  // peculiar to pluralization:
  rule "insert vowel":
    match: [word +plural X#1 within word +continuant#1 coronal#1
            final X#2 within word +continuant#2 coronal#2 ]#1
    effect: insert [i_bar] before X#1:2
    ;

  rule "assimilate voicing":
    match: [word +plural X#1 within word glottal#1 voiced#1
            final X#2 within word glottal#2 ]#1
    effect: link voiced#1:1 to glottal#1:2
    ;

  // resyllabify here
  ;

//
// Samples
//
generate "cabs" [word +plural "cab"]
generate "cuff" [word +plural "cuff"]
generate "cats" [word +plural "cat"]
generate "dogs" [word +plural "dog"]
generate "bus" [word +plural "bus"]
generate "brushes" [word +plural "brush"]
generate "judges" [word +plural "judge"]
generate "churches" [word +plural "church"]

```

A.3 Margi Contour Tones

```

/*
 * Margi Counter Tones
 *
 * [Hoffman 1963]
 *
 * See [Kenstowicz 1994, p. 317] for an explanation of this account.
 */
#include "std.ph"

//
// Add tonal root node to vowels.
// This makes them tone-bearing units (TBU's).
//
append phoneme i      [ T ]
append phoneme u      [ T ]
append phoneme o      [ T ]
append phoneme schwa  [ T ]
append phoneme a      [ T ]

//
// The u phoneme is actually higher than we define it in the
// standard inventory, but this doesn't affect the tonal phenomena
// we're modeling here.
//
lexicon "cu" [          // 'speak'
  morpheme
    sigma#1
      onset#1  ts
      nucleus#1 u
      unlinked H
]
lexicon "gha" [        // 'reach'
  morpheme
    sigma#1
      onset#1  g h
      nucleus#1 a
      unlinked L
]
lexicon "fi" [         // 'swell'
  morpheme
    sigma#1
      onset#1  f
      nucleus#1 i
      unlinked L#1
      unlinked H#2
]
lexicon "sa" [         // 'go astray'
  morpheme
    sigma#1
      onset#1  s
      nucleus#1 a
      unlinked H
]
lexicon "dla" [        // 'fall'
  morpheme
    sigma#1
      onset#1  d l
      nucleus#1 a

```

```

                unlinked L
]
lexicon "bdlu" [           // 'forge'
  morpheme
    sigma#1
      onset#1  b d l
      nucleus#1 u
                unlinked L#1
                unlinked H#2
]
lexicon "ndabya" [        // 'touch'
  morpheme
    sigma#1
      onset#1  n d
      nucleus#1 a
                unlinked H#1
    sigma#2
      onset#2  b y
      nucleus#2 a
]
lexicon "mbidu" [        // 'blow'
  morpheme
    sigma#1
      onset#1  m b
      nucleus#1 i
                unlinked L#1
    sigma#2
      onset#2  d
      nucleus#2 u
                unlinked H#1
]
lexicon "ulu" [          // 'see'
  morpheme
    sigma#1
      nucleus#1 u
                unlinked L#1
    sigma#2
      onset#2  l
      nucleus#2 u
]

//
// So-called "changing verbs".
// These have no tone underlyingly.
//
lexicon "fa" [           // 'take'
  morpheme
    sigma#1
      onset#1  f
      nucleus#1 a
]
lexicon "hu" [           // 'take'
  morpheme
    sigma#1
      onset#1  h
      nucleus#1 u
]

//
// Affixes
//

```

```

lexicon "na" [                               // '(verbal suffix)'
  morpheme
  sigma#1
  onset#1  n
  nucleus#1 a
  // no underlying tone
]
lexicon "ba" [                               // '(verbal suffix)'
  morpheme
  sigma#1
  onset#1  b
  nucleus#1 a
  unlinked H
]
lexicon "[ng]g[schwa]ri" [                 // '(verbal suffix)'
  morpheme
  sigma#1
  onset#1  ng
  nucleus#1 schwa
  sigma#2
  onset#2  r
  nucleus#2 i
  unlinked H
]

//
// Rules
//
rule "association convention":
  iterative

  match: [X T !modal]#1 [unlinked modal]#2
  effect: link modal#2 to T#1
;
rule "docking":
  iterative
  rightmost

  match: [X T] [unlinked modal]
  effect: link modal to T
;
rule "spreading":
  iterative
  match matched

  match: [T modal]#1 [T !modal]#2
  effect: link modal#1 to T#2
;
rule "default tone":
  iterative

  match: [T !modal]
  effect: insert [L] under T
;
rule "OCP":
  iterative
  match matched
  no warnings

  match: ([T H]#1 [unlinked H]* [shared T H]#2) | ([T L]#3 [unlinked H]* [shared T L]#4)
  effect:

```



```

// case 1: successive high tones:
delink H#2 from T#2
link H#1 to T#2

// case 2: successive low tones:
delink L#4 from T#4
link L#3 to T#4
;
rule "stray erasure":
  iterative
  match: [unlinked modal]
  effect: stray erase modal
;

//
// Samples
//
generate "ciba 'tell'" [word "cu" "ba"] // -> HH
generate "ghaba 'reach'" [word "gha" "ba"] // -> LH
generate "fiba 'make swell'" [word "fi" "ba"] // -> LH

generate "sana 'lead astray'" [word "sa" "na"] // -> HH
generate "dlana 'overthrow'" [word "dla" "na"] // -> LL
generate "bdluna 'forge'" [word "bdlu" "na"] // -> LH (actually -> bdl[schwa]L naH)

generate "fa[ng]g[schwa]ri 'take many'"
  [word "fa" "[ng]g[schwa]ri"] // -> HHH

generate "hu" [word "hu"] // -> L

```

A.4 Sudanese Place Assimilation

```

/*
 * Sudanese Arabic Imperfect
 * Point of articulation assimilation
 *
 * [Hamid 1984]
 */
#include "std.ph"

/*
 * [Kenstowicz 1994, p. 158]
 *
 * The presence of the cavity nodes in the feature tree ... is motivated
 * more on phonetic than on phonological grounds. However, it is clear
 * that many phonological processes single out the Labial, Coronal, and
 * Dorsal articulators. For example, in Sudanese Arabic the coronal
 * nasal [n] assimilates the point of articulation of the following
 * consonant, becoming the labial [m] before [b], the coronal [n] before
 * [z], and the velar [ɛŋma] before [k].
 */
phoneme pharyngeal_h [h pharyngeal]
phoneme pharyngeal_stop [gstop pharyngeal]

lexicon "bark" [
  morpheme
    sigma#1
      onset#1 n
      nucleus#1 a
    sigma#2
      onset#2 b
      nucleus#2 a
      coda#2 pharyngeal_h
  ]
lexicon "save" [
  morpheme
    sigma#1
      onset#1 n
      nucleus#1 a
    sigma#2
      onset#2 f
      nucleus#2 a
      coda#2 d
  ]
lexicon "descend" [
  morpheme
    sigma#1
      onset#1 n
      nucleus#1 a
    sigma#2
      onset#2 z
      nucleus#2 a
      coda#2 l
  ]
lexicon "demolish" [
  morpheme
    sigma#1
      onset#1 n
      nucleus#1 a
    sigma#2

```

```

        onset#2  s
        nucleus#2 a
        coda#2   f
    ]
lexicon "spread" [
    morpheme
    sigma#1
        onset#1  n
        nucleus#1 a
    sigma#2
        onset#2  sh
        nucleus#2 a
        coda#2   r
    ]
lexicon "succeed" [
    morpheme
    sigma#1
        onset#1  n
        nucleus#1 a
    sigma#2
        onset#2  dj
        nucleus#2 a
        coda#2   pharyngeal_h
    ]
lexicon "deny" [
    morpheme
    sigma#1
        onset#1  n
        nucleus#1 a
    sigma#2
        onset#2  k
        nucleus#2 a
        coda#2   r
    ]
lexicon "puncture" [
    morpheme
    sigma#1
        onset#1  n
        nucleus#1 a
    sigma#2
        onset#2  x
        nucleus#2 a
        coda#2   r
    ]
lexicon "transfer" [
    morpheme
    sigma#1
        onset#1  n
        nucleus#1 a
    sigma#2
        onset#2  g
        nucleus#2 a
        coda#2   l
    ]
lexicon "slaughter" [
    morpheme
    sigma#1
        onset#1  n
        nucleus#1 a
    sigma#2
        onset#2  pharyngeal_h
    ]

```

```

        nucleus#2 a
        coda#2  r
    ]
lexicon "fall asleep" [
    morpheme
    sigma#1
    onset#1  n
    nucleus#1 i
    sigma#2
    onset#2  pharyngeal_stop
    nucleus#2 i
    coda#2  s
]
lexicon "rob" [
    morpheme
    sigma#1
    onset#1  n
    nucleus#1 a
    sigma#2
    onset#2  h
    nucleus#2 a
    coda#2  b
]

//
// Rules
//
block "imperfect":
    rule "delete first vowel":
        //
        // We find the first vowel in the word and delete the X
        // slot it's linked to.
        //
        // The match specification is a bit roundabout because
        // we want to include the X in the match but don't know where
        // it will be in the word -- it might be initial or medial,
        // and might be in the onset or the nucleus of the first syllable.
        //
        // Note that "-consonantal within X" places an additional
        // restriction on the previously declared -consonantal node;
        // it does not introduce a new node. (To do that, we'd just
        // have to give the two -consonantals different numbers.)
        //
        match: [word +imperfect
                X within word
                initial -consonantal within word
                -consonantal within X]

        effect: delete X
    ;

    rule "assimilate oral place":
        //
        // Now we spread the second X slot's oral place node to the
        // first X slot and delete the first X slot's oral place
        // subtree.
        //
        // Note that we'll only get a match if the second X slot
        // has an oral place node. This is key, and explains why
        // the pharyngeal [h] and pharyngeal stop do not alter the
        // first segment -- these segments have no oral place
        // specification.

```

```

//
match: [word +imperfect initial X#1 within word supralaryngeal#1 oral_place#1
      X#2 within word oral_place#2 ]#1
effect: delete oral_place#1:1
      link oral_place#1:2 to supralaryngeal#1:1
;

rule "prefix":
//
// Now we add the [ya] prefix.
//
match: [word +imperfect initial sigma within word]#1
effect: insert [sigma onset#1 y nucleus#1 a] before sigma#1
      change imperfect to -
;

// resyllabify here
;

//
// Examples
//
generate "bark (im.)" [word +imperfect "bark"] // -> ya-mbah
generate "save (im.)" [word +imperfect "save"] // -> ya-[mg]fid
generate "descend (im.)" [word +imperfect "descend"] // -> ya-nzil
generate "demolish (im.)" [word +imperfect "demolish"] // -> ya-nsif
generate "spread (im.)" [word +imperfect "spread"] // -> ya-n~shur
generate "succeed (im.)" [word +imperfect "succeed"] // -> ya-n`djah
generate "deny (im.)" [word +imperfect "deny"] // -> ya-[ng]kur
generate "puncture (im.)" [word +imperfect "puncture"] // -> ya-[ng]xar
generate "transfer (im.)" [word +imperfect "transfer"] // -> ya-[ng]gul

//
// Oral place assimilation does not occur in these derivations:
//
generate "slaughter (im.)" [word +imperfect "slaughter"] // -> ya-nhar
generate "fall asleep (im.)" [word +imperfect "fall asleep"] // -> ya-n[pharyngeal_stop]as
generate "rob (im.)" [word +imperfect "rob"] // -> ya-nhab

```

A.5 Japanese Rendaku

```

/*
 * Japanese Rendaku Voicing
 *
 * [Ito and Mester 1986]
 */
#include "std.ph"

/*
 * [Kenstowicz 1994, p. 162]
 *
 * The combination of voiced obstruents within a root avoided in the
 * native Japanese vocabulary. Thus, while voiceless obstruents and
 * voiced and voiceless obstruents combine freely,
 *
 *     futa 'lid'   fuda 'sign'   buta 'pig'
 *
 * voiced ones do not (*buda).
 *
 * The constraint is actively enforced by blocking or undoing an
 * otherwise general rule voicing the initial obstruent in the second
 * member of a compound (Lyman's Law). Thus:
 *
 *     iro 'color' + kami 'paper' -> irogami
 *     kami 'divine' + kaze 'wind' -> kamikaze (*kamigaze)
 *
 * Ito and Mester express the contrast ... as a dissimilation process
 * that deletes the voicing specification inserted in compounds when
 * another one follows in the same morpheme.
 *
 * ...
 *
 * It is worth observing that if the intervening vowels are not
 * underspecified for voicing ... the rule becomes much more complex.
 *
 * ...
 *
 * Mester and Ito try to reconcile the contradictory results from Japanese
 * with the hypothesis that [voiced] is in fact a UG monovalent feature
 * marking just voiced obstruents.
 */

//
// First we remove all the vowels' voicing specifications.
//
append phoneme i      [ ~voiced ]
append phoneme I      [ ~voiced ]
append phoneme e      [ ~voiced ]
append phoneme eh     [ ~voiced ]
append phoneme ae     [ ~voiced ]
append phoneme u      [ ~voiced ]
append phoneme U      [ ~voiced ]
append phoneme o      [ ~voiced ]
append phoneme schwa  [ ~voiced ]
append phoneme a      [ ~voiced ]
append phoneme aw     [ ~voiced ]

//
// Now remove voicing specifications from all but the voiced obstruents.

```

```

//
append phoneme p      [ ~voiced ]
append phoneme k      [ ~voiced ]
append phoneme t      [ ~voiced ]
append phoneme f      [ ~voiced ]
append phoneme s      [ ~voiced ]
append phoneme sh     [ ~voiced ]
append phoneme ts     [ ~voiced ]
append phoneme ch     [ ~voiced ]

//
// Nasals are unspecified for voicing as well.
//
append phoneme m      [ ~voiced ]
append phoneme n      [ ~voiced ]

//
// Lexicon entries
//
lexicon "kami" [      // 'divine', 'paper'
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 a
    sigma#2
      onset#2 m
      nucleus#2 i
  ]
lexicon "kaze" [     // 'wind'
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 a
    sigma#2
      onset#2 z
      nucleus#2 e
  ]
lexicon "iro" [      // 'color'
  morpheme
    sigma#1
      nucleus#1 i
    sigma#2
      onset#2 r
      nucleus#2 o
  ]
lexicon "eda" [      // 'branch'
  morpheme
    sigma#1
      nucleus#1 e
    sigma#2
      onset#2 d
      nucleus#2 a
  ]
lexicon "ke" [       // 'hair'
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 e
  ]
lexicon "unari" [    // 'moan'
  morpheme

```

```

    sigma#1
      nucleus u
    sigma#2
      onset#2 n
      nucleus#2 a
    sigma#3
      onset#3 r
      nucleus#3 i
  ]
lexicon "koe" [ // 'voice'
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 o
    sigma#2
      nucleus#2 e
  ]
lexicon "mizu" [ // 'water'
  morpheme
    sigma#1
      onset#1 m
      nucleus#1 i
    sigma#2
      onset#2 z
      nucleus#2 u
  ]
lexicon "seme" [ // 'torture'
  morpheme
    sigma#1
      onset#1 s
      nucleus#1 e
    sigma#2
      onset#2 m
      nucleus#2 e
  ]
lexicon "ori" [ // 'fold'
  morpheme
    sigma#1
      nucleus#1 o
    sigma#2
      onset#2 r
      nucleus#2 i
  ]
lexicon "neko" [ // 'cat'
  morpheme
    sigma#1
      onset#1 n
      nucleus#1 e
    sigma#2
      onset#2 k
      nucleus#2 o
  ]
lexicon "sita" [ // 'tongue'
  morpheme
    sigma#1
      onset#1 sh
      nucleus#1 i
    sigma#2
      onset#2 t
      nucleus#2 a
  ]
]
```



```

lexicon "kita" [           // 'north'
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 i
    sigma#2
      onset#2 t
      nucleus#2 a
  ]
lexicon "siro" [          // 'white'
  morpheme
    sigma#1
      onset#1 sh
      nucleus#1 i
    sigma#2
      onset#2 r
      nucleus#2 o
  ]
lexicon "tabi" [          // 'tabi'
  morpheme
    sigma#1
      onset#1 t
      nucleus#1 a
    sigma#2
      onset#2 b
      nucleus#2 i
  ]
lexicon "taikutsu" [     // 'time'
  morpheme
    sigma#1
      onset#1 t
      nucleus#1 a
    sigma#2
      nucleus#2 i
    sigma#3
      onset#3 k
      nucleus#3 u
    sigma#4
      onset#4 ts
      nucleus#4 u
  ]
lexicon "sinogi" [       // 'avoiding'
  morpheme
    sigma#1
      onset#1 sh
      nucleus#1 i
    sigma#2
      onset#2 n
      nucleus#2 o
    sigma#3
      onset#3 g
      nucleus#3 i
  ]
lexicon "onna" [         // 'woman'
  morpheme
    sigma#1
      nucleus#1 o
      coda#1 n
    sigma#2
      onset#2 n
      nucleus#2 a
  ]

```

```

]
lexicon "kotoba" [      // 'words'
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 o
    sigma#2
      onset#2 t
      nucleus#2 o
    sigma#3
      onset#3 b
      nucleus#3 a
]
lexicon "doku" [      // 'poison'
  morpheme
    sigma#1
      onset#1 d
      nucleus#1 o
    sigma#2
      onset#2 k
      nucleus#2 u
]
lexicon "tokage" [    // 'lizard'
  morpheme
    sigma#1
      onset#1 t
      nucleus#1 o
    sigma#2
      onset#2 k
      nucleus#2 a
    sigma#3
      onset#3 g
      nucleus#3 e
]
lexicon "ko" [        // 'child'
  morpheme
    sigma#1
      onset#1 k
      nucleus#1 o
]
lexicon "tanuki" [    // 'raccoon'
  morpheme
    sigma#1
      onset#1 t
      nucleus#1 a
    sigma#2
      onset#2 n
      nucleus#2 u
    sigma#3
      onset#3 k
      nucleus#3 i
]

//
// Rules
//
rule "compounding":
  no warnings

  match [word +compound morpheme#1 morpheme#2 initial glottal within morpheme#2 optional voiced]
  effects:

```

```

//
// Delete the voiced node if there already is one.
// This rule will not apply if no voiced node got matched.
//
delete voiced
insert [+voiced] under glottal
change compound to -
;

rule "lyman's law":
  iterative

  match: [morpheme +voiced#1 within morpheme +voiced#2 within morpheme]#1
  effect: delete voiced #1:1
;

rule "feature-fill voicing (sonorants)":
  iterative

  match: [+sonorant glottal !voiced]
  effect: insert [+voiced] under glottal
;

rule "feature-fill voicing (obstruents)":
  iterative

  match: [-sonorant glottal !voiced]
  effect: insert [-voiced] under glottal
;

//
// Samples
//
generate "split hair (edage)"
  [word +compound "eda" "ke"] // -> edage
generate "groan (unarigoe)"
  [word +compound "unari" "koe"] // -> unarigoe
generate "water torture (mizuzeme)"
  [word +compound "mizu" "seme"] // -> mizuzeme
generate "origami paper (origami)"
  [word +compound "ori" "kami"] // -> origami
generate "aversion to hot food (nekojita)"
  [word +compound "neko" "sita"] // -> nekojita
generate "baby raccoon (kodanuki)"
  [word +compound "ko" "tanuki"] // -> kodanuki

generate "freezing north wind (kitakaze)"
  [word +compound "kita" "kaze"] // -> kitakaze (*kitagaze)
generate "white tabi (sirotabi)"
  [word +compound "siro" "tabi"] // -> sirotabi (*sirodabi)
generate "time-killer (taikutsusinogi)"
  [word +compound "taikutsu" "sinogi"] // -> taikutsusinogi (*taikutsujinogi)
generate "feminine speech (onnakotoba)"
  [word +compound "onna" "kotoba"] // -> onnakotoba (*onnagotoba)
generate "Gila monster (dokutokage)"
  [word +compound "doku" "tokage"] // -> dokutokage (*dokudokage)

```

A.6 Ilokano Reduplication

```

/*
 * Ilokano Reduplication
 *
 * [McCarthy and Prince 86]
 *
 * For a complete explanation, see [Kenstowicz 1994, p.623]
 */
#include "cv.ph"

//
// Lexicon entries
//
lexicon "basa" [      // 'read'
  morpheme
    sigma#1
      onset#1  b
      nucleus#1 a
    sigma#2
      onset#2  s
      nucleus#2 a
  ]
lexicon "adal" [      // 'study'
  morpheme
    sigma#1
      nucleus#1 a
    sigma#2
      onset#2  d
      nucleus#2 a
      coda#2  l
  ]
lexicon "dait" [      // 'sew'
  morpheme
    sigma#1
      onset#1  d
      nucleus#1 a
    sigma#2
      nucleus#2 i
      coda#2  t
  ]
lexicon "takder" [    // 'standing'
  morpheme
    sigma#1
      onset#1  t
      nucleus#1 a
      coda#1  k
    sigma#2
      onset#2  d
      nucleus#2 e
      coda#2  r
  ]
lexicon "trabaho" [   // 'work'
  morpheme
    sigma#1
      onset#1  t r
      nucleus#1 a
    sigma#2
      onset#2  b
      nucleus#2 a
  ]

```

```

sigma#3
  onset#3 h
  nucleus#3 o
]

//
// Rules
//
block "progressive":
  //
  // Stage 1: Duplicate the entire word.
  //
  rule "reduplication stage 1 (duplicate word)":
    match: [word +progressive morpheme]
    effect: copy morpheme under word before morpheme
  ;

  //
  // Stage 2: Delink all consonantal nodes in copied material.
  //
  rule "reduplication stage 2 (delink consonantal nodes)":
    iterative
    match matched
    no warnings

    match: [word +progressive initial morpheme X within morpheme C consonantal]#1 |
           [word +progressive initial morpheme X within morpheme V consonantal]#2
    effect: delink consonantal#1 from C
           delink consonantal#2 from V
  ;

  //
  // Stage 3: Delete the duplicated morpheme node and all its inferiors.
  //           Since we have delinked all the consonantal nodes, this won't
  //           affect the segmental (feature tree) material.
  //
  rule "reduplication stage 3 (delete morpheme)":
    match: [word +progressive initial morpheme]
    effect: delete morpheme
  ;

  //
  // Stage 4: Insert the reduplication template. Not coincidentally,
  //           the template is the maximal syllable (CCVC) for Ilokano.
  //
  rule "insert template":
    match: [word +progressive initial morpheme]
    effect:
      insert [
        morpheme
          sigma
            onset X#1 C#1
                  X#2 C#2
            nucleus X#3 V#3
                  coda X#4 C#4
          ] before morpheme
  ;

  //
  // Stage 4: Now associate floating consonantal nodes with matching
  //           consonantal nodes in the template. This fills in the

```

```

//      template with as much reduplicated material as possible.
//
//      We first have to find an anchor point. Ilokano requires
//      the first segment of the reduplicated material to be
//      matched with the template, but note that it need not
//      associate with the first C/V slot in the template.
//
//      Once we've found an anchor, we continue matching up
//      segments with free C/V slots in the template. We stop,
//      however, when we get to a segment that cannot be matched.
//      Getting this right is crucial:
//
//      adal -> agadadal (*agadladal)
//
//      This is why we have to do this stage with two rules ---
//      we have to require all but the first reduplicated segment
//      to be next to a segment that's already been matched with
//      the template so we don't skip over segments that can't
//      match (like [a] in [adal]).
//
//      Note that we need both parts of the "|" (or) in the same
//      match specification. If we try to do all the vowels and
//      then all the consonants, we could get crossing association
//      lines.
//
//      Finally, when searching for the anchor we want to look for
//      the leftmost consonant. By default, the matcher gives
//      precedence to higher tiers when deciding which of two matches
//      is the leftmost, so in this case it would favor a match that
//      involved the first C/V slot in the template even if it matched
//      with a noninitial reduplicated segment. Specifying that
//      the matcher should only consider the consonantal tier when making
//      ordering judgments is a simple way to always grab the first
//      consonantal node.
//
rule "anchor template":
  no warnings
  consonantal

  match: ([X childless V]#1 [unlinked -consonantal]#2) |
         ([X childless C]#3 [unlinked +consonantal]#4)
  effect:
    // Vowel case:
    link consonantal#2 to V#1

    // Consonant case:
    link consonantal#4 to C#3
;
rule "fill template":
  iterative
  match matched
  no warnings

  match: ([X consonantal linked within X] [X]* [X childless V]#1 [unlinked -consonantal]#2) |
         ([X consonantal linked within X] [X]* [X childless C]#3 [unlinked +consonantal]#4)
  effect:
    // Vowel case:
    link consonantal#2 to V#1

    // Consonant case:
    link consonantal#4 to C#3

```

```

;

//
// Stage 5: Add the prefix ag-
//
rule "add prefix ag-":
    match: [word +progressive initial sigma]
    effect: insert [sigma nucleus a coda g] before sigma
           change progressive to -
;

//
// Stage 6: We erase any material that's not linked:
//
// - Delete any segments that didn't get linked to
//   C or V nodes.
//
// - Delete any V and C nodes from the template
//   that didn't get filled.
//
// - Delete any X slots that have no children.
//
// - Remove any onsets or codas with no children.
//
rule "stay erasure 1":
    iterative

    match: [unlinked consonantal]
    effect: delete consonantal
;
rule "stray erasure 2":
    iterative
    no warnings

    match: [childless C] | [childless V]
    effect: delete C
           delete V
;
rule "stray erasure 3":
    iterative

    match: [childless X]
    effect: delete X
;
rule "stray erasure 4":
    iterative
    no warnings

    match: [childless onset] | [childless coda]
    effect: delete onset
           delete coda
;

// For absolute correctness, we should merge the two morphemes
// into a single morpheme since the reduplicated string acts
// as a unit with respect to morphology.
;

//
// Samples
//

```

```
generate "agbasbasa" [word +progressive "basa"]
generate "agadadal" [word +progressive "adal"]
generate "agdadait" [word +progressive "dait"]
generate "agtaktakder" [word +progressive "takder"]
generate "agtrabtrabaho" [word +progressive "trabaho"]
```


A.7 Register Tone in Bamileke-Dschang

```

/*
 * Register tone in Bamileke-Dschang
 */
#include "std.ph"

//
// Add tonal root node to vowels.
// This makes them tone-bearing units (TBU's).
//
append phoneme i      [ T ]
append phoneme u      [ T ]
append phoneme o      [ T ]
append phoneme schwa  [ T ]
append phoneme a      [ T ]
append phoneme aw     [ T ]
append phoneme gstop  [ T ]
append phoneme n      [ T ]
append phoneme ny     [ T ]
append phoneme ng     [ T ]

phoneme bare_V [X -consonantal +sonorant]

lexicon "chief" [
  morpheme
    sigma#1
      nucleus#1 e L#1
    sigma#2
      onset#2 f
      nucleus#2 aw L#1
  ]
lexicon "axe" [
  morpheme
    sigma#1
      nucleus#1 n L#1
    sigma#2
      onset#2 d z
      nucleus#2 a L#1
    sigma#3
      nucleus#3 bare_V L#1 down
  ]
lexicon "horn" [
  morpheme
    sigma#1
      nucleus#1 n L#1
    sigma#2
      onset#2 d
      nucleus#2 aw L#1 down
      coda#2 ng
  ]
lexicon "machete" [
  morpheme
    sigma#1
      nucleus#1 ny L#1
    sigma#2
      onset#2 ny
      nucleus#2 i L#1 H#2 unlinked down
  ]
]

```

```

lexicon "song" [
  morpheme
    sigma#1
      nucleus#1 a L#1
    sigma#2
      onset#2 z
      nucleus#2 aw L#1
      coda#2 b
]
lexicon "stool" [
  morpheme
    sigma#1
      nucleus#1 a L#1
    sigma#2
      onset#2 l
      nucleus#2 schwa L#1
      coda#2 ng
]
lexicon "country" [
  morpheme
    sigma#1
      nucleus#1 a L#1
    sigma#2
      onset#2 l
      nucleus#2 a H#2 down
      coda#2 gstop
]
lexicon "tail" [
  morpheme
    sigma#1
      nucleus#1 a L#1
    sigma#2
      onset#2 s
      nucleus#2 a H#2
      coda#2 ng
]

lexicon "leopard" [
  morpheme
    sigma#1
      onset#1 m
      nucleus#1 schwa L#1 down
      coda#1 n
    sigma#2
      onset#2 d z w
      nucleus#2 i L#1
]
lexicon "rooster" [
  morpheme
    sigma#1
      onset#1 m
      nucleus#1 schwa L#1 down
      coda#1 ng
    sigma#2
      onset#2 k
      nucleus#2 u L#1
      coda#2 aw gstop
]
lexicon "dog" [
  morpheme
    sigma#1

```

```

        onset#1  m
        nucleus#1 schwa L#1 down#1
        coda#1   m
    sigma#2
        onset#2  b h
        nucleus#2 u H#2 down#2
]
lexicon "thief" [
    morpheme
        sigma#1
            onset#1  m
            nucleus#1 schwa L#1 down
            coda#1   t
        sigma#2
            onset#2  s
            nucleus#2 aw L#1 H#3
            coda#2   ng
    ]
lexicon "animal" [
    morpheme
        sigma#1
            onset#1  n
            nucleus#1 a unlinked H#1 L#2
    ]
lexicon "squirrel" [
    morpheme
        sigma#1
            onset#1  k
            nucleus#1 a unlinked H#1 L#2
            coda     ng
    ]
lexicon "child" [
    morpheme
        sigma#1
            onset#1  m
            nucleus#1 aw H#1 down
    ]
lexicon "bird" [
    morpheme
        sigma#1
            onset#1  s
            nucleus#1 schwa H#1
            coda     ng
    ]
]

// affixes

lexicon "e" [
    morpheme#1
        unlinked nucleus
        unlinked H
    ]
lexicon "a" [
    morpheme
        sigma#1
            nucleus#1  bare_V L
    ]
]

//
// Rules
//

```

```

rule "TBU cleanup":
  iterative
  match matched
  no warnings

  match: [ onset T ]#1 | [ coda T ]#2
  effect: delete T#1
         delete T#2
;

block "continuous": // rules that apply continuously
  rule "contour reduction":
    iterative

    match: [ T L#1 H#2 down#1 ]
    effect: delink L from T
  ;

  rule "down deletion":
    iterative

    match: [ H ]#1 [ T L down ]#2
    effect: delete down
  ;

  rule "double downstep reduction":
    iterative
    match matched

    match: [ T down#1 down#2 ]
    effect: delete down#1
  ;
  rule "floating L deletion":
    iterative

    match: [unlinked L]
    effect: stray erase L
  ;
;

rule "downstep hoppng":
  iterative
  match matched

  match: [ morpheme T#1 within morpheme L#1 down#1 T#2 within morpheme !register ]#1
  effect: link down#1 to T#2
         delink down#1 from T#1
;

call "continuous"

rule "H spreading right":
  iterative
  match matched

  match: [ T H ]#1 [ T L !register ]#2
  effect: link H#1 to T#2
         delink L#2 from T#2
         stray erase L#2
;

```

```

call "continuous"

rule "H spreading left 1":
  match: [ T L ]#1
         [ T L ]#2
         [ word final T within word H down ] #3

  effect: link H#3 to T#2
         delink L#2 from T#2
         link down#3 to T#2
         delink down#3 from T#3
;

rule "H spreading left 2":
  match: [ T H ]#1
         [ T L ]#2
         [ word final T within word H down ] #3

  effect: link H#3 to T#2
         delink L#2 from T#2
;

call "continuous"

rule "down docking":
  rightmost

  match: [ T unlinked down within T ]#1 [ T ]#2
  effect: link down to T#2
;

call "continuous"

rule "stray erasure":
  iterative

  match: [unlinked H]
  effect: stray erase H
;

//
// Examples
//

// table 1

generate "chief of leopards" [word "chief" "e" "leopard"]
generate "chief of roosters" [word "chief" "e" "rooster"]
generate "chief of dogs" [word "chief" "e" "dog"]
generate "chief of thieves" [word "chief" "e" "thief"]
generate "axe of leopards" [word "axe" "e" "leopard"]
generate "axe of roosters" [word "axe" "e" "rooster"]
generate "axe of dogs" [word "axe" "e" "dog"]
generate "axe of thieves" [word "axe" "e" "thief"]
generate "horn of leopards" [word "horn" "e" "leopard"]
generate "horn of roosters" [word "horn" "e" "rooster"]
generate "horn of dogs" [word "horn" "e" "dog"]
generate "horn of thieves" [word "horn" "e" "thief"]
generate "machete of leopards" [word "machete" "e" "leopard"]
generate "machete of roosters" [word "machete" "e" "rooster"]
generate "machete of dogs" [word "machete" "e" "dog"]

```

```

generate "machete of thieves" [word "machete" "e" "thief"]
generate "song of leopards" [word "song" "a" "leopard"]
generate "song of roosters" [word "song" "a" "rooster"]
generate "song of dogs" [word "song" "a" "dog"]
generate "song of thieves" [word "song" "a" "thief"]
generate "stool of leopards" [word "stool" "a" "leopard"]
generate "stool of roosters" [word "stool" "a" "rooster"]
generate "stool of dogs" [word "stool" "a" "dog"]
generate "stool of thieves" [word "stool" "a" "thief"]
generate "country of leopards" [word "country" "a" "leopard"]
generate "country of roosters" [word "country" "a" "rooster"]
generate "country of dogs" [word "country" "a" "dog"]
generate "country of thieves" [word "country" "a" "thief"]
generate "tail of leopards" [word "tail" "a" "leopard"]
generate "tail of roosters" [word "tail" "a" "rooster"]
generate "tail of dogs" [word "tail" "a" "dog"]
generate "tail of thieves" [word "tail" "a" "thief"]

// table 2
generate "chief of animal" [word "chief" "e" "animal"]
generate "chief of squirrel" [word "chief" "e" "squirrel"]
generate "chief of child" [word "chief" "e" "child"]
generate "chief of bird" [word "chief" "e" "bird"]
generate "axe of animal" [word "axe" "e" "animal"]
generate "axe of squirrel" [word "axe" "e" "squirrel"]
generate "axe of child" [word "axe" "e" "child"]
generate "axe of bird" [word "axe" "e" "bird"]
generate "horn of animal" [word "horn" "e" "animal"]
generate "horn of squirrel" [word "horn" "e" "squirrel"]
generate "horn of child" [word "horn" "e" "child"]
generate "horn of bird" [word "horn" "e" "bird"]
generate "machete of animal" [word "machete" "e" "animal"]
generate "machete of squirrel" [word "machete" "e" "squirrel"]
generate "machete of child" [word "machete" "e" "child"]
generate "machete of bird" [word "machete" "e" "bird"]
generate "song of animal" [word "song" "a" "animal"]
generate "song of squirrel" [word "song" "a" "squirrel"]
generate "song of child" [word "song" "a" "child"]
generate "song of bird" [word "song" "a" "bird"]
generate "stool of animal" [word "stool" "a" "animal"]
generate "stool of squirrel" [word "stool" "a" "squirrel"]
generate "stool of child" [word "stool" "a" "child"]
generate "stool of bird" [word "stool" "a" "bird"]
generate "country of animal" [word "country" "a" "animal"]
generate "country of squirrel" [word "country" "a" "squirrel"]
generate "country of child" [word "country" "a" "child"]
generate "country of bird" [word "country" "a" "bird"]
generate "tail of animal" [word "tail" "a" "animal"]
generate "tail of squirrel" [word "tail" "a" "squirrel"]
generate "tail of child" [word "tail" "a" "child"]
generate "tail of bird" [word "tail" "a" "bird"]

```

Bibliography

- Albro, D. M. 1994. AMAR: A computational model of autosegmental phonology. Bachelor's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, February.
- 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.
- Bird, S. 1993. The morphology of the Dschang-Bamileke associative construction: a pilot experiment. In T. M. Eillison and J. M. Scobbie (Eds.), *Computational Phonology*, 139–56. Edinburgh, Scotland: University of Edinburgh.
- Bird, S., and T. M. Ellison. 1992. One level phonology: Autosegmental representations and rules as finite automata. Research Paper EUCCS/RP-51, University of Edinburgh, Edinburgh, Scotland, April.
- Chomsky, N. A. 1986. *Knowledge of language: Its nature, origin, and use*. New York: Prager.
- Chomsky, N. A., and M. Halle. 1968. *The Sound Pattern of English*. New York, NY: Harper and Row.
- Dell, F., and M. Elmedlaoui. 1985. Syllabic consonants and syllabification in Imdlawn Tashlhiyt Berber. *Journal of African languages and linguistics* 7:105–30.
- Dell, F., and O. Tangi. 1991. Syllabification and empty nuclei in Ath-Sidhar Rifian Berber. Research paper, Centre National de la Recherche Scientifique, Paris.
- Durand, J. 1990. *Generative and non-linear phonology*. New York, New York: Longman.
- Giegerich, H. J. 1992. *English Phonology*. New York, NY: Cambridge University Press.
- Goldsmith, J. A. 1976. *Autosegmental Phonology*. PhD thesis, Massachusetts Institute of Technology.
- Goldsmith, J. A. 1990. *Autosegmental and Metrical Phonology*. Cambridge, MA: Blackwell Publishers.
- Halle, M. 1992. Phonological features. In W. Bright (Ed.), *International Encyclopedia of Linguistics*, vol. 3, 207–12. Oxford: Oxford University Press.
- Hamid, A. H. 1984. *The Phonology of Sudanese Arabic*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois.
- Harris, J. 1990. *Syllable structure and stress in Spanish: a nonlinear analysis*. Cambridge, MA: MIT Press.

- 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.
- Hewitt, M., and A. Prince. 1989. OCP, locality, and linking: the N. Karanga verb. In *Proceedings of the west coast conference on formal linguistics 8*, 176–91.
- Hopcroft, J. E., and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley.
- Hyman, L. M. 1985. Word domains and downstep in Bamileke-Dschang. *Phonology Yearbook* 2:45–83.
- Itô, J., and A. Mester. 1986. The phonology of voicing in Japanese. *Linguistic Inquiry* 17:49–73.
- Johnson, S. 1986. YACC: Yet another compiler compiler. In *UNIX Programmer's Manual: Supplementary Documents 1*. Berkeley, California: University of California, Berkeley CSRG.
- Kenstowicz, M. 1994. *Phonology in Generative Grammar*. Cambridge, MA: Blackwell Publishers.
- Kernighan, B. W., and D. M. Ritchie. 1988. *The C Programming Language*. AT&T Bell Laboratories, second edition.
- Koskenniemi, K. 1983. Two-level morphology: A general computational model for word-form recognition and production. Phd thesis, University of Helsinki, Helsinki, Finland.
- Lesk, M. 1986. Lex — a lexical analyzer generator. In *UNIX Programmer's Manual: Supplementary Documents 1*. Berkeley, California: University of California, Berkeley CSRG.
- Lombardi, L. 1991. *Laryngeal features and laryngeal neutralization*. PhD thesis, University of Massachusetts at Amherst, Amherst, MA.
- Marantz, A. 1982. Re reduplication. *Linguistic Inquiry* 13:435–82.
- Maxwell, M. 1994. Parsing using linearly ordered rules. In *Proceedings of the ACL Special Interest Group on Phonology*, 59–70.
- McCarthy, J. J., and A. S. Prince. 1986. Prosodic morphology. Unpublished manuscript, October.
- McCarthy, J. J. 1975. Formal problems in Semitic phonology and morphology. Phd thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June.
- Newman, P. 1986. Tone and affixation in Hausa. *Studies in African linguistics* 17:249–67.
- Prince, A., and P. Smolensky. 1993. Optimality theory. Unpublished manuscript.
- Pulleyblank, D. 1986. *Tone in Lexical Phonology*. Dordrecht: Reidel.
- Pullyblank, D. 1982. Downstep in Dschang. Occasional paper, Massachusetts Institute of Technology, Cambridge, Massachusetts.
- Rialland, A., and M. Badjime. 1989. Réanalyse des tons du Bambara: Des tons du nom à l'organisation générale du système. *Studies in African linguistics* 20:1–28.
- Sagey, E. 1986. *The Representation of Features and Relations in Nonlinear Phonology*. PhD thesis, Massachusetts Institute of Technology.

Schaub, W. 1985. *Babungo*. Dover, New Hampshire: Croom Helm.

Snider, K. L. 1990. Tonal upstep in Krachi: Evidence for a register tier. *Language* 66(3):453–73.

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.