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 that does exactly that. It handles all eight verb types that traditional Ge’ez grammar recognizes: ቀተለ, ቀደሰ, ባረከ, ጦመረ, ሴሰየ, 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

EthioMorph is 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 **q-t-l** (**ቀተለ**, “kill”). Same three consonants, different vowels:

$$\text{q-t-l} + [\text{1-1-1}] \rightarrow \textbf{ቀተለ} \text{ (past: “he killed”)}$$
$$\text{q-t-l} + [\text{yi-1-6-6}] \rightarrow \textbf{ይቅትል} \text{ (present: “he kills”)}$$

EthioMorph does this in both directions:

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

Generation: (**ቀተለ**, imperfective, 3PM) → **ይቀትሉ**

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 (qäf)

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:

$$\text{order}(c) = ((c - \text{base\_offset}) \mod 8) + 1 \tag{1}$$

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

To change a vowel:

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

Implementation:

```
def get_vowel_order(char):
    """Extract vowel order (1-7) from character."""
    return ORDER_MAP.get(char, 1)

def devowelize(char):
    """Return base consonant (1st order form)."""
    return DEVOWELIZATION_MAP.get(char, char)

def get_char_by_order(base, order):
    """Apply vowel order to base consonant."""
    return REVOWELIZATION_MAP.get((base, order), base)
```

Example transformation:

$$\text{devowelize}(\text{ቃ}) = \text{ቀ}$$
$$\text{get\_vowel\_order}(\text{ቃ}) = 4$$
$$\text{get\_char\_by\_order}(\text{ቀ}, 6) = \text{ቅ}$$

III. The C1-C2-C3 Radical System

A. Naming the Radicals

C1, C2, C3 refer to the consonants:

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

For **ቀተለ**: C1=**ቀ**, C2=**ተ**, C3=**ለ**

B. How Templates Work

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

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

```
# Imperfective 3rd singular masculine
template = {
    "vowel_map": {"C1": 1, "C2": 6, "C3": 6},
    "prefix": "yi",
    "suffix": ""
}
# 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 ← TEMPLATES[verb_type][tense][subject]
2: vowel_map ← template["vowel_map"]
3: radicals ← extract_radicals(root)
4: result ← template["prefix"]
5: **for** each position in ["C1", "C2", "C3"] **do**
6: base ← devowelize(radicals[position])
7: order ← vowel_map[position]
8: result ← result + get_char_by_order(base, order)
9: **end for**
10: result ← result + template["suffix"]
11: result ← 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:

$$\mathrm{C}_{\mathrm{6th}} + \mathrm{V}_{\mathrm{suffix}} \rightarrow \mathrm{C}_{\mathrm{order}(V)} \tag{4}$$

Example: **ቅትል** + **ኡ** ("they") ≠ **ቅትልኡ**
Instead: **ቅትል** + **ኡ** → **ቅትሉ**

B. Fusion Algorithm

```
def apply_suffix_fusion(stem, suffix):
    """Fuse␣6th-order␣final␣+␣vowel␣suffix."""
    if not suffix:
        return stem

    last_char = stem[-1]
    last_order = get_vowel_order(last_char)

    # Check if stem ends in 6th order (consonant)
    if last_order != 6:
        return stem + suffix

    # Map suffix to target order
    suffix_vowel_map = {
        '\u12A1': 2,  # u-vowel suffix
        '\u12A0': 4,  # a-vowel suffix
        '\u12A5': 5,  # e-vowel suffix
    }

    first_suffix_char = suffix[0]
    if first_suffix_char in suffix_vowel_map:
        target_order = suffix_vowel_map[first_suffix_char]
        base = devowelize(last_char)
        fused = get_char_by_order(base, target_order)
        return stem[:-1] + fused + suffix[1:]

    return stem + suffix
```

V. Verb Home Detection Algorithm

A. The Eight Canonical Verb Types

In Classical Ge'ez grammar, verbs are classified into "homes" (**ቤት**) 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	**ቀተለ**	Type A	Strong triradical (C1 = 1st order)
2	**ቀደሰ**	Type B	Geminate (C2 doubles in conjugation)
3	**ባረከ**	Type C	Long vowel (C1 = 4th order **ራብዕ**)
4	**ጦመረ**	Type C-O	O-initial (C1 = 7th order **ሳብዕ**)
5	**ሴሰየ**	Weak-Final	Final radical is **የ** (weak)
6	**ክህለ**	Laryngeal	Contains laryngeal (**ሀ**) radical
7	**ማኅረከ**	Quadriliteral	Four consonant radicals
8	**ተንበለ**	T-Quad	Quadriliteral with **ተ**- 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 (**ሀ, ሐ, ኀ, አ, ዐ**) or weak consonants (**ወ, የ**)?
- Does C2 double? (That's Type B)

TABLE III
Algorithmic Classification Rules

Condition	Type	Example
$\lvert\text{radicals}\rvert \geq 4$	Quadriliteral	**ማሕረከ, ተንበለ**
C1 order = 7	Type C-O	**ጦመረ**
C1 order = 4	Type C	**ባረከ**
C3 $\in$ {weak}	Weak-Final	**ሴሰየ**
C2 $\in$ {laryngeal}	Laryngeal	**ክህለ**
C2 geminated	Type B	**ቀደሰ**
Default	Type A	**ቀተለ**

C. Detection Algorithm

D. Feature Detection

Some roots have special properties that affect conjugation:

```
LARYNGEALS = {'h', 'H', 'x', 'a', 'A'}  # Gutturals
WEAK_CONSONANTS = {'w', 'y'}

def detect_features(root, radicals):
    features = {}

    # Check for laryngeal consonants
    for rad in radicals:
        base = devowelize(rad)
        if base in LARYNGEAL_BASES:
            features['has_laryngeal'] = True
            break

    # Check for hollow verb (weak C2)
    if len(radicals) >= 2:
        c2_base = devowelize(radicals[1])
        if c2_base in WEAK_BASES:
            features['is_hollow'] = True

    # Check for weak initial
    c1_base = devowelize(radicals[0])
    if c1_base in WEAK_BASES:
        features['weak_initial'] = True

    return features
```

VI. Laryngeal Vowel Shift Rules

A. Gutturals Are Picky

Laryngeal consonants (**ሀ, ሐ, ኀ, አ, ዐ**) 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)

```
radicals ← extract_consonants(root)
features ← {}
// Check radical count first
if len(radicals) ≥ 4 then
    if root starts with ተ then
        return ("type_tanbala", features) {ተንበለ}
    else
        return ("type_mahraka", features) {ማሕረከ}
    end if
end if
// Check C1 vowel order
c1_order ← get_vowel_order(root[0])
if c1_order == 7 then
    return ("type_c_o", features) {ጦመረ}
else if c1_order == 4 then
    return ("type_c", features) {ባረከ}
end if
// Check for weak final radical
if devowelize(root[2]) ∈ WEAK_CONSONANTS then
    features["weak_final"] ← True
    return ("type_sesaya", features) {ሴሰየ}
end if
// Check for laryngeal radicals
for each radical in radicals do
    if devowelize(radical) ∈ LARYNGEALS then
        features["has_laryngeal"] ← True
        return ("type_kahla", features) {ክህለ}
    end if
end for
// Check for gemination (Type B)
if is_geminate_pattern(root) then
    return ("type_b", features) {ቀደሰ}
end if
// Default: Type A
return ("type_a", features) {ቀተለ}
```

$$\text{Laryngeal} + \text{1st order} \rightarrow \text{Laryngeal} + \text{4th order} \tag{5}$$

B. Implementation

```
def apply_laryngeal_rules(char, position, features):
    """Shift vowels near laryngeal consonants."""
    if not features.get('has_laryngeal'):
        return char

    base = devowelize(char)
    order = get_vowel_order(char)

    # Laryngeals prefer 4th order in jussive
    if is_laryngeal(base) and order == 1:
```

```
        return get_char_by_order(base, 4)

    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 (ይ, ት, እ, etc.)
3) Strip suffixes (ኡ, ክ, 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:

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

C. Pattern Identification

```
def identify_verb_pattern(stem, stripped_prefixes):
    """Determine grammatical pattern from affixes."""

    # Check stem markers in priority order
    if 'aste' in stripped_prefixes:
        return {'stem': 4, 'name': 'Causative-Passive'}

    if 'as' in stripped_prefixes or has_causative_vowel(stem):
        return {'stem': 3, 'name': 'Causative'}

    if 'te' in stripped_prefixes or 'yit' in stripped_prefixes:
        return {'stem': 2, 'name': 'Passive'}

    return {'stem': 1, 'name': 'Basic'}
```

VIII. Data Structures

A. Templates JSON

```
{
  "type_a": {
    "perfective": {
      "3sm": {
        "vowel_map": {"C1": 1, "C2": 1, "C3": 1},
        "prefix": "",
        "suffix": ""
      },
      "3sf": {
        "vowel_map": {"C1": 1, "C2": 1, "C3": 4},
        "prefix": "",
        "suffix": "t"
      }
      // ... 10 person forms
    },
    "imperfective": {
      "3sm": {
        "vowel_map": {"C1": 1, "C2": 6, "C3": 6},
        "prefix": "yi",
        "suffix": ""
      }
      // ...
    }
    // jussive, imperative, derived forms
  }
}
```

B. Lookup Maps

Three precomputed dictionaries make character lookups instant:

```
# Character -> Base consonant
DEVOWELIZATION_MAP = {
    '\u1240': '\u1240',  # qe -> qe (already base)
    '\u1241': '\u1240',  # qu -> qe
    '\u1242': '\u1240',  # qi -> qe
    # ... all 300+ characters
}

# Character -> Vowel order (1-7)
ORDER_MAP = {
    '\u1240': 1,  # qe = 1st order
    '\u1241': 2,  # qu = 2nd order
    # ...
}

# (Base, Order) -> Character
REVOWELIZATION_MAP = {
    ('\u1240', 1): '\u1240',
    ('\u1240', 2): '\u1241',
    ('\u1240', 3): '\u1242',
    # ...
}
```

IX. System Architecture

X. API Usage

A. Analysis Endpoint

```
+-------------------+
|    Input Word     |
+---------+---------+
          |
          v
+---------+---------+
|    Normalizer     | (homophone mapping)
+---------+---------+
          |
          v
+---------+---------+
|     Stemmer       | (analysis)
|  - Strip affixes  |
|  - Extract root   |
|  - Detect pattern|
+---------+---------+
          |
          v
+---------+---------+
|   Decomposer      | (Unicode math)
|  - devowelize     |
|  - get_order      |
|  - revowelize     |
+---------+---------+
          |
          v
+---------+---------+
|   Conjugator      | (generation)
|  - Apply template|
|  - Fuse suffixes |
|  - Laryngeal adj |
+---------+---------+
          |
          v
+-------------------+
|   Output Word     |
+-------------------+
```

Fig. 1. How a word flows through the system

```
POST /api/xray
{
  "word": "yiqatlu"
}

Response:
{
  "root": "qatala",
  "pattern": "imperfective",
  "stem": 1,
  "person": "3pm",
  "verb_type": "type_a"
}
```

B. Generation Endpoint

```
POST /api/morph
{
  "root": "qatala"
}

Response:
{
  "perfective": {
    "3sm": "qatala",
    "3sf": "qatalat",
    ...
  },
  "imperfective": {...},
  "jussive": {...}
}
```

XI. Observations

Ge'ez morphology turns out to be more regular than expected. The Unicode trick (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 not yet implemented. Still, it works well enough to conjugate most verbs you'd encounter in classical texts.

Code: `github.com/Esubaalew/EthioMorph`