The Delta Programming Language: An Integrated Approach to Non-Linear Phonology, Phonetics, and Speech Synthesis

Susan R. Hertz

1 Brief Overview

The Delta programming language is designed to let linguists easily formalize and test phonological and phonetic theories. Its central data structure lets rule-writers represent utterances as multiple "streams" of synchronized units of their choice, giving them considerable flexibility in expressing the relationship between phonological and phonetic units. This paper presents the Delta language, showing how it can be applied to two linguistic models, one for Bambara tone and fundamental frequency patterns and one for English formant patterns. While Delta is a powerful, special-purpose language that alone should serve the needs of most phonologists, phoneticians, and linguistics students who wish to test their rules, the Delta System also provides the flexibility of a general-purpose language by letting users intermingle C programming language statements with Delta statements.

2 Introduction

Despite their common interest in studying the sounds of human language, the fields of phonology and phonetics have developed largely independently in recent years. One of the contributing factors to this unfortunate division has been the lack of linguistic rule development systems. Such systems are needed to let linguists easily express utterance representations and rules, and facilitate the computational implementation and testing of phonological and phonetic models.

SRS (Hertz, 1982) is a rule development system that was designed, starting in 1974, for just this purpose—to let linguists easily test phonological and phonetic rules, and explore the interface between phonology and phonetics through speech synthesis. SRS, however, was influenced quite heavily by the theory of generative phonology that was prevalent at the time, a theory that posited linear utterance representations consisting

of a sequence of phoneme-sized segments represented as bundles of features (Chomsky and Halle, 1968). Although at the phonetic level, SRS uses different "streams" for different synthesizer parameters, the parameter values and segment durations must all be set in relation to the phoneme-sized segments at the linear phonological level.

Thus, while SRS lets users express rules in a well-known linguistic rule notation, and easily change the rules, it forces them to work within a particular framework. Because SRS was biased toward a particular theory of sound systems, we became equally biased in our approach to data analysis and rule formulation. For example, we took for granted that phonemes (more precisely, phoneme-sized units) were the appropriate units for the assignment of durations and formant patterns, a basic assumption that blinded us for years to the possibility of alternative models. As alternatives finally emerged, however, the need for a more flexible system for expressing and testing phonological and phonetic rules became apparent.

The clearest requirements for a more flexible rule development tool were a multi-level (or multi-tiered) data structure that could make explicit the relationship between phonological and phonetic units, and a precise and flexible rule formalism for manipulating this structure. In response to these needs, in July 1983 I began the development of a new synthesis system, the Delta System (Hertz, Kadin, and Karplus, 1985; Hertz, 1986), in consultation with two computer scientists, Jim Kadin and Kevin Karplus. The Delta System provides a high-level programming language specifically designed to manipulate multi-level utterance representations of the sorts suggested by our rule-writing experience with SRS (Hertz, 1980; Hertz, 1981; Hertz, 1982; Hertz and Beckman, 1983; Beckman, Hertz, and Fujimura, 1983). This language lets users write and test rules that operate on multi-level utterance representations without having to take care of the programming details that would be required in an ordinary programming language like C. A one-line delta statement might easily take a page to accomplish in C. The ease of expressing and reading rules in Delta enables rule-writers to test alternative strategies freely and conveniently.

While the move from the linear utterance representations central to SRS to the multi-level representations central to Delta parallels the move by phonologists from linear to non-linear representations, the Delta System is a direct consequence of our SRS experience, and, unlike SRS, was developed independently of the phonological theories in vogue at the time. The Delta System is flexible enough to let phonologists and

phoneticians of different persuasions express and test their ideas, constraining their representations and rules in the ways they, rather than the system, see fit. The system assumes as little as possible about the phonological and phonetic relationships that rule-writers may wish to represent in their utterance representations, and the manner in which their rules should apply, allowing them to make dependencies between rules explicit and giving them full control, for example, over whether their rules should apply cyclically or non-cyclically, sequentially or simultaneously, left-to-right or right-to-left, morph by morph or syllable by syllable, to the entire utterance or only a portion thereof, and so on.

In addition to a powerful programming language for building and manipulating multi-level utterance representations, the Delta System provides a flexible interactive debugger. The debugger lets users issue commands to interact with a program while it is executing. It lets users trace their rules during program execution, stop the execution of their program at selected points (e.g., each time the utterance representation or a particular variable changes), display the utterance representation and other data structures, modify the utterance representation "on the fly" (e.g., to hear the result of a longer duration for a particular unit in a program designed for synthesis), and so on. The debugger, like the system in general, is designed for speed and flexibility in the development of phonological and phonetic rules, letting rule-writers test and modify their hypotheses quickly and easily. The debugger is an essential part of the system, but a description of it is outside the scope of this paper. Hertz et al. (1985) describes an early version of the debugger. The debugger has been enhanced substantially since that paper was written. It is now a complete source-level debugger with many more capabilities than those shown in that paper.

The Delta System has been designed to be as portable as possible, to give linguists the widest possible access to it. A Delta program is compiled by the Delta compiler into a C program, and can be run on any computer with a standard C compiler and at least 512K of memory, such as an IBM PC-AT or a Macintosh. Compiling into C has the additional advantage that it lets users integrate C programs with Delta programs at will, even intermingling Delta and C code in a single procedure.[1] The system is also made

---

1. Earlier versions of the system compiled Delta programs into pseudo-machine instructions. The current approach (compiling into C) sacrifices the extremely compact storage of rules achieved by the pseudo-machine approach in favor of much faster rule execution and flexibility in interfacing C routines.

accessible through the comprehensive Delta User's Manual, which contains an overview of the system, a tutorial, extensive reference sections, and sample programs.

The next section of this paper (Section 3) presents selected features of the Delta language, introducing many of the concepts needed to understand the sample programs in subsequent sections. It uses examples from Bambara, a Mande tone language spoken in Mali. These examples anticipate the programs in Section 4, which illustrate how tone patterns and corresponding fundamental frequency values might be assigned in Bambara. Bambara is chosen because it exhibits many of the properties of tone languages that have motivated multi-level representations in phonology (e.g., tone spreading and floating tones), while at the same time providing good examples of the function of phonological units in determining actual phonetic values (e.g., the role of the tones in determining fundamental frequency values). Section 5 presents a model of English formant timing that further illustrates Delta's flexibility in accomodating a wide range of theories about the interface between phonology and phonetics. Finally, Section 6 presents conclusions, gives a brief overview of the features of Delta not described in the paper, and discusses our plans for enhancing Delta and complementing it with another system in the future.

3 Selected Features of Delta

The Delta programming language is a high-level language designed to create, test, and manipulate a data structure called a *delta* for representing utterances. A delta consists of a set of user-defined *streams* of *tokens* that are synchronized with each other at strategic points. The tokens can represent anything the rule-writer wishes—phrases, morphs, syllables, tones, phonemes, sub-phonemic units, acoustic parameters, articulators, durations, classes of features, and so on. This section describes the structure of deltas, focusing first on the kinds of relationships that can exist between tokens in different streams, and then on the language for determining and testing these relationships.

For most of its sample deltas, this section uses the Bambara phrase *muso jaabi* 'answering the woman', which can be transcribed as [mùsó ! já:bí]. In this transcription, the grave accent ` represents a low tone and the acute accent ´ a high tone. Thus the first syllable has low tone, and all the other syllables have high tone. The exclamation point represents tonal downstep, the lowering in pitch of the following high tones. This

lowering occurs in Bambara after a definite noun. To account for the tonal downstep, linguists, following Bird (1966), posit as a definite marker a "floating low tone" that occurs after the noun and is not associated with any syllable.

Following is a sample delta that a program couched in the framework of autosegmental CV phonology (Clements and Keyser, 1983) might build for the phrase *muso jaabi*:

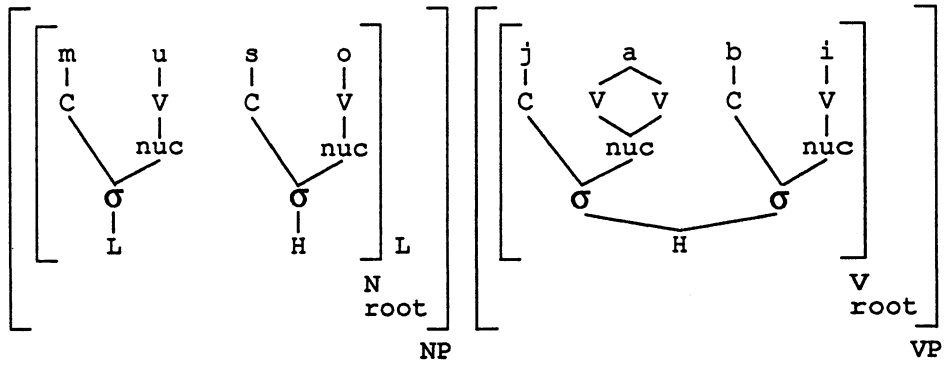
```
(1)  phrase:   |          NP              |          VP          |
     word:     |         noun         |   |         verb         |
     morph:    |        root          |   |         root         |
     phoneme:  | m |  u  | s |  o   |   | j |   a   | b |  i   |
     CV:       | C |  V  | C |  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
```

This delta consists of eight streams: `phrase`, `word`, `morph`, `phoneme`, `CV`, `nucleus`, `syllable`, and `tone`. The `phrase` stream has two tokens, `NP` (noun phrase) and `VP` (verb phrase); the `word` stream has two tokens, `noun` and `verb`; and so on. The tokens in the `CV` stream represent abstract timing units, in accordance with CV theory. The long phoneme `a` is synchronized with two `V` tokens in the `CV` stream, while the short vowels are synchronized with a single `V`. The `nucleus` stream marks each vowel, regardless of length, as the nucleus of the syllable. The `tone` stream has four tokens, two `L` (low) tokens and two `H` (high) tokens, reflecting the tone pattern given in the transcription above.

The vertical bars in each stream are called *sync marks*. Our sample delta has eleven sync marks, numbered at the bottom for ease of reference. Sync marks are used to synchronize tokens across streams. For example, sync marks 1 and 3 synchronize all of the tokens that constitute the first syllable. Sync marks 5 and 6 surround a `L` tone that is not synchronized with any tokens in any other stream. It represents a floating low tone that marks the noun phrase as definite, as discussed above. Sync marks 1 and 6 synchronize this floating tone along with the preceding `L` and `H` tones of the root with the `NP` token in the `phrase` stream. Following the analysis of Rialland and Sangare (1985), sync marks 6 and 11 synchronize a single `H` tone with two syllables and one root, rather than a separate `H` tone with each syllable.

This delta could be expressed in more familiar autosegmental terms as follows:

(2)



Note that while the delta representation makes the appropriate associations among tokens using sync marks alone, the autosegmental representation must use brackets in addition to association lines in order to show that the floating low tone is part of the noun phrase.

Furthermore, while the tiers in autosegmental representations are critically ordered with respect to each other in the sense that "tokens" on one tier can only be explicitly linked to tokens on particular other tiers (see, for example, Clements, 1985) the streams of a delta can always occur in any order with respect to each other. The same relationships exist between the tokens in different streams regardless of the order in which the streams are listed. For example, the `phoneme`, `syllable`, and `CV` streams in delta (1) above could also be displayed in the following order:

```
(3)  phoneme:  | m | u | s | o |   | j |   a   | b | i |
     syllable: |  syl  |  syl  |   |    syl    |  syl  |
     CV:       | C | V | C | V |   | C | V | V | C | V |
```

As in this example, examples of deltas from here on will show only those streams relevant to the discussion at hand.

In addition to syntactic, morphological, and phonological streams, a delta can also have phonetic streams. For example, a time stream might be added to delta (1) as follows:

```
(4)  phoneme:       |  s  |  o  |   |  j  |   a   |
     CV:       ... |  C  |  V  |   |  C  | V | V | ...
     duration:     | 200 | 150 |   | 140 |  200  |
```

Here, a duration (in milliseconds) is synchronized with each phoneme token.

In a delta, unlike in an autosegmental representation, a time stream can be used to time articulatory movements or acoustic patterns with respect to phonological units like

phonemes. In the following delta fragment, an `F0` (fundamental frequency) stream is used to time F_0 targets with respect to the syllable nuclei:

```
(5)  phoneme:       | s |        o        |    | j |         a         |
     nucleus:       |   |       nuc       |    |   |        nuc        |
     tone:      ... |          H          | L |            H          |  ...
     F0:            |         | 150 |     |    |          | 130 |      |
     duration:      |200| 75  |  0  | 75  |    |140| 100  |  0  | 100 |
```

An F_0 target is placed halfway through each nucleus. The targets themselves have no duration, being used only to shape the F_0 pattern, which moves from target to target in accordance with the specified durations. A program designed for actual synthesis could interpolate the values (transitions) between the targets and send them, along with the values for other synthesizer parameters, to the synthesizer.

3.1 Token Structure

Each token in a stream is a collection of user-defined *fields* and *values*. Each token has at least a *name field*. It is the value of the name field that is displayed for each token in the deltas shown above. Tokens can be given other fields as well, as shown below for a phoneme token named `m`:

```
(6)  name:      m
     place:     labial
     manner:    sonorant
     class:     cons
     nasality:  nasal
```

where `cons` = "consonantal". All tokens in a given stream have the same fields, but of course the values for the fields can differ.[2] Also, the value for a field can be undefined, when the value is not relevant for the token. Fields can be of different types, as discussed below. In most of the sample deltas, only the value of the name field is displayed, but it should be kept in mind that the tokens may have other fields as well.

A field together with a particular value is called an *attribute*. Thus the phoneme token illustrated above has the attributes `<name: m>`, `<place: labial>`, `<manner: sonorant>`, and `<nasality: nasal>`. The non-name attributes of a

---

2. A future version of Delta is planned that will allow tokens in the same stream to have different fields. This next version will also allow tokens to be represented as trees of features, as in the model of autosegmental phonology proposed by Clements (1985). See Section 6, "Final Remarks", for more details.

token are also called *features*, so that we can speak of the token named m as having the features <place: labial>, <manner: sonorant>, etc. In general, token features are distinguished from token names in this paper by being enclosed in angle brackets. When the value of a field is unambiguous (i.e., when it is a possible value for only one field in the stream in question), the field name can be omitted in Delta programs, so we can also consider the m to have the features <labial>, <sonorant>, and so on. This abbreviated form for features will be used throughout the paper.

3.2 Delta Definitions

The first thing that a Delta rule-writer must do is give a *delta definition.* A delta definition consists of a set of *stream definitions* that define the streams to be built and manipulated by the program (rules). Figure 1 shows fragments of possible phoneme and F0 stream definitions for Bambara. All text following a double colon (::) to the end of a line is a comment that is not part of the actual stream definition.

****

Insert Figure 1 here

****

The phoneme stream definition defines the tokens in the phoneme stream as having name, place, manner, class, nasality, voicing, height, and backness fields. (The stream names in the program fragments in this paper are all preceded by a percent sign.) The name field is a *name-valued* field; the place, manner, height, and backness fields are *multi-valued* fields; and the class, nasality and voicing fields are *binary* fields. A name-valued field is a field that contains the token names of some stream as possible values.[3] A multi-valued field is a field that has more than two posible values and is not name-valued and not *numeric* (see the next paragraph). A binary field is a non-name-valued field that has exactly two possible values, such as <nasal> or <~nasal>, where "~" is Delta notation for "not".

---

3. It is not only the name field that can be name-valued. For example, Hertz et al. (1985) gives an example of a text stream definition that defines a name-valued field called default_pronunc with the names of the tokens in the phoneme stream as possible values, and shows how this field might be used in a Delta program for English text-to-phoneme conversion to synchronize default pronunciations (phoneme tokens) with text characters.

A binary field is always defined by specifying only one of the two possible values. The opposite value is assumed.

The phoneme tokens in this example do not have any numeric fields. A numeric field would be defined by following the field name with a keyword specifying the kind of number that can be a value of the field, as shown in the `F0` stream definition, where the name field is defined as having integers as possible values. Thus a field can be both name-valued and numeric; all other field types are mutually exclusive. The keywords for numeric values are the same as the numeric type specifiers in C.

Below the field definitions for the `phoneme` stream are a set of initial feature definitions. These definitions assign values to tokens with particular names. When a token with a particular name is inserted into a delta stream, the token's initial field values are automatically set, as discussed below.

If a token is not given a value for some field by an initial feature definition, it is automatically given an initial default value. For binary fields, the initial default value is the binary value not specified for the field in the stream definition. For example, in this case, all tokens not given the value `<cons>` for the `class` field will automatically be given the value `<~cons>`, and all tokens not given the value `<~voiced>` for the `voicing` field will automatically be given the value `<voiced>`. For multi-valued fields, the default value is `<undefined>`, a built-in value automatically defined for any multi-valued field. For numeric fields (other than numeric name fields), the default value is `<0>`. For name-valued fields, the default value is `GAP`, a built-in value automatically defined for any name-valued field. Gaps (i.e., tokens named `GAP`) are generally used as special "filler tokens" that separate tokens that would otherwise be considered adjacent, as discussed below.

3.3 Sample Program—Synchronizing Tokens

A Delta program consists of a delta definition followed by a set of procedures that operate on the delta. Figure 2 shows a short sample program that reads a sequence of phoneme tokens representing a Bambara word from the terminal into the `phoneme` stream, and synchronizes a `C` token in the `CV` stream with an initial consonantal token in the `phoneme` stream. (Later it will be shown how to apply the rule across the entire delta, synchronizing a `C` token with each consonantal phoneme.)

****

Insert Figure 2 here

****

This program consists of a single procedure called `main`. Every program must have at least a procedure called `main`, where execution of the program begins. When the program begins execution, the delta has the following form (assuming that the streams shown are those defined by the delta definition):[4]

```
(7)  phrase:    |  |
     word:      |  |
     morph:     |  |
     phoneme:   |  |
     CV:        |  |
     syllable:  |  |
     tone:      |  |
```

The first program line,

```
(8)  read %phoneme;
```

reads a sequence of phoneme token names from the terminal and places the tokens in the `phoneme` stream. The fields of each phoneme are set as specified in the `phoneme` stream definition. For example, given the `phoneme` stream definition in Figure 1, if the sequence `m u s o` is entered, the delta would have the following form after the read statement has been executed:

```
(9)  phrase:       |             |          |
     word:         |             |          |
     morph:        |             |          |
     phoneme:      |             |          |
       name:       | m           | u        |
       place:      | labial      | ----     |
       manner:     | sonorant    | ----     | ...
       class:      | cons        | ~cons    |
       nasality:   | nasal       | ~nasal   |
       voicing:    | voiced      | voiced   |
       height:     | ----        | high     |
       backness:   | ----        | back     |
     syllable:     |             |          |
     tone:         |             |          |
```

---

4. Earlier papers about Delta showed initial deltas with a gap in each stream. The current version of Delta lets rule-writers specify in the stream definition whether a stream should be initialized to contain a gap or nothing. "Nothing" is assumed by default.

The dashes for the height and backness fields in the m and the place and manner fields in the u represent the value <undefined>.

The next statement,

```
(10) [%phoneme _^left <cons> !^ac] -> insert [%CV C] ^left...^ac;
```

is a *rule*. A rule consists of a *test* and an *action*. The action in this rule is separated from the test by a right arrow (->).

The test portion of the rule,

```
(11) [%phoneme _^left <cons> !^ac]
```

is a *delta test*, which tests the delta for a particular sequence of tokens and sync marks. Sync marks are referred to in Delta programs by means of *pointer variables* (also called pointers), such as ^left and ^ac in our sample rule. Pointer variable names in this paper always begin with a carat (^). The variable ^left is a built-in pointer that always points at the leftmost sync mark in the delta, while ^ac ("after consonant") is a user-defined pointer whose use is explained below.

The sync mark where testing starts is the *anchor* of the delta test, and is marked by an underscore before the appropriate pointer name. In the above test, ^left is the anchor, so testing starts at the leftmost sync mark in the delta.

The test looks in the phoneme stream immediately to the right of ^left for a token that has the feature <cons>. If such a token is found, the expression !^ac sets ^ac to point at the sync mark immediately to its right. Since our sample phoneme stream does start with a consonantal token, the delta test succeeds and ^ac is set to the sync mark following the token:

```
(12) phoneme: | m   | u | s | o |
     CV:      |                 |
              ^left ^ac
```

The action of the rule,

```
(13) insert [%CV C] ^left...^ac;
```

inserts a token named C into the CV stream between ^left and ^ac:

```
(14) phoneme: |  m  | u | s | o |
     CV:      |  C  |           |
              ^left ^ac
```

The expression `^left...^ac`, which specifies where the insertion is to take place, is called the *insertion range.*

Note that the sync mark pointed to by `^ac`, originally defined only in the `phoneme` stream, is now defined in the `CV` stream as well. In general, when an insertion is made between two sync marks in the delta, each sync mark is put into the insertion stream if it does not already exist in that stream. A sync mark that is put (defined) in a new stream is said to be *projected* into that stream.

The final statement in our sample program,

```
(15) print delta;
```

displays the delta, showing only the token names in each stream. (Other print statements can be used to display features.) The sample program ends after this statement.

3.4 Adjacent Sync Marks

Adjacent sync marks in a stream act like a single sync mark for purposes of testing the stream. Consider, for example, the `morph` and `tone` streams of delta (1):

```
(16) morph: | root  |    | root |
     tone:  | L | H | L |  H   |
```

The following delta test would succeed, despite the intervention of two sync marks between the roots in the `morph` stream:

```
(17) [%morph _^left root root]
```

Thus the floating `L` tone is in effect "invisible" in the `morph` stream. It could be made visible by placing a gap between the adjacent sync marks. If a gap were present, the above test would fail.

3.5 One-Point vs. Two-Point Insertions

The insert statement presented in Figure 2 specified a *two-point insertion*, which places a token sequence between two sync marks already existing in the delta. An insert statement can also specify a *one-point insertion*, which places a token sequence to the right or left of a single sync mark. For example, given the delta

```
(18) phoneme:   | m | u | s | o |   |
     syllable:  |  syl  |  syl  |   | ...
     tone:      |       L       | L |
                1   2   3   4   5   6
```

the one-point insert statement

```
(19) insert [%tone H] ...^5;
```

would insert a `H` token into the `tone` stream just before sync mark 5, automatically creating a new sync mark before the inserted `H` token:[5]

```
(20) phoneme:   | m | u | s | o |   |
     syllable:  |  syl  |  syl  |   | ...
     tone:      |     L     |  H  | L |
                1   2   3 ↑ 4   5   6
                    (new sync mark)
```

Note that the new sync mark is unordered with respect to sync marks 2, 3, and 4.

3.6 Sync Mark Ordering

Within each stream, all the sync marks have an obvious left to right ordering. Across streams, however, two sync marks may or may not have a relative left to right ordering. Consider, for example, the following delta:

```
(21) morph:     |      root     |   |        root       |
     phoneme:   | m | u | s | o |   | j | a | a | b | i |
     syllable:  |  syl  |  syl  |   |    syl    |  syl  |
     tone:      |   L     |  H  | L |         H         |
                1   2   3 4 5   6   7   8   9   10  11  12
```

---

5. In the examples of one-point insertions in earlier papers about Delta, the new sync mark bounding the inserted token was automatically projected into all streams in which the sync mark designated by the range pointer was defined. The system has been changed so that by default the new sync mark is only defined in the insertion stream. The `project` option to the insert statement can be used to cause one-point insert statements to work the way they did in earlier versions of Delta, as illustrated in example (55).

In this delta, sync mark 4 is to the left of sync marks 6 through 12, because sync mark 6 is in the `tone` stream after sync mark 4, and is also in the `phoneme` stream before sync marks 7 through 12. Similarly, sync mark 4 is to the left of sync mark 8, because sync marks 6 and 7 are to the right of sync mark 4 in the `tone` stream, and to the left of sync mark 8 in the `phoneme` stream. However, sync mark 4 is not ordered with respect to sync mark 5, because sync mark 5 does not exist in the `tone` stream, and there is no sync mark between sync mark 4 and 5 that is to the right of 4 or the left of 5 or vice versa. (Sync mark 4 could just as well have been displayed after sync mark 5.) By the same logic, sync mark 4 is also not ordered with respect to sync mark 2 or 3.

Delta (21) might be posited as an early form for the word *muso* during the derivation of its surface representation, as explained below. It gives the tone pattern `L H` to the first root, without synchronizing either tone with a particular syllable.

Delta's *merge statement* can be used to merge two unordered sync marks into a single sync mark, creating the appropriate synchronizations. For example, if `^3` points at sync mark 3 in delta (21) and `^4` at sync mark 4, the statement

```
(22) merge ^3 ^4;
```

would produce the following delta:

```
(23) morph:     |       root       |    |
     phoneme:   | m | u | s | o |    | ...
     syllable:  |  syl  |  syl  |    |
     tone:      |   L   |   H   | L  |
                1   2   3   5   6    7
                      (4)
```

It will be assumed in the remainder of this paper that in the examples that contain deltas with numbered sync marks, `^1` has been set to point at sync mark 1, `^2` at sync mark 2, etc.

3.7 Contexts

The ordering of sync marks is important for determining sync mark *contexts*, which are in turn important for determining where sync marks can be legally projected and for determining the relationship between tokens across streams, as explained below. The context of a sync mark in stream x is the portion of stream x where the sync mark

could be put without causing it to cross the closest sync mark to its left or right. For example, in the delta

```
(24) morph:     |       root      |    |
     phoneme:   | m | u | s | o |    | ...
     syllable:.|  syl  |  syl  |    |
     tone:      |   L     |  H  | L  |
                1   2   3 4 5   6    7
```

the context in the `morph` stream of sync mark 2 is the `root` token; the context in the `phoneme` stream of sync mark 4 is the sequence of phonemes and sync marks between sync marks 1 and 6 (recall that sync mark 4 is unordered with respect to sync marks 2, 3, and 5); the context in the `syllable` stream of sync mark 5 is the second `syl` token, and so on;

The *left context* in stream *x* of a sync mark *n* is the sync mark that is the left boundary of the context in stream *x* of sync mark *n*. Thus in delta (24), the left context of sync mark 2 in the `morph` stream is sync mark 1, and the left context of sync mark 5 in the `syllable` stream is sync mark 3. Similarly, the *right context* in stream *x* of a sync mark *n* is the sync mark in stream *x* that is the right boundary of the context in stream *x* of sync mark *n*. If sync mark *n* is defined in stream *x*, its context, left context, and right context in stream *x* is itself.

The Delta language has the operators `\\` and `//` for taking the left and right context of a sync mark. For example, the statement

```
(25) ^x = \\ %syllable ^5;
```

sets `^x` to point at the sync mark that is the left context of `^5` in the `syllable` stream—at sync mark 3 in the case of delta (24).

In the following test, the context operators are used to test whether a `H` tone is the only token between the left context in the `tone` stream of sync mark 5 and the right context in the `tone` stream of sync mark 6. This test can be applied to delta (24) to determine whether the vowel `o` (surrounded by sync marks 5 and 6) is "contained in" a `H` tone.

```
(26) [%tone _(\\ %tone ^5) H (// %tone ^6)]
```

(The multiple references to the `tone` stream could be eliminated by specifying in a previous statement a stream to be assumed if no other is specified. In general, Delta lets users set default streams for different purposes.) This test would fail in the case of delta (24), since the `o` is "contained in" a sequence of two tones, `L H`, but would succeed in delta (23), where the merging of sync marks 3 and 4 has provided an ordering of sync mark 5 with respect to the sync mark between the two tones.

3.8 Time Streams

While sync marks can be used to specify gross temporal relationships, such as whether one token is before, after, partway through, or concurrent with another, *time streams* are needed to make precise temporal specifications—for example, that a token in one stream begins 75% of the way through the duration of a token in another stream, or 50 milliseconds after its end. The tokens in a time stream are different from other (non-gap) tokens in that sync marks can be projected into the middle of them. When a sync mark is projected into a time token, the token is automatically divided into the appropriate pieces. For example, if a sync mark is placed a quarter of the way through a token with a duration (name) of `100`, the token would get divided into a token with duration `25` and a second token with duration `75`.

A time stream is defined very much like any other stream, as illustrated by the following definition of a time stream called `duration`:

```
(27) time stream %duration;
        name: int;
     end %duration;
```

The keyword `time` in the first line indicates that the stream is a time stream.

Any number of time streams can be defined and used in a single delta. For example, a rule-writer might use one time stream for slow speech and another for fast speech, or a rule-writer might use one time stream for actual milliseconds, and another for abstract phonological "time". Consider, for example, a hypothetical language that is like Bambara in all respects except that intervocalic consonants are ambisyllabic. In the following delta for *muso* in this language, an abstract time stream called `time` is used to represent the `s` as ambisyllabic:

```
(28) syllable: |       syl       |     syl    |
     phoneme:  | m  |  u  |    s    |  o  |
     time:     | 1  |  1  | .5 | .5 |  1  |
     duration: | 70 | 120 |   200   | 150 |
```

Note that given such a representation of ambisyllabicity, the context operators can be used to determine the phonemic composition of the syllables. The phonemes that comprise a syllable are those phonemes that are between the left context in the `phoneme` stream of the sync mark before the syllable and the right context in the `phoneme` stream of the sync mark following the syllable.

In the Delta language, a *time expression* is used to refer to a particular point in a time stream. A time expression consists of a time stream name, a pointer name, a plus or minus sign, and an expression representing a quantity of time. For example, given the delta

```
(29) phoneme:       | s   |  o  |   | j   |   a   |
     CV:            | C   |  V  |   | C   | V | V |
     nucleus:  ...  |     | nuc |   |     |  nuc  | ...
     tone:          |     |  H  |   |     |   H   |
     duration:      | 200 | 150 |   | 140 |  200  |
                    1     2     3   4     5   6   7
```

the following time expressions would all refer to the instant midway between `^2` and `^3`:

```
(30) %duration ^2 + 75
(31) %duration ^3 - 75
(32) %duration ^4 - 75
(33) %duration ^5 - 215
(34) %duration ^2 + (.5 * dur(^2...^3))
```

where `dur(^2...^3)` in the last time expression uses the built-in function `dur`, which returns the duration between the specified points in the delta, in this case `^2` and `^3`. In general, the expression after the plus or minus sign in a time expression can be an arbitrarily complex numeric expression.

Delta lets users define a particular time stream as the default stream, so that the stream name can be omitted in time expressions that refer to that stream. Users can also specify in a time stream definition whether time should be measured from left to right or right to left. When time is measured from left to right, a time expression with a positive stream offset, such as expression (30) above, refers to a point that is *n* time units to the right of the sync mark specified by the pointer variable, while one with a negative time offset refers to a point *n* time units to the left. When time is measured from right to left

(as it might be, for example, for a Semitic language, which is written from right to left), a positive time offset specifies a time to the left, and a negative one a time to the right. It will be assumed in the remainder of this paper that the deltas have a single time stream called `duration`, that this stream is the default time stream, and that time is measured from left to right.

Time expressions can be used anywhere ordinary pointer variables can be used—for example, in the range of an insert statement. Thus, given delta (29) above, the statements

```
(35) n = (.4 * dur(^2...^3));

     [%tone _\\^2 H //^3] -> insert [%F0 160] (^2 + n)...(^3 - n);
```

would produce the following result:

```
(36) phoneme:       | s   |        o         |
     CV:            | C   |        V         |
     tone:     ... |           H              | ...
     F0:           |             | 160 |
     duration:     | 200 |  60 |  30 | 60 |
                   ^1    ^2                 ^3
```

Notice that sync marks have automatically been placed at the appropriate timepoints in the delta. In general, when a time expression is used where a sync mark is required, the sync mark is automatically created in the time stream.

Delta has the special operator `at` for placing a token at a single point in time. Thus if instead of the rule in (35), the rule

```
(37) [%tone _\\^2 H //^3]
               -> insert [%F0 160] at (^2 + (.5 * dur(^2...^3)));
```

had been executed, the result would be the following:

```
(38) phoneme:       | s   |        o         |
     CV:            | C   |        V         |
     tone:     ... |          H               | ...
     F0:           |             | 160 |
     duration:     | 200 |  75 |  0  | 75 |
                   ^1    ^2                 ^3
```

The `at` operator is special in that it will create two sync marks if no sync marks exist at the specified point in time, it will add one sync mark if a single sync mark exists, and it will use the outermost sync marks if several exist. Thus the `at` operator makes it easy to

place tokens in different streams at the same point in time, and synchronize the tokens with each other.

3.9 Forall Loops

The previous sections have included several examples of rules. Each of these rules was restricted to operating at one particular point in the delta. For example, rule (10),

```
(10) [%phoneme _^left <cons> !^ac] -> insert [%CV C] ^left...^ac;
```

is only tested against the first token in the phoneme stream. Usually, however, a rule is meant to apply at all appropriate points across the entire delta. Delta has several control structures for applying rules across the entire delta or selected stretches of the delta. One of the most useful of these is the *forall loop*, which performs the body of the loop each time the test at the top of the loop succeeds.

Consider, for example, the following forall loop, which applies rule (10) across the entire delta.

```
(39) loop forall [%phoneme _^bc <cons> !^ac];
        insert [%CV C] ^bc...^ac;
     pool;
```

The first time the loop is executed, the *forall test* at the top of the loop is tested with ^bc ("before consonant"), the anchor of the test, automatically set to ^left. If a consonantal phoneme follows ^bc, the loop body is executed, synchronizing a C token in the CV stream with the phoneme. After execution of the loop body, and also whenever the forall test fails to match, the *advance pointer* ^bc is automatically advanced to the next sync mark, and the test is repeated. This advancing is continued until the advance pointer hits the rightmost sync mark in the delta, in which case the loop terminates.

Assume that loop (39) is being applied to the following delta:

```
(40) phoneme: | m | u | s | o |
     CV:      |               |
              1   2   3   4   5
```

First, ^bc would be set at sync mark 1. The forall test would succeed, setting ^ac at sync mark 2, and the body of the loop would insert a C token between ^bc and ^ac:

```
(41) phoneme: | m | u | s | o |
     CV:      | C |           |
              1   2   3   4   5
              ^bc ^ac
```

Then ^bc would be advanced to sync mark 2. The forall test would fail, since a vocalic, rather than consonantal, phoneme follows. Pointer ^bc would then be advanced to sync mark 3, the forall test would succeed (setting ^ac at sync mark 4), and the body of the loop would insert a C token:

```
(42) phoneme: | m | u | s | o |
     CV:      | C |   | C |   |
              1   2   3   4   5
                      ^bc ^ac
```

The loop would continue in this fashion until ^bc reaches sync mark 5.

Users can override the system's default assumptions about forall loop application by specifying the initial setting of the advance pointer, where the advance pointer should start from on subsequent iterations of the loop, a particular pointer to advance, the stream through which to advance, in which direction to advance (left to right or right to left), and so on. In general, the forall loop is a powerful construct with which users can specify precisely how their rules should apply.

3.10 Token Variables

The program fragments in previous sections have included several examples of pointer variables and one example of a numeric variable (example (35)). In addition, Delta provides *token variables*, which hold entire tokens. The following forall loop uses a token variable to replace with a single phoneme all token pairs consisting of two identical vocalic phonemes. For example, it replaces two adjacent a's with a single a:

```
(43) loop forall [%phoneme _^bv <~cons> !$vowel $vowel !^av];
         insert [%phoneme $vowel] ^bv...^av;
     pool;
```

Assume that this loop is being applied to the following delta:

```
(44) phoneme: | j | a | a | b | i |
```

(The two a tokens might be used in the initial underlying representation for *jaabi* to represent the long vowel [a:], as discussed below.) The forall test first looks for a vocalic

(<~cons>) phoneme following ^bv ("before vowel"). It will succeed when ^bv precedes the first a. The expression !$vowel puts a copy of the first a token in the token variable $vowel. (Token variable names in this paper always start with a dollar sign.) The next expression, $vowel, tests the next token to see whether it is the same as the token in $vowel—that is, whether it is also a. If so, ^av is set after this vowel, and the body of the loop inserts a copy of the token in $vowel between ^bv and ^av, thereby replacing the two vowels and their intervening sync mark with a single a:

```
(45) phoneme: | j | a | b | i |
```

3.11 Fences

It is often necessary to prevent a delta test from crossing a linguistic boundary, such as a morph boundary. Delta has a special construct, called a *fence*, for limiting the scope of a delta test. A fence is a pair of built-in pointer variables, ^lfence and ^rfence. ^lfence delimits the left side of the fence, and ^rfence delimits the right.

Fences are often used in conjunction with forall loops. For example, the forall loop below, whose body is itself a loop, adds a fence to loop (43) to restrict it to operating only on identical vowels that are within the same morph (see below for a simpler, unnested loop that accomplishes the same thing):

```
(46) loop forall [%morph _^lfence <> !^rfence];
         loop forall [%phoneme _^bv <~cons> !$vowel $vowel !^av]
                                              advance from ^lfence;
             insert [%phoneme $vowel] ^bv...^av;
         pool;
     pool;
```

The empty angle brackets in the outer forall loop match any token in the morph stream. Thus the outer loop surrounds the morph with ^lfence and ^rfence. For each morph, the inner forall loop reduces two identical vowels to a single one. The expression advance from ^lfence initializes the advance pointer, ^bv, to ^lfence, the beginning of the morph matched by the outer loop. The forall test in the inner loop will fail if it has to cross one of the fence variables in order to succeed. Thus it will not match vowels across a morph boundary.

In loop (46), the sole purpose of the outer forall loop is to set a fence. An alternative way to set the fence that does away with the outer loop altogether is to include the option `fence %morph` after the delta test in the inner loop:

```
(47) loop forall [%phoneme _^bv <~cons> !$vowel $vowel !^av]
                                                    fence %morph;
          insert [%phoneme $vowel] ^bv...^av;
     pool;
```

The expression `fence %morph` prevents the forall test from crossing a morph boundary, just as the outer loop did in the previous example.

3.12 Conclusion

This section has presented the delta data structure, and has shown various ways in which this data structure can be tested and manipulated. The concepts and constructs presented—streams, tokens, sync marks, contexts, delta tests, one-point and two-point insertions, time streams, forall loops, fences, and others—are by no means a complete inventory of Delta language features; rather, they have been strategically selected for presentation in order to provide enough information to follow the sample programs in the next two main sections, which show applications of the Delta language to Bambara and English.

4 Modeling Tone and Fundamental Frequency Patterns in Bambara

This section presents some sample programs that illustrate how Delta can be used to formalize and test a model of Bambara tone realization. This model is actually an integration of two separate models, a model of phonological tone assignment based on the work of Rialland and Sangare (1985), and a model of the phonetic realization of the phonological tones based on the work of Mountford (1983). Only the facts concerning monosyllabic, bisyllabic, and trisyllabic non-compound words in Bambara are considered. Words with four syllables exist, but are rare.

4.1 A Model of Bambara Tone Assignment

Each Bambara word has an inherent (unpredictable) tone pattern. Monosyllabic and bisyllabic words either have a low or a high tone on all the syllables. For example,

the word *muso* is inherently low-toned ([mùsò]), and the word *jaabi* is inherently high-toned ([já:bí]). The reason that *muso* was shown throughout this paper with a high tone on the final syllable will become clear in the ensuing discussion.

A trisyllabic morph can have one of several patterns: 1) It can be inherently high-toned or low-toned in the same way as the monosyllabic and bisyllabic words are. For example, the morph [gàlàmà] 'ladle' is inherently low-toned, while [súngúrún] 'girl' is inherently high-toned. (In the transcriptions, a word-final [n] or an [n] before a consonant marks the preceding vowel as nasalized.) 2) It can have the pattern low-high-low, as in [sàkénè] 'lizard'. 3) It can have the pattern low-high or high-low. In this case, it is optional whether the first tone is associated with the first syllable and the second tone with the last two syllables, or the first tone with the first two syllables and the last tone with the last syllable. For example, the high-low morph *mangoro* 'mango' can be realized as [mángòrò] or [mángórò]. Only the second realization will be considered for purposes of the program shown below.

In addition to the inherent tones of morphs, Rialland and Sangare posit two other kinds of tones in underlying representations of Bambara utterances: 1) a floating low tone that follows definite noun phrases, as illustrated in delta (1) above, and suggested by the work of Bird (1966), and 2) a floating high tone suffix that follows all content morphs (as opposed to function morphs).

A floating low tone definite marker is not associated with any particular syllable in surface representations; it serves to trigger tonal downstep at the phonetic level, the lowering in fundamental frequency of following high tones.

The high tone suffix for content morphs is realized on different syllables, depending on whether the content morph is part of a definite noun phrase—that is, on whether a floating low tone immediately follows the content morph. If there is no floating low tone, the high tone is realized on the first syllable of the morph following the content morph, if there is one. Otherwise, it is realized on the last syllable of the content morph itself.

For example, consider the phrase *muso don* 'It's a woman', which has an indefinite noun phrase, and hence, no floating low tone. In this phrase, the high tone associated with *muso* is added to the inherently low-toned *don*, producing the pattern

[mùsò dôn], where the circumflex accent ˆ represents the tone pattern high-low. (Here, the content tone has been added to the tone of the one-syllable morph. In a polysyllabic morph, the high tone simply replaces the inherent tone associated with the first syllable.) On the other hand, the phrase *muso don* 'It's the woman', which has a definite noun phrase, is realized as [mùsó dòn]. See Rialland and Sangare (1985) and Bird, Hutchison, and Kanté (1977) for a fuller description and analysis of Bambara tone patterns.

4.2 Formulation of the Tone Model in Delta

According to the model just presented, Bambara tone patterns consist of sequences of high and low tones. Each tone is either an inherent part of a morph, a floating definite marker, or a floating content marker, as illustrated in Figure 3, which shows several underlying representations of Bambara utterances in delta form. In the transcription in example 5 in the figure, the wedge ˇ represents the tone pattern low-high.

****

Insert Figure 3 here

****

Given underlying forms like these, the correct tone pattern can be assigned in two main steps:

1. Assign each floating H tone to the appropriate morph. Attach it to the end of a preceding morph if a floating L tone follows. Otherwise, attach it to the beginning of the following morph.

2. Merge unordered pairs of syllable and tone sync marks from right to left until there are no unordered pairs left, thereby creating the appropriate synchronizations between tones and syllables.

Step 1 would create the forms shown in Figure 4 for the sample deltas in Figure 3:

****

Insert Figure 4 here

****

Step 2 would operate on the deltas in Figure 4 to produce those shown in Figure 5.

****

Insert Figure 5 here

****

Note that this step does not change delta 1, since this delta contains no sync marks in the `syllable` and `tone` streams that are unordered with respect to each other. In delta 5, the right to left merging of sync marks correctly leaves the first syllable synchronized with two tones, `L` and `H`. Left to right merging of the sync marks would produce the wrong result.

A final step might be to combine adjacent `H` tones and `L` tones into single `H` and `L` tones, but it does not make any difference for purposes of the program below that generates fundamental frequency patterns on the basis of the tone patterns whether a single tone is associated with several syllables, or each of the syllables has its own tone.

Step 1 can be accomplished in Delta by the forall loop shown in Figure 6.

****

Insert Figure 6 here

****

The forall test

```
(48) ([%tone _^bh H !^ah] & [%morph _^bh ^ah])
```

contains two parts. The first matches any `H` token after `^bh`, setting `^ah` after it. The second part tests whether `^bh` and `^ah` point at adjacent sync marks in the `morph` stream—i.e., whether the `H` token matched by the first part is a floating tone.

The second step, which merges the appropriate sync marks in the `syllable` stream with those in the `tone` stream can be expressed as the forall loop shown in Figure 7.

****

Insert Figure 7 here

****

The outer forall loop invokes the inner *simple loop* for each token in the `morph` stream. A simple loop continues to repeat until a statement in the loop body, in this case an *exit statement*, causes the loop to terminate. The loop moves pointers `^bt` and `^bs` right to left through the morph, merging the appropriate sync marks, and terminating whenever `^bt` or `^bs` equals `^bm`—that is, when one of these pointers reaches the beginning of the morph.

4.3 Building Underlying Representations Using Dictionaries

The program fragments shown above assume an underlying representation in which each morph is associated with a tone pattern, and floating high and low tones are present where appropriate. A Delta program designed to test the above program fragments would have to build the appropriate underlying representations in the first place. To prevent users from having to enter underlying representations for the utterances they wish to test (a tedious task), the program could be designed such that the user only has to specify phoneme names, morph boundaries, and the unpredictable low floating tones. For example, our sample utterance *muso jaabi* might be entered as follows:

```
(49) + m u s o ` + j a a b i +
```

where + designates a morph boundary and ` a floating low tone. Two `a` tokens in succession represent the long vowel [a:], as discussed earlier. On the basis of this information, the program could easily build the following structure:

```
(50) morph:   |      root      |   |      root      |
     phoneme: | m | u | s | o |   | j | a | a | b | i |
     CV:      | C | V | C | V |   | C | V | V | C | V |
     tone:    |               | L |                   |
```

The next step would be to insert the inherent morph tones and the floating high content tones. Since neither the inherent tones nor the content tones are predictable from this delta, a dictionary must be used. Delta has two kinds of dictionaries, *action dictionaries* and *set dictionaries*, or simply *sets*, both of which are useful for creating underlying representations from forms like the above.

An action dictionary contains token name strings (henceforth "search strings") and associated actions. An action for an entry, which can consist of any legal Delta statements, is automatically performed whenever a search string is matched—that is, an identical token name sequence is looked up in the dictionary. For our purposes, an action dictionary named `morphs` might be defined to contain all morphs (represented in terms of phoneme names) and associated tone patterns.[6] More specifically, the action for each morph might be an insert statement that inserts the appropriate tones for the morph into the delta, as shown in Figure 8.[7]

****

Insert Figure 8 here

****

An action dictionary always has four pointer variables associated with it, in this case `^b`, `^e`, `^1`, and `^2`. The first two pointer variables take on the values of the pointers delimiting the token name sequence being looked up. The last two pointer variables are special pointers that can be used inside the dictionary search strings to isolate parts of the dictionary entries to which the associated actions need to refer. These special pointers are not used in this example, and can be ignored. See Hertz et al. (1985: 1597-1598) for an example of their use.

---

6. In earlier papers about Delta, the sample action dictionaries were unnamed, reflecting the then-current version of the system in which it was only possible to define a single, unnamed action dictionary. Delta has been enhanced to allow rule-writers to use any number of named action dictionaries.

7. Bambara contains sets of morphs that differ phonetically only in their tone pattern. The dictionary action for such `morphs` might prompt the user for the intended tone pattern. Alternatively, the user might annotate the input string with the intended tone pattern, or, in some cases, the system might be able to perform some syntactic analysis to determine the correct pattern automatically.

The following forall loop operates on a delta of the form shown in delta (50) above, looking up in dictionary `morphs`, the phoneme names associated with each morph in the delta:

```
(51) loop forall [%morph _^bm <> !^am];
         find %phoneme ^bm...^am in morphs;
     pool;
```

If the phoneme sequence is found, the dictionary automatically synchronizes the appropriate tones with the morph, creating, for example, the following deltas for *muso jaabi* and *mangoro don*:

```
(52) morph:    |      root      |   |      root       |
     phoneme:  | m | u | s | o |   | j | a | a | b | i |
     tone:     |       L        | L |       H         |

(53) morph:    |          root          |   |  root  |
     phoneme:  | m | an | g | o | r | o |   | d | on |
     tone:     |   H    |     L     | L |   L    |
```

The next step would be to insert the floating high content tones after the content morphs. This step could be accomplished by adding the appropriate insert statement to the action of all content morphs in dictionary `morphs`. However, since there are so many content morphs, a simpler strategy would be to "mark" the non-content morphs as function morphs, and insert the content tone for all morphs that are not function morphs. The simplest way to mark the appropriate morphs as function morphs is to place them in a set.

A set contains search strings, but no actions. For example, one might define a set called `function_morphs` as follows:

```
(54) set function_morphs contains %phoneme: a, d on, ...;
```

(Since all of the morphs in set `function_morphs` are also in the action dictionary `morphs`, an alternative to the above set statement could be used in which each function morph in the action dictionary is followed by an expression that places the morph in set `function_morphs`.) Given set `function_morphs`, a rule could be added to forall loop (51) above to insert a `H` tone after any morph not found in the set:

```
(55) loop forall [%morph _^bm <> !^am];
          ...
          ~find %phoneme ^bm...^am in function_morphs
                         -> insert [%tone H] ^am...  project;
     pool;
```

The option `project` at the end of the insert statement specifies that the new sync mark after the inserted `H` tone should be projected into all streams in which `^am`, the range pointer, is defined. This rule would operate on deltas (52) and (53) to produce the following underlying representations for *muso jaabi* and *mangoro don*:

```
(56) morph:     |      root      |   |   |      root        |   |
     phoneme:   | m | u | s | o |   |   | j | a | a | b | i |   |
     CV:        | C | V | C | V |   |   | C | V | V | C | V |   |
     tone:      |       L        | H | L |        H         | H |

(57) morph:     |          root          |  .  |   | root  |
     phoneme:   | m | an | g | o | r | o |     |   | d | on |
     CV:        | C | V  | C | V | C | V |     |   | C | V  |
     tone:      |     H      |     L      | H | L |   L     |
```

A final step in creating the underlying representation would be to reduce sequences of identical vocalic phonemes that are not separated by a morph boundary, such as the `a a` of *jaabi*, into a single phoneme, as shown earlier in example (47).

4.4 A Model of Bambara Fundamental Frequency Patterns

On the basis of the `CV` and `tone` streams, the appropriate F_0 pattern for the tones can be determined. This section considers one possible strategy for F_0 determination, based on the work of Mountford (1983). According to Mountford, high and low tones descend along relatively straight and independent lines, as shown schematically in the diagram of a sentence with the tone pattern H L L H L H L in Figure 9.

****

Insert Figure 9 here

****

The starting and ending frequencies of the baseline are relatively independent of the sentence duration. Thus the slope of the baseline varies with sentence duration. The sample program below uses a starting frequency of 170 hertz and an ending frequency of 130 hertz for the baseline. The starting and ending frequencies of the topline are a function of sentence duration. For the sake of simplicity, this detail is ignored in the

sample program, which assumes a starting frequency of 230 hertz and an ending frequency of 150 hertz.[8]

My own very preliminary laboratory studies of Bambara tone patterns suggest that the F_0 tone targets are generally realized halfway through each syllable nucleus—that is, long vowels behave the same way as short vowels for the purpose of F_0 target placement. Furthermore, the F_0 targets tend to have no durations of their own, serving only to shape the overall F_0 contour.

4.5 Formulation of the Fundamental Frequency Model in Delta

The model of F_0 assignment just outlined can easily be implemented in Delta, as shown by the program fragment in Figure 10, which generates a descending topline and a descending baseline, and determines the appropriate F_0 values along those ramps. The program assumes that each phoneme was given a duration by earlier statements.

****

Insert Figure 10 here

****

The forall loop in this program would generate the following values for our sample delta for *muso jaabi* (assuming the segment durations shown and a sentence duration of 1080 milliseconds):

```
(58) morph:     |           root              |   |           root
     phoneme:   | m |    u      | s |    o    |   | j |    a     |
     CV:        | C |    V      | C |    V    |   | C | V  |  V  |
     nucleus:   |   |   nuc     |   |   nuc   |   |   |    nuc   |   ...
     syllable:  |     syl       |     syl     |   |     syl      |
     tone:      |      L        |      H      | L |              H
     F0:        |      |165|    |     |  |195|  |   |     |  |172|  |
     duration:  |70 |60| 0 |60 |200|75| 0 |75|   |140 |95| 0 |95|
```

The program has been oversimplified for purposes of illustration. For example, it does not handle single syllables that have two tones, such as *don* in utterance 1 in Figure 5 above, nor does it compute the starting and ending frequencies of the high tone line as

---

8. Current research on downtrend in other languages suggests that this model may be oversimplified. Since Delta has fully general numeric capabilities, it should be equally well-suited for expressing other algorithms for computing F_0 values.

a function of sentence duration. Furthermore, it does not handle the tone raising and lowering phenomena that occur in Bambara for tones in particular contexts. Rialland and Sangare (1985) posit a rule that downsteps (lowers) a high tone and all subsequent high tones after a floating low tone. See also Mountford (1983) for some posited raising and lowering rules. Our sample program could easily be expanded to handle all of these phenomena.

In addition to F_0 values, a program designed for synthesis would have to compute or extract from a dictionary the values for other synthesizer parameters. Delta's *generate statement* can be used to create a file of parameter values and durations for the synthesizer on the basis of the parameter streams and an associated time stream:

```
(59) generate (%duration, %F0, %F1, %F2, %F3, ...);
```

Also, users can tailor their program to their particular synthesizer setup by writing their own C programs to generate values for the synthesizer, and their own synthesizer driver. The ability to interface C programs at will also lets Delta be used in conjunction with demisyllable and diphone libraries.

5 Modeling English Formant Patterns

The section just completed illustrated how Delta can be used to formalize and integrate a phonological model of tone assignment and a phonetic model of tone realization. This section further demonstrates Delta's flexibility by drawing from my own work on modeling English formant patterns to show how different hypotheses I have come up with over the years can be formulated in Delta. The section focuses on my most recent model, in which formant targets and intervening transitions are represented as independent durational units that are related to higher-level phonological constituents in well-defined ways. This model relies critically on the concept of synchronized units at both the phonological and phonetic levels.

5.1 Model 1: Linear Utterance Representations, Implicit Transitions

The first work I did on modeling English formant patterns was in the context of my students' and my synthesis rule development. Our SRS synthesis rules for English, developed between 1978 and 1982, are based on the hypothesis that every phoneme (i.e.,

phoneme-sized unit) has an intrinsic duration that is modified according to such factors as segmental context and stress. Formant targets (usually two of them) are set in relation to the segment's duration—for example, 20% and 80% of the way through the segment.[9] For the most part, all durational adjustments, such as stretching before voiced segments, are made between the formant targets within given segments, so that the durations of the formant transitions between targets in adjacent segments remain constant.

5.2 Model 2: Multi-level Utterance Representations, Implicit Transitions

In the course of implementing a set of SRS rules based on the model just outlined, we realized that our formant target and duration rules might be simplified if we treated certain sonorant sequences that act as a single syllable nucleus, such as [ei], [ai], and [ar], as two units for the purpose of assigning formant targets, and as single units for other purposes, such as assigning amplitude patterns. (A syllable nucleus in this model consists of a vowel + a tautosyllabic sonorant, if such a sonorant exists.) Such a structure was impossible to represent straightforwardly with SRS, which relies on linear utterance representations, but is quite simple to represent with Delta, as shown below for the word *ice*:

```
(60) syllable: |     syl     |
     nucleus:  |  nuc  |     |
     phoneme:  | a | i | s |
```

By representing the [i] of syllable nuclei like [ai] as an independent phoneme token, the rules can assign a single target value for each formant—say, an F_2 value of 2000 hertz—to all `i`'s, regardless of their context. Syllable nuclei can still be treated as single units where appropriate.

While this model simplifies the prediction of formant values, however, it leads to complicated rules for positioning the formant values with respect to the edges of segments, since the formant target positions for a token in one context (e.g., `i` as the sole component of the nucleus) are not necessarily the same as those in another (e.g., `i` at the end of a diphthong).

---

9. In all our SRS rules, not just those for English, some segments only have a single target for a particular formant, and others, like [h] in English and [r] in Japanese, have no targets for some formants. Non-initial English [h] is modeled by linear transitions connecting the last formant targets in the preceding segment and the first in the following segment.

5.3 Model 3: Multi-level Representations, Explicit Transitions

My recent durational studies support the view that formant transitions are relatively stable in duration. To avoid the complicated target placement rules alluded to above, however, I am exploring a new model, in which the transitions are represented as independent durational units, as shown below for the word *ice*:

```
(61) syllable: |             syl                |
     nucleus:  |       nuc       |     |       |
     phoneme:  |  a    |     | i   |     |  s    |
     F2:       | 1400  |     |2000 |     | 1800  |
     duration: |  45   | 90  | 30  | 40  |  90   |
```

Note that the formant transitions are represented by adjacent sync marks in the `phoneme` and `F2` streams, with intervening durations in the `duration` stream.

In this model, each formant target is synchronized with an entire phoneme token, and the duration-modification rules modify entire phoneme durations. In diphthongs, such as [ai], the duration rules modify only the first portion of the diphthong (e.g., the [a] of [ai]), thereby keeping the duration of the transition portion of the diphthong (e.g., from [a] to [i]) constant.

Below are some forall loops that might be used to insert formant values and formant transitions, and to modify phoneme durations in the appropriate contexts, building a delta like (61) above. These examples assume that earlier statements have inserted the appropriate initial phoneme durations so that the delta for *ice* looks as follows before the loops apply:

```
(62) syllable: |     syl      |
     nucleus:  |   nuc    |   |
     phoneme:  | a  | i   | s |
     F2:       |           |
     duration: | 75 | 30  | 90 |
```

The following forall loop matches each phoneme, synchronizing a value for each formant with the phoneme:

```
(63) loop forall [%phoneme _^1 <> !^2];

     :: F2 values:

        if
         [%phoneme _^1 high]     -> insert [%F2 2000] ^1...^2;
         [%phoneme _^1 low]      -> insert [%F2 1400] ^1...^2;
         [%phoneme _^1 alveolar] -> insert [%F2 1800] ^1...^2;
        fi;
       ...
     pool;
```

This loop operates on delta (62) to produce the following:

```
(64) syllable: |         syl         |
     nucleus:  |      nuc      |      |
     phoneme:  |  a   |  i   |  s   |
     F2:       | 1400 | 2000 |1800  |
     duration: |  75  |  30  | 90   |
```

The next loop modifies the durations of vowels in certain contexts. In the case of delta (62), it shortens the duration of a to 60% of its previous duration, since it is in a nucleus that precedes a tautosyllabic voiceless segment. The forall option `fence %syllable` prevents the delta tests in the loop body from crossing a syllable boundary.

```
(65) loop forall [%phoneme _^bv <~cons> !^av] fence %syllable;

        [%nucleus _^bv <nuc> [%phoneme <~voic>]]
                                    -> dur(^bv...^av) *= .6;
        ...
     pool;
```

where the expression

```
dur(^bv...^av) *= .6
```

is a shorthand for

```
dur(^bv...^av) = .6 * dur(^bv...^av)
```

This loop shortens the a in delta (64) from 75 milliseconds to 45:

```
(66) syllable: |         syl         |
     nucleus:  |      nuc      |      |
     phoneme:  |  a   |  i   | s    |
     F2:       | 1400 | 2000 |1800  |
     duration: |  45  |  30  | 90   |
```

A more complete program would modify the duration of consonants as well, depending, for example, on their position in the syllable and whether the syllable is stressed.

Finally, the following forall loop inserts transitions between adjacent phonemes:

```
(67) loop forall [%phoneme _^1 <> !^2];
       if
         [_^1 <sonorant> <obstruent>]
               -> insert [%duration 40] ^2...  project;
         [_^1 <sonorant> <sonorant>]
               -> insert [%duration 90] ^2...  project;
         ...
       fi;
     pool;
```

This loop operates on delta (66) to produce the following:

```
(68) syllable: |             syl                |
     nucleus:  |        nuc          |     |     |
     phoneme:  |  a    |     |  i    |     |  s  |
     F2:       | 1400  |     | 2000  |     | 1800|
     duration: |  45   | 90  |  30   | 40  |  90 |
```

The transition rules presented here are hypothetical; at the time of writing (December 1987), I have not yet investigated in any systematic way what factors determine the transition durations. The transition durations probably depend as much on the place of articulation of the two phonemes as on their manner, or possibly even directly on the formant values they connect, or on other as yet to be discovered factors.

Independent developments in autosegmental CV theory (e.g., Clements and Keyser, 1983) have suggested some minor revisions to our current model, as illustrated by the following representation for *ice*, in which the diphthong-final `i` is differentiated from the vowel `i` by virtue of being synchronized with a `C`, rather than with a `V`:

```
(69) syllable:  |             syl                |
     nucleus:   |        nuc          |     |     |
     CV:        |  V    |     |  C    |     |  C  |
     phoneme:   |  a    |     |  i    |     |  s  |
     F2:        | 1400  |     | 2000  |     |1800 |
     duration:  |  45   | 90  |  30   | 40  |  90 |
```

The remainder of this discussion will assume a representation like the one in delta (69).

Given a `CV` stream, one might hypothesize that every `C` and `V` is associated with a single value for each formant, reflecting the segment's place of articulation. However, this assumption immediately gets us in trouble with [h], which Nick Clements and I studied recently over a period of two semesters. [h] exhibits no targets of its own; rather, it acquires its formant structure on the basis of the preceding and following segments, its formants looking very much like aspirated transitions from the last formant target in the

preceding segment to the first formant target in the following segment. Consider, for example, the schematic representation of the second formant pattern of the sequence [g h a͡i s], as in *big heist*, in Figure 11.

****

Insert Figure 11 here

****

The traditional way to segment this utterance would be into the four chunks shown in the figure. A problem caused by this segmentation, however, becomes apparent when we consider the second formant pattern of the sequence [g a͡i s], the same sequence without the [h], shown in Figure 12.

****

Insert Figure 12 here

****

The second formant patterns of the two sequences are for all intents and purposes identical, except that the one with [h] has aspiration, rather than voicing, during the transition from the [g] to the target for the [a]. In order to write rules assigning a duration to the [a] in each of these examples, one would have to posit a rule that shortens the vowel after [h]. Such a rule would be very unusual in English, since modifications to English vowel durations are not generally triggered by preceding segments. Furthermore, in order to generate the F_2 pattern in the shortened vowel, one would have to move the target by precisely the amount that the vowel has been shortened.

Given the transition model posited above, we might hypothesize instead that [h] adds no duration of its own, but rather, is realized by aspiration superimposed on the transition between the preceding and following segment, as illustrated in the following delta for *big heist*:

(70)

syll:						syl						
nucleus:							nuc					
CV:		C		C		V				C		
phoneme:	...	g				a				i		...
aspiration:				asp								
F2:		2000				1400				2000		
asp_amp:				15								
duration:		50		60		45		90		30		

where asp_amp is a stream containing amplitude values for aspiration. Here [h] is represented as a C in the CV stream, with an associated asp token in an aspiration stream, but no associated phoneme token. Since the tokens in the phoneme stream have values for place and manner of articulation, among others, h in this representation is phonologically unspecified for these attributes. The absence of h in the phoneme stream simplifies the transition rules, which can refer directly to the sequence g a in determining the transition duration.[10] (The C is necessary in the CV stream, since /h/ functions phonologically as a consonant, preventing flapping of a preceding /t/ or /d/, for example.)

This model makes the prediction that the duration of the aspiration associated with h will be no greater than the duration of the transition between segments. For many speakers, however, [h] seems to consistently add a small amount of duration, on the order of 25 milliseconds, to the transition. For such speakers, an additional rule would be required that adds a fixed duration increment for h to the transition.

An obvious question that arises is whether aspiration in stops might be represented in a way similar to that for h, as in the following delta for *tie*:

```
(71) syllable:   |                 syl                  |
     nucleus:    |       |       |          nuc         |
     CV:         |   C   |       |   V   |      |   C   |
     phoneme:    |   t   |       |   a   |      |   i   |
     aspiration: |       |  asp  |                      |
     F2:         |  1800 |       |  1400 |      |  2000 |
     duration:   |   70  |  60   |   75  |  90  |   30  |
```

Note that the token asp in this case is not associated with a C or V in the CV stream, reflecting its different phonological status from that of h. Note also that the asp token is not associated with any phoneme, which completely eliminates the much-debated problem of whether aspiration of stops in English should be considered for segmentation purposes as part of the preceding stop or part of the following vowel.[11]

---

10. The phonetic properties of [h] provide excellent evidence that /h/ must be treated as phonologically underspecified, a point made by Keating (1985).

11. It is interesting to note that Peterson and Lehiste (1960) found that when aspiration is considered to be part of the vowel, the vowel is lengthened by about 25 milliseconds, the same duration increment we found for [h].

This model is undoubtedly oversimplified in several ways. For example, aspiration does not always coincide with the entire transition; sometimes it stops before the end of the transition and sometimes after. Also, transitions for different formants do not always reach their targets at the same time. The model could be extended to handle these cases by adding additional duration tokens at the appropriate points. For example, a transition that is only partially aspirated could be modeled as two duration tokens, the first synchronized with an `asp` token in the `aspiration` stream, and the second with a voicing token in the `voicing` stream (not shown above).

Certainly the model has been oversimplified in other ways as well. The discussion has not been meant to give a full account of this model or even to be correct in all the details presented; the model is still very much under development at the time of writing. Rather, the discussion is intended to show how a model of this type, which relies heavily on synchronized phonological and phonetic units on different levels, can easily be accomodated in Delta, allowing linguists to test their ideas and revise them in the ways they see fit.

6 Final Remarks

This paper has presented selected features of the Delta programming language and two linguistic models formulated in that language, one for Bambara tone and fundamental frequency patterns and one for English formant patterns. The paper has focused on Delta's central data structure, which represents utterances as multiple streams of synchronized units. This data structure gives rule-writers considerable flexibility in expressing the relationship between phonological and phonetic units.

The examples of programming statements in the paper have been meant to give the flavor of the Delta language, rather than to describe all possible Delta constructs. No complete programs were presented, other than the oversimplified program in Figure 2. Unlike the program in that figure, Delta programs generally consist of several procedures. Variables in Delta can either be local to a particular procedure or global to all procedures.

Delta has several kinds of statements not described in the paper. For example, in addition to the insert, delete, and merge statements for modifying deltas, Delta has a *mark statement* for changing the attributes of tokens and a *project statement* for projecting an existing sync mark into a new stream; and in addition to the forall loop and simple loop,

Delta has a *while loop*, which performs the body of the loop so long as the test at the top of the loop succeeds. Delta also has far more flexible input and output facilities than the simple read and print statements illustrated, and it has fully general numeric capabilities.

Delta tests can also include many kinds of expressions not illustrated, including expressions that test for optional occurrences of a pattern in the delta, for one of a set of alternative patterns, and for a pattern that must *not* be present for the test to succeed. Delta tests can even be temporarily suspended in order to execute arbitrary statements, a capability that adds enormous power to the language.

Delta also has many kinds of tests not illustrated at all, including procedure calls, which succeed or fail according to the form of the return statement in the called procedure. A particularly nice feature of the Delta language is its ability to group tests of different types into a single test—such as a delta test, a dictionary lookup, and a procedure call, making a single action dependent on the success of several tests.

The interactive source-level debugger has not been illustrated at all, but it is an essential part of the system that greatly enhances the system's utility as a rule-development tool. With the debugger, users can issue commands to temporarily stop execution each time the delta changes, each time a particular variable changes, each time a particular line in the program is executed, each time a particular procedure is called, and so on. When the program stops, users can examine the contents of the delta, the contents of program variables, the contents of the run-time stack, and so on; they can also modify the delta, making any of the delta manipulations allowed by the Delta language. The debugger also gives users very flexible facilities for specifying the files that programs should read from and print to, and it lets users create logs of their terminal session and their program input and output.

A particularly interesting use of the debugger is as a well-formedness checker. Commands could be issued to the debugger that cause it to execute a particular procedure each time the delta or a particular stream in the delta changes. This procedure could test the delta to make sure that whatever manipulations have been made to it do not violate any well-formedness conditions of the language in question. Furthermore, the debugger could be instructed to print out any statement that causes such a condition to be violated, making it easy for rule-writers to find out exactly what parts of their program cause such constraints to be violated.

The debugger can be invoked interactively or via an *execcmd statement* in a Delta program. A program might use an execcmd statement, for example, to invoke the well-formedness commands discussed above, so that the program automatically tests for any violations of well-formedness conditions whenever it is executed.

At the time of writing (December, 1987) most, but not all of the features of Delta described in this paper have been implemented. At the time of publication, we expect to have implemented all of these features and to have begun implementing those planned for the next version of the system. This version will let users construct and test arbitrary data structures, such as trees, arrays, and graphs. A user might, for example, choose to represent the tokens in a particular stream as trees of features, as in the multi-dimensional model of autosegmental phonology proposed by Clements (1985).[12] We also plan to add a macro processor to the system, which will let users tailor the syntax of their program statements to their particular needs.

In addition to enhancing the existing system, we are considering implementing a new rule interpreter that would let linguists interactively enter, test, and modify rules without recompiling the rules, much like SRS (Hertz, 1982) does. Like SRS, the system would be oriented to the novice rule-writer with no prior programming experience, and would have several built-in assumptions about how the rules should apply—for example, top to bottom through the rules, left to right across the delta. Unlike SRS, however, the rules accepted by the system would be in Delta form, and would thus be able to build and manipulate multi-level structures (deltas). This rule interpreter would be particularly useful as an instructional tool for beginning rule-writers, and would be a good stepping stone to the full-fledged system. At a later time, we may implement an interpreter for the entire Delta language.

Because the Delta System is still under development at the time of writing, little has been said about its performance or ease of use. We are encouraged, however, by our experience with an early version of the system. Students in speech synthesis classes felt comfortable writing programs with this version in anywhere from a few days to a few weeks. While large programs written with this version do not generally run in real-time,

---

12. In the current system, rule-writers can use C to construct data structures of their choice. For example, they could construct tree-structured feature representations for tokens by using a numeric field in a token as a pointer to a tree structure defined in C.

we have designed the current version of the system with real-time rule execution as a primary goal.

An immediate plan of mine is to use the Delta System to further explore the model of English formant transitions hypothesized above. (It is possible that at the time of publication of this paper, the model will have been altered in significant ways.) Users elsewhere plan to use Delta to explore many other kinds of models, including articulatory ones. Because of its ability to accomodate different models, the Delta System should help us increase our understanding of phonology and phonetics, and learn more about the interface between the two.

Acknowledgements

The author is the President of Eloquent Technology. The Delta System is an Eloquent Technology product, developed in cooperation with the Department of Modern Languages and Linguistics and the Computer Science Department at Cornell University. Several people have contributed to the system. Kevin Karplus and Jim Kadin continue to be instrumental in its design, and Jim Kadin also in its implementation.

I thank Mary Beckman, Nick Clements, John Drury, Greg Guy, Jim Kadin, Peggy Milliken, and Annie Rialland for their helpful suggestions on earlier drafts of this paper, and John McCarthy for his insightful comments on the version of this paper presented at the First Conference on Laboratory Phonology at the Ohio State University in June 1987. These comments motivated several of the revisions in this paper.

References

Beckman, M., S. Hertz, & O. Fujimura (1983). SRS pitch rules for Japanese. *Working Papers of the Cornell Phonetics Laboratory*, No. 1, pp. 1-16.

Bird, C. (1966) *Aspects of Bambara Syntax*, Ph.D. Dissertation, UCLA.

Bird, C., J. Hutchison, & Mamadou Kanté (1977). *An ka Bamankankan kalan: Introductory Bambara.* Reproduced by Indiana University Linguistics Club.

Chomsky, N. & M. Halle (1968). *The Sound Pattern of English.* Harper and Row, New York.

Clements, G. N. (1985). The Geometry of phonology features. *Phonology* 2.

Clements, G. N. & S. J. Keyser (1983). *CV phonology: a generative theory of the syllable*. Cambridge, Mass., MIT Press.

Hertz, S. (1980). Multi-language speech synthesis. A search for synthesis universals (abstract). *J. Acoust. Soc. Amer.*, V. 67, Suppl. 1.

Hertz, S. (1981). SRS text-to-phoneme Rules: a Three-Level Rule Strategy. *Proc. IEEE Int. Conf. Acoust. Speech, Signal Processing*, pp. 102-105.

Hertz, S. (1982). From text to speech with SRS., J. Acoust. Soc. America 72, No. 4, pp. 1155-1170.

Hertz, S. (1986). English text to speech rules with Delta, *Proc. 1986 IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 2427-2430.

Hertz, S. & M. Beckman (1983). A look at the SRS synthesis rules for Japanese, *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 1336-1339.

Hertz, S., J. Kadin, and K. Karplus (1985). The Delta rule development system for speech synthesis from text. *Proc. IEEE* 73, No. 11, pp. 1589-1601.

Keating, P. (1985). Phonologial patterns in coarticulation, presented at the 60th Annual Meeting of the LSA, Seattle, Washington.

Mountford, K. (1983) Bambara delcarative sentence intonation. *Research in Phonetics*, No. 3, Indiana University, Bloomington.

Peterson and Lehiste (1960). Duration of Syllable Nuclei in English. *J. Acoust. Soc. Amer.*, V. 32, pp. 693-703.

Rialland, A. and Sangare, M. (forthcoming) *Tons Lexicaux et tons flottants dans les noms en bambara,* Actes de la conference de phonologie plurilinéaire (Lyon, June 1985).

```
:: Phoneme Stream Definition:

   stream %phoneme;

   :: Fields and Values:

      name:      m, n, ng, b, d, j, g, p, t, c, k,...,
                 i, in, I, e, en, E, a, an,...u, un;
      place:     labial, alveolar, palatal, velar,...;
      manner:    sonorant, obstruent, ...;
      class:     cons;
      nasality:  nasal;
      voicing:   ~voiced;
      height:    high, mid, low;
      backness:  front, central, back;
      ...

   :: Initial Features:

      m  has labial, sonorant, nasal, cons;
      n  like m except alveolar;
      ...
      j  has voiced, palatal, stop, cons;
      g  like j except velar;
      ...
      u  has back, high;
      un like u except nasal;

   end %phoneme;

:: F0 Stream Definition:

   stream %F0;
      name: int;    :: defines all integers as possible names
   end %F0;
```

Figure 1. Sample Delta Definition.

```
:: Delta Definition:
  <delta definition goes here>
:: Main Body of Program:
   proc main();
     :: Read phonemes from terminal into phoneme stream:
        read %phoneme;
     :: Synchronize a C token with an initial consonantal token:
        [%phoneme _^left <cons> !^ac]
                  -> insert [%CV C] ^left...^ac;
     :: Print the resulting delta:
        print delta;
   end main;
```

Figure 2. Sample Delta Program.

1) 'It's a woman' (surface form: mùsò dôn):

morph:		root						root					
phoneme:	m	u	s	o				d	on				
syllable:	syl		syl					syl					
tone:		L			H			L					

2) 'It's the woman' (surface form: mùsó dòn):

morph:		root						root					
phoneme:	m	u	s	o				d	on				
syllable:	syl		syl					syl					
tone:		L			H	L		L					

3) 'answering the woman' (surface form: mùsó jábí):

morph:		root						root					
phoneme:	m	u	s	o			j	a	b	i			
syllable:	syl		syl				syl		syl				
tone:		L			H	L		H			H		

4) 'It's a mango' (surface form: mángórò dôn):

morph:		root						root					
phoneme:	m	an	g	o	r	o		d	on				
syllable:	syl		syl		syl			syl					
tone:		H			L		H		L				

5) 'It's the lizard' (surface form: sǎkèné dòn):

morph:		root						root					
phoneme:	s	a	k	e	n	e		d	on				
syllable:	syl		syl		syl			syl					
tone:	L		H		L		H	L		L			

Figure 3. Underlying Deltas for Bambara Utterances.

1) 'It's a woman' (surface form: mùsò dôn):

morph:		root				root				
phoneme:	m		u		s		o	d		on
syllable:	syl				syl			syl		
tone:		L						H		L

2) 'It's the woman' (surface form: mùsó dòn):

morph:		root						root		
phoneme:	m		u		s		o	d		on
syllable:	syl				syl			syl		
tone:		L				H		L		

3) 'answering the woman' (surface form: mùsó jábí):

morph:		root						root				
phoneme:	m		u		s		o	j		a	b	i
syllable:	syl				syl			syl			syl	
tone:	L				H			H			H	

4) 'It's a mango' (surface form: mángórò dôn):

morph:		root						root				
phoneme:	m		an	g		o	r		o	d		on
syllable:	syl			syl			syl			syl		
tone:		H				L				H		L

5) 'It's the lizard' (surface form: sǎkèné dòn):

morph:		root							root					
phoneme:	s		a	k		e	n		e	d		on		
syllable:	syl			syl			syl			syl				
tone:	L			H		L	H			L				

Figure 4. Underlying Deltas after Floating High Tone Assignment.

1) 'It's a woman':

morph:		root			root			
phoneme:	m	u	s	o	d	on		
syllable:	syl		syl		syl			
tone:		L			H		L	

2) 'It's the woman':

morph:		root				root		
phoneme:	m	u	s	o		d	on	
syllable:	syl		syl			syl		
tone:	L		H		L	L		

3) 'answering the woman':

morph:		root				root				
phoneme:	m	u	s	o		j	a	b	i	
syllable:	syl		syl			syl		syl		
tone:	L		H		L	H		H		

4) 'It's a mango':

morph:		root				root			
phoneme:	m	an	g	o	r	o	d	on	
syllable:	syl		syl		syl		syl		
tone:		H			L		H	L	

5) 'It's the lizard':)

morph:		root					root			
phoneme:	s	a	k	e	n	e		d	on	
syllable:	syl		syl		syl			syl		
tone:	L	H	L		H		L	L		

Figure 5. Surface Forms Produced by Sync Mark Merging.

```
:: Forall floating H tones (^bh = "before H", ^ah = "after H")...

  loop forall ([%tone _^bh H !^ah] & [%morph _^bh ^ah]);

      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 the original floating H and following sync mark:

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

  pool;
```

Figure 6. Forall Loop for Floating High Tone Assignment.

```
:: For each morph (^bm = "begin morph", ^am = "after morph") ...

  loop forall [%morph _^bm <> !^am];

     ^bs = ^am;  :: Set ^bs (begin syllable) to ^am (after morph)
     ^bt = ^am;  :: Set ^bt (begin tone)     to ^am

     loop

       :: Set ^bt before the next tone token to the left.  If
       :: there are no more tone tokens in the morph (i.e., ^bt
       :: has reached ^bm), exit the inner 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
       :: inner loop.

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

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

          merge ^bt ^bs;

     pool;
  pool;
```

Figure 7. Forall Loop for Sync Mark Merging.

```
dict morphs(^b, ^e, ^1, ^2);

  %phoneme:

    :: morphs with low tones:

       a,                    :: 'he'
       b a l a,              :: 'porcupine'
       d on,                 :: 'this is'
       g a l a m a,          :: 'ladle'
       m u s o,              :: 'woman'
       ...
            ->  insert [%tone L] ^b...^e;


    :: morphs with high tones:

       s un g u r un,        :: 'girl'
       k u r un,             :: 'canoe'
       j a a b i,            :: 'answer'
       j i,                  :: 'water'
       ...
            ->  insert [%tone H] ^b...^e;

    :: morphs with high-low patterns:

       m an g o r o,         :: 'mud'
       ...
            ->  insert [%tone H L] ^b...^e;

    :: morphs with low-high-low patterns:

       s a k e n e,          :: type of lizard
       ...
            ->  insert [%tone L H L] ^b...^e;

    ...
  %%
end morphs;
```

Figure 8. Sample Morph Dictionary.

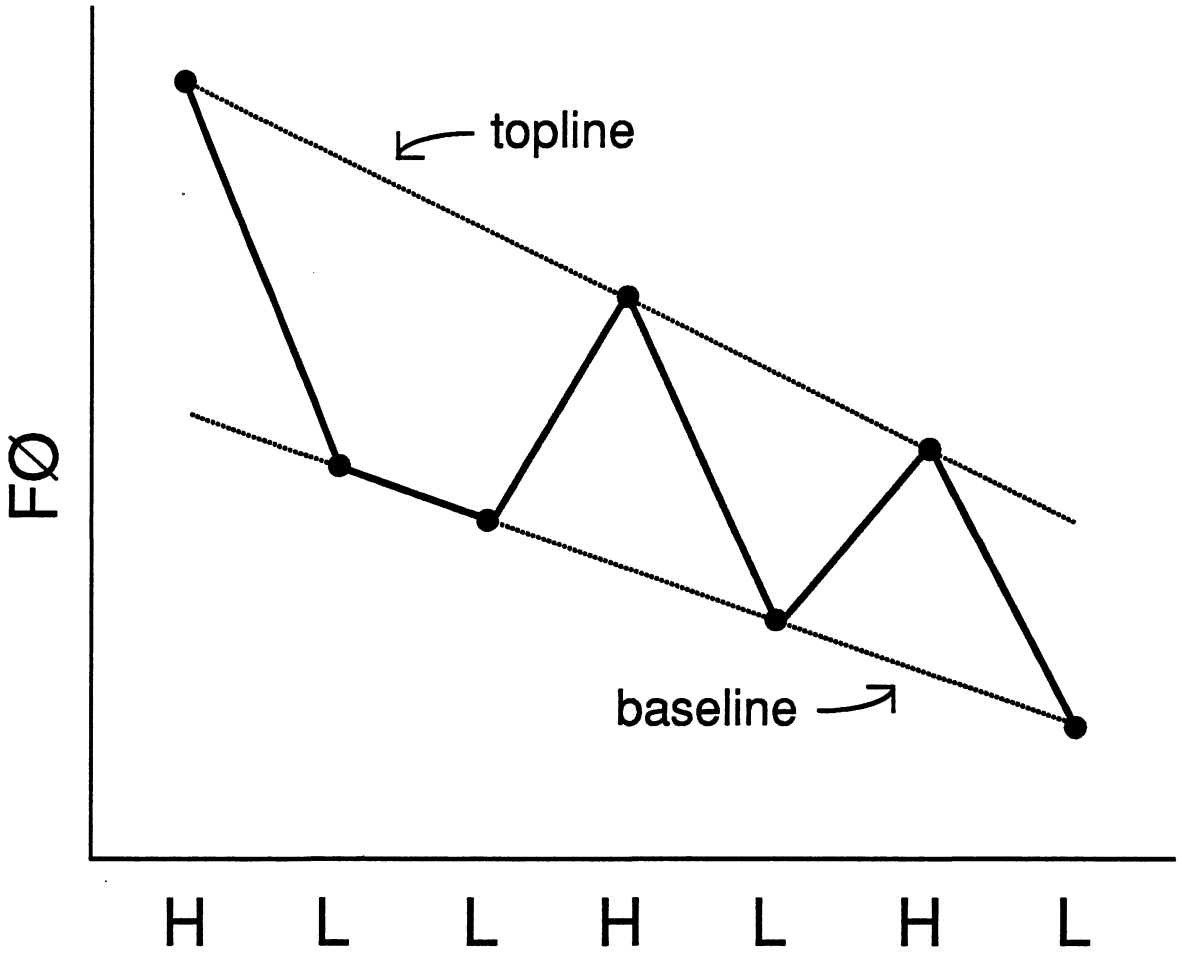


Figure 9. Baseline and Topline.

```
   H_start_freq  = 230;     :: Starting frequency of high tone line
   H_end_freq    = 150;     :: Ending frequency of high tone line
   L_start_freq  = 170;     :: Starting frequency of low tone line
   L_end_freq    = 130;     :: Ending frequency of low tone line

   sent_dur  = dur(^left...^right);  :: Sentence Duration

   H_slope   =  (H_end_freq - H_start_freq) / sent_dur;
                           :: Slope of high tone line

   L_slope   =  (L_end_freq - L_start_freq) / sent_dur;
                           :: Slope of low tone line

:: Insert an F0 value for each nucleus:

   loop forall [%nucleus _^bn nuc !^an];

    :: Compute the duration from the beginning of the sentence to
    :: the point halfway through the V (the point where we wish
    :: to compute and insert the F0 value):

       half_nuc_dur = .5 * dur(^bn...^an);
       elapsed_time = dur(^left...(^bn + half_nuc_dur));

    :: Compute the F0 value depending on whether the nucleus
    :: is low- or high-toned:

       if
        [%tone _\\^bn H //^an]
             -> f0_val = H_slope * elapsed_time + H_start_freq;
        else -> f0_val = L_slope * elapsed_time + L_start_freq;
       fi;

    :: Insert the computed F0 value halfway through the nucleus:

       insert [%F0 f0_val] at (^bn + half_nuc_dur);
   pool;
```

Figure 10. Delta Program for F_0 Target Assignment.

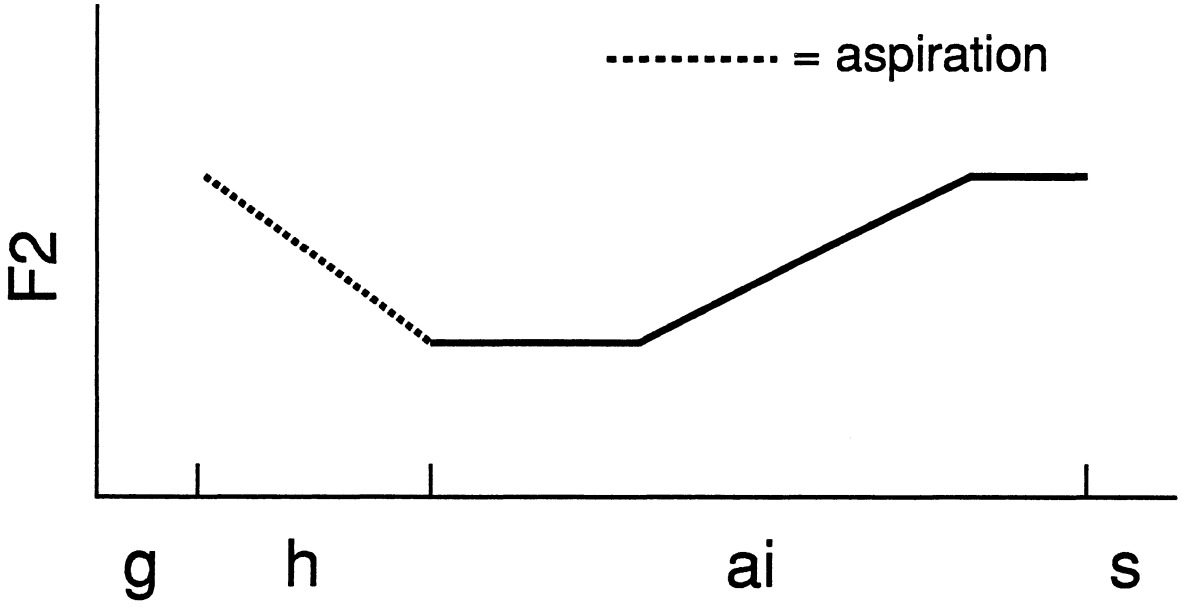


Figure 11. Second Formant Pattern for English [g h aɪ̯ s].

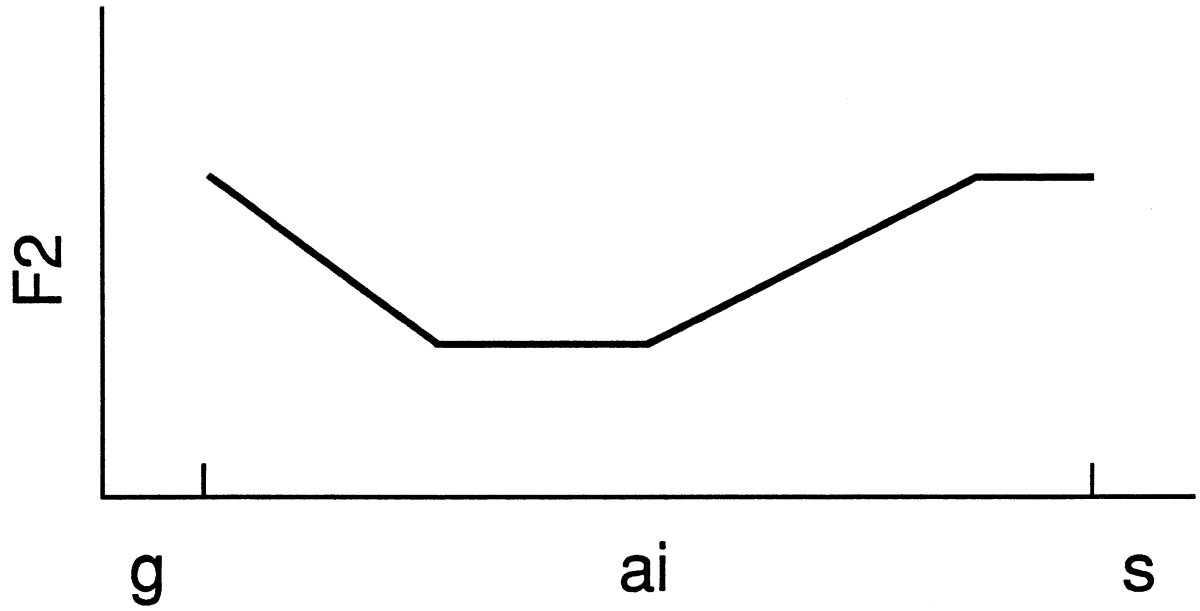


Figure 12. Second Formant Pattern for English [g aɪ̯ s].