

EthioMorph: Technical Implementation of a Rule-Based Ge'ez Morphological Engine

Esubalew Chekol
College of Technology and Built Environment
Addis Ababa University
NLP Course Project

Abstract—Ge'ez verb conjugation follows strict patterns. Given a root like ቀተለ, every form (perfective, imperfective, jussive, imperative) can be computed by rearranging vowels and attaching affixes. No dictionary lookup, no statistical model—just rules.

This paper documents EthioMorph, a Python system I built to do exactly that. It handles all eight verb types that traditional Ge'ez grammar recognizes: ቀተለ, ቀደሰ, ባረከ, ጠመረ, ሰሰየ, ክህለ, ማሕረከ, and ተንበለ. The core trick: Ethiopic Unicode assigns consecutive codepoints to vowel forms, so changing a vowel is just arithmetic. The rest is template matching and a few special rules for laryngeal consonants.

Index Terms—Ge'ez, morphological analysis, NLP, Unicode processing, Semitic morphology, rule-based systems

I. Introduction

I wanted to build a Ge'ez conjugator that works from first principles. Not a lookup table with thousands of pre-stored forms, and not a neural network trained on text. Just the rules, written in code.

Why is this even possible? Because Ge'ez is regular. The same root (ቀተለ, meaning “kill”) always conjugates the same way. There are eight verb types and maybe a dozen special cases for guttural consonants. Once you encode those rules, you can generate or analyze any verb.

A. What the System Does

Ge'ez verbs have two parts: a consonant root (usually 3 letters) and a vowel pattern. The root carries meaning, the pattern carries grammar.

Take ቀ-ተ-ለ (ቀተለ, “kill”). Same three consonants, different vowels:

q-t-l + [1-1-1] → ቀተለ (past: “he killed”)
q-t-l + [yi-1-6-6] → ይቅጥል (present: “he kills”)

EthioMorph does this in both directions:

Analysis: ይቅጥል → root: ቀተለ, tense: imperfective, subject: 3rd plural masculine

Generation: (ቀተለ, imperfective, 3PM) → ይቅጥል

II. Unicode Mathematics for Ethiopic

A. The Ethiopic Block Structure

The Ethiopic Unicode block (U+1200–U+137F) organizes characters in a systematic pattern. Each consonant has 7 forms representing vowel variations:

TABLE I
Unicode Pattern for Consonant ቀ (qāf)

Order	Unicode	Character	Vowel
1	U+1240	ቀ	ä (schwa)
2	U+1241	ቁ	u
3	U+1242	ቂ	i
4	U+1243	ቃ	a
5	U+1244	ቄ	e
6	U+1245	ቅ	(none)
7	U+1246	ቆ	o

B. Vowel Arithmetic

Since vowel forms are consecutive in Unicode, you can do math on them:

$$\text{order}(c) = ((c - \text{base_offset}) \bmod 8) + 1 \quad (1)$$

$$\text{base}(c) = c - (\text{order}(c) - 1) \quad (2)$$

To change a vowel:

$$\text{revowelize}(\text{base}, \text{order}) = \text{base} + (\text{order} - 1) \quad (3)$$

Implementation:

```
1 def get_vowel_order(char):  
2     """Extract vowel order (1-7) from  
3     character."""  
4     return ORDER_MAP.get(char, 1)  
5  
6 def devowelize(char):  
7     """Return base consonant (1st order  
8     form)."""  
9     return DEVOWELIZATION_MAP.get(char,  
10     char)  
11  
12 def get_char_by_order(base, order):  
13     """Apply vowel order to base consonant.  
14     """  
15     return REVOWELIZATION_MAP.get((base,  
16     order), base)
```

Example transformation:

devowelize(ቃ) = ቀ
get_vowel_order(ቃ) = 4
get_char_by_order(ቀ, 6) = ቅ

III. The C1-C2-C3 Radical System

A. Naming the Radicals

I use C1, C2, C3 to refer to the consonants:

- C1: First radical (consonant 1)
- C2: Second radical
- C3: Third radical
- C4: Fourth radical (quadriliterals only)

For $\Phi\tau\Lambda$: C1= Φ , C2= τ , C3= Λ

B. How Templates Work

Each conjugation form is just a vowel pattern plus optional prefix/suffix:

```

1 # Perfective 3rd singular masculine for
  Type A
2 template = {
3   "vowel_map": {"C1": 1, "C2": 1, "C3":
4     1},
5   "prefix": "",
6   "suffix": ""
7 }
8 # Result: C1(1st) + C2(1st) + C3(1st) = qä-
  tä-lä

1 # Imperfective 3rd singular masculine
2 template = {
3   "vowel_map": {"C1": 1, "C2": 6, "C3":
4     6},
5   "prefix": "yi",
6   "suffix": ""
7 }
8 # Result: yi + C1(1st) + C2(6th) + C3(6th)
  = yi-qä-t-l

```

C. Putting It Together

Algorithm 1 Generate a conjugated word

Require: root (string), tense, subject, verb_type
Ensure: conjugated word (string)

- 1: template \leftarrow TEMPLATES[verb_type][tense][subject]
- 2: vowel_map \leftarrow template["vowel_map"]
- 3: radicals \leftarrow extract_radicals(root)
- 4: result \leftarrow template["prefix"]
- 5: **for** each position in ["C1", "C2", "C3"] **do**
- 6: base \leftarrow devowelize(radicals[position])
- 7: order \leftarrow vowel_map[position]
- 8: result \leftarrow result + get_char_by_order(base, order)
- 9: **end for**
- 10: result \leftarrow result + template["suffix"]
- 11: result \leftarrow apply_suffix_fusion(result)
- 12: **return** result

IV. Suffix Fusion Rules

A. When Suffixes Collide

When you attach a vowel suffix to a consonant-only ending (6th order), you can't just concatenate them:

$$C_{6th} + V_{suffix} \rightarrow C_{order(V)} \quad (4)$$

Example: $\Phi\tau\Lambda + \text{ኡ}$ ("they") $\neq \Phi\tau\Lambda\text{ኡ}$

Instead: $\Phi\tau\Lambda + \text{ኡ} \rightarrow \Phi\tau\Lambda$

B. Fusion Algorithm

```

1 def apply_suffix_fusion(stem, suffix):
2     """Fuse 6th-order final + vowel suffix.
3     """
4     if not suffix:
5         return stem
6
7     last_char = stem[-1]
8     last_order = get_vowel_order(last_char)
9
10    # Check if stem ends in 6th order (
11    consonant)
12    if last_order != 6:
13        return stem + suffix
14
15    # Map suffix to target order
16    suffix_vowel_map = {
17        '\u12A1': 2, # u-vowel suffix
18        '\u12A0': 4, # a-vowel suffix
19        '\u12A5': 5, # e-vowel suffix
20    }
21
22    first_suffix_char = suffix[0]
23    if first_suffix_char in
24        suffix_vowel_map:
25        target_order = suffix_vowel_map[
26            first_suffix_char]
27        base = devowelize(last_char)
28        fused = get_char_by_order(base,
29            target_order)
30        return stem[:-1] + fused + suffix
31        [1:]
32
33    return stem + suffix

```

V. Verb Home Detection Algorithm

A. The Eight Canonical Verb Types

In Classical Ge'ez grammar, verbs are classified into "homes" ($\mathbf{\Lambda\tau}$) named after their canonical representative. EthioMorph implements all eight standard verb types:

TABLE II
The Eight Canonical Ge'ez Verb Types (Homes)

#	Type Head	Name	Description
1	$\Phi\tau\Lambda$	Type A	Strong triradical (C1 = 1st order)
2	$\Phi\mathfrak{L}\Lambda$	Type B	Geminate (C2 doubles in conjugation)
3	$\mathfrak{q}\mathfrak{L}\mathfrak{h}$	Type C	Long vowel (C1 = 4th order $\mathfrak{L}\mathfrak{h}$)
4	$\mathfrak{a}\mathfrak{a}\mathfrak{L}$	Type C-O	O-initial (C1 = 7th order $\mathfrak{a}\mathfrak{a}\mathfrak{L}$)
5	$\mathfrak{a}\mathfrak{a}\mathfrak{f}$	Weak-Final	Final radical is \mathfrak{f} (weak)
6	$\mathfrak{h}\mathfrak{u}\mathfrak{L}$	Laryngeal	Contains laryngeal (\mathfrak{u}) radical
7	$\mathfrak{a}\mathfrak{q}\mathfrak{h}\mathfrak{L}\mathfrak{h}$	Quadriliteral	Four consonant radicals
8	$\mathfrak{t}\mathfrak{h}\mathfrak{L}\mathfrak{h}$	T-Quad	Quadriliteral with \mathfrak{t} - prefix

B. How to Classify a Verb

Each verb type uses different conjugation templates, so you need to figure out the type first. The algorithm checks:

- How many radicals? (3 = triradical, 4 = quadriliteral)
- What vowel does C1 have? (4th order = Type C, 7th order = Type C-O)
- Any laryngeals (**ʕ**, **ħ**, **ʕ̣**, **ħ̣**, **ʕ̣̣**) or weak consonants (**ʕ̣̣̣**, **ʕ̣̣̣̣**)?
- Does C2 double? (That's Type B)

TABLE III
Algorithmic Classification Rules

Condition	Type	Example
$ \text{radicals} \geq 4$	Quadriliteral	ʕ̣̣̣̣ħħħħ, ʕ̣̣̣̣ʕ̣̣̣̣
C1 order = 7	Type C-O	ʕ̣̣̣̣ʕ̣̣̣̣
C1 order = 4	Type C	ʕ̣̣̣̣ħ
$C3 \in \{\text{weak}\}$	Weak-Final	ħ̣̣̣̣ʕ̣̣̣̣
$C2 \in \{\text{laryngeal}\}$	Laryngeal	ħ̣̣̣̣ħ
C2 geminated	Type B	ʕ̣̣̣̣ʕ̣̣̣̣
Default	Type A	ʕ̣̣̣̣ʕ̣̣̣̣

C. Detection Algorithm

D. Feature Detection

Some roots have special properties that affect conjugation:

```

1 LARYNGEALS = {'h', 'H', 'x', 'a', 'A'} #
  Gutturals
2 WEAK_CONSONANTS = {'w', 'y'}
3
4 def detect_features(root, radicals):
5     features = {}
6
7     # Check for laryngeal consonants
8     for rad in radicals:
9         base = devowelize(rad)
10        if base in LARYNGEAL_BASES:
11            features['has_laryngeal'] =
              True
12            break
13
14    # Check for hollow verb (weak C2)
15    if len(radicals) >= 2:
16        c2_base = devowelize(radicals[1])
17        if c2_base in WEAK_BASES:
18            features['is_hollow'] = True
19
20    # Check for weak initial
21    c1_base = devowelize(radicals[0])
22    if c1_base in WEAK_BASES:
23        features['weak_initial'] = True
24
25    return features

```

VI. Laryngeal Vowel Shift Rules

A. Gutturals Are Picky

Laryngeal consonants (**ʕ**, **ħ**, **ʕ̣**, **ħ̣**, **ʕ̣̣**) don't accept certain vowels. When you try to apply a 1st order vowel to a laryngeal, it shifts:

Algorithm 2 Figure out which verb type this is

Require: root (string)

Ensure: (type, features)

```

1: radicals ← extract_consonants(root)
2: features ← {}
3: // Check radical count first
4: if len(radicals) ≥ 4 then
5:     if root starts with ʕ̣̣̣̣ then
6:         return ("type_tanbala", features) {ʕ̣̣̣̣ʕ̣̣̣̣}
7:     else
8:         return ("type_mahraka", features) {ʕ̣̣̣̣ħħħħ}
9:     end if
10: end if
11: // Check C1 vowel order
12: c1_order ← get_vowel_order(root[0])
13: if c1_order == 7 then
14:     return ("type_c_o", features) {ʕ̣̣̣̣ʕ̣̣̣̣}
15: else if c1_order == 4 then
16:     return ("type_c", features) {ʕ̣̣̣̣ħ}
17: end if
18: // Check for weak final radical
19: if devowelize(root[2]) ∈ WEAK_CONSONANTS then
20:     features["weak_final"] ← True
21:     return ("type_sesaya", features) {ħ̣̣̣̣ʕ̣̣̣̣}
22: end if
23: // Check for laryngeal radicals
24: for each radical in radicals do
25:     if devowelize(radical) ∈ LARYNGEALS then
26:         features["has_laryngeal"] ← True
27:         return ("type_kahla", features) {ħ̣̣̣̣ħ}
28:     end if
29: end for
30: // Check for gemination (Type B)
31: if is_geminate_pattern(root) then
32:     return ("type_b", features) {ʕ̣̣̣̣ʕ̣̣̣̣}
33: end if
34: // Default: Type A
35: return ("type_a", features) {ʕ̣̣̣̣ʕ̣̣̣̣}

```

$$\text{Laryngeal} + 1\text{st order} \rightarrow \text{Laryngeal} + 4\text{th order} \quad (5)$$

B. Implementation

```

1 def apply_laryngeal_rules(char, position,
  features):
2     """Shift vowels near laryngeal
      consonants."""
3     if not features.get('has_laryngeal'):
4         return char
5
6     base = devowelize(char)
7     order = get_vowel_order(char)
8
9     # Laryngeals prefer 4th order in
      jussive
10    if is_laryngeal(base) and order == 1:

```

```

11         return get_char_by_order(base, 4)
12
13     return char

```

VII. Stemmer: Going Backwards

The Stemmer takes a conjugated word and figures out its root. It's the reverse of generation.

A. The Steps

- 1) Normalize spelling (Ge'ez has some redundant characters)
- 2) Strip prefixes (ደ, ተ, አ, etc.)
- 3) Strip suffixes (ኡ, ከ, etc.)
- 4) Get the consonant skeleton
- 5) Restore any weak consonants that disappeared
- 6) Figure out tense/person from what you stripped
- 7) Classify the verb type

B. Checking Prefixes

You have to check longer prefixes first, or you'll make wrong matches:

```

1 PREFIXES = [
2     # Stem IV (Causative-Passive) - check
      first
3     'yaste', 'taste', 'naste', 'laste',
4     # Stem III (Causative)
      'yas', 'tas', 'nas', 'las',
5     'ya', 'ta', 'na', 'a',
6     # Stem II (Passive)
      'yit', 'tit', 'te',
7     # Stem I (Basic)
      'yi', 'ti', 'ni', 'li'
8 ]
9
10 # Sorted by length descending
11 PREFIXES.sort(key=len, reverse=True)

```

C. Pattern Identification

```

1 def identify_verb_pattern(stem,
2     stripped_prefixes):
3     """Determine grammatical pattern from
4     affixes."""
5
6     # Check stem markers in priority order
7     if 'aste' in stripped_prefixes:
8         return {'stem': 4, 'name': 'Causative-Passive'}
9
10    if 'as' in stripped_prefixes or
11        has_causative_vowel(stem):
12        return {'stem': 3, 'name': 'Causative'}
13
14    if 'te' in stripped_prefixes or 'yit'
15        in stripped_prefixes:
16        return {'stem': 2, 'name': 'Passive'}
17
18    return {'stem': 1, 'name': 'Basic'}

```

VIII. Data Structures

A. Templates JSON

```

1 {
2     "type_a": {
3         "perfective": {
4             "3sm": {
5                 "vowel_map": {"C1": 1, "C2": 1, "C3": 1},
6                 "prefix": "",
7                 "suffix": ""
8             },
9             "3sf": {
10                "vowel_map": {"C1": 1, "C2": 1, "C3": 4},
11                "prefix": "",
12                "suffix": "t"
13            }
14            // ... 10 person forms
15        },
16        "imperfective": {
17            "3sm": {
18                "vowel_map": {"C1": 1, "C2": 6, "C3": 6},
19                "prefix": "yi",
20                "suffix": ""
21            }
22            // ...
23        }
24        // jussive, imperative, derived forms
25    }
26 }

```

B. Lookup Maps

I precompute three dictionaries so character lookups are instant:

```

1 # Character -> Base consonant
2 DEVOWELIZATION_MAP = {
3     '\u1240': '\u1240', # qe -> qe (
4         already base)
5     '\u1241': '\u1240', # qu -> qe
6     '\u1242': '\u1240', # qi -> qe
7     # ... all 300+ characters
8 }
9
10 # Character -> Vowel order (1-7)
11 ORDER_MAP = {
12     '\u1240': 1, # qe = 1st order
13     '\u1241': 2, # qu = 2nd order
14     # ...
15 }
16
17 # (Base, Order) -> Character
18 REVOWELIZATION_MAP = {
19     ('\u1240', 1): '\u1240',
20     ('\u1240', 2): '\u1241',
21     ('\u1240', 3): '\u1242',
22     # ...
23 }

```

IX. System Architecture

X. API Usage

A. Analysis Endpoint

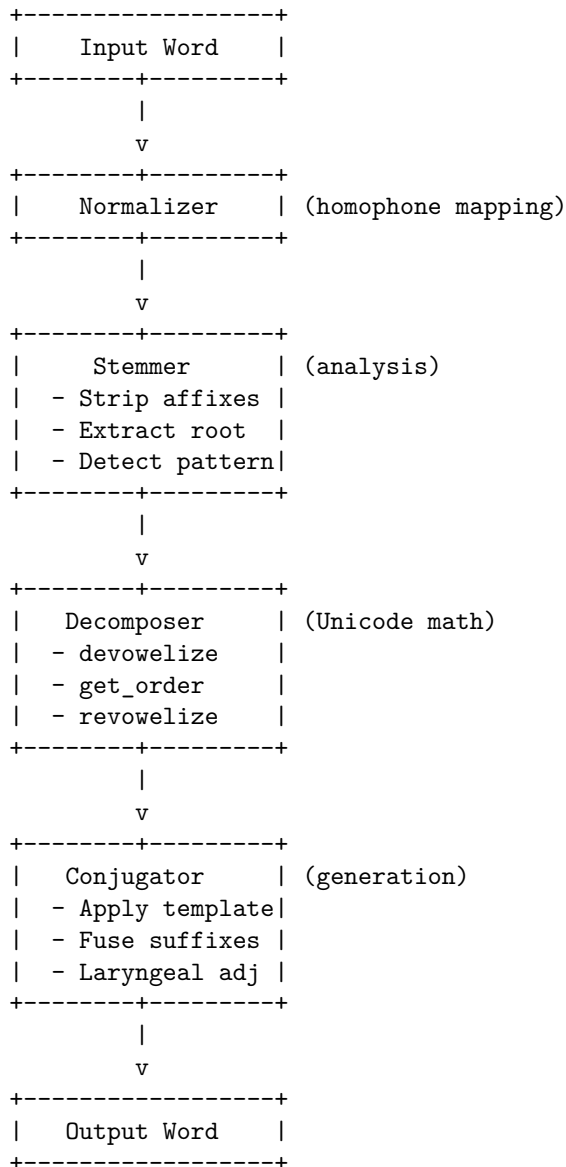


Fig. 1. How a word flows through the system

```

2 {
3   "root": "qatala"
4 }
5
6 Response:
7 {
8   "perfective": {
9     "3sm": "qatala",
10    "3sf": "qatalat",
11    ...
12  },
13  "imperfective": {...},
14  "jussive": {...}
15 }

```

XI. What I Learned

Building this taught me that Ge'ez morphology is more regular than I expected. The Unicode trick—that vowel forms are consecutive codepoints—turns what seems like a complex linguistic operation into basic arithmetic. Most of the code is just applying templates and handling edge cases for gutturals.

The system isn't complete. It handles the eight main verb types and basic stems, but there are derived forms and obscure patterns I haven't implemented. Still, it works well enough to conjugate most verbs you'd encounter in classical texts.

Code: github.com/esubalew/EthioMorph

```

1 POST /api/xray
2 {
3   "word": "yiqatlu"
4 }
5
6 Response:
7 {
8   "root": "qatala",
9   "pattern": "imperfective",
10  "stem": 1,
11  "person": "3pm",
12  "verb_type": "type_a"
13 }

```

B. Generation Endpoint

```

1 POST /api/morph

```