CHAPTER 6

Morphological Analysis and Generation

The previous chapter included several examples in which rewrite rules were used to generate variants—including plurals and locative case forms—of various words. Building on these techniques, this chapter provides a systematic finite-state treatment of **morphology**, the study of word formation and word-internal structure.

Much of the early work in speech and language processing treated words—however defined—as indivisible units of linguistic meaning, and as such ignored the relationship between English words like *lock*, *locked*, *locksmiths*, *padlock*, and *unlockable*. Modeling these relations soon became necessary, however. Early work in **information retrieval** (effectively, early versions of search engines) found it helpful to reduce sparsity by conflating related English words using **stemmers**, cascades of handwritten, language-specific suffix-stripping rules (e.g., Porter 1980). The "stems" produced need not even be real words—for instance, applying the Porter stemmer to some of the previous text yields non-words like *cascad*, *languag*, and *produc*—so long as words sharing the same stem fall into semantically coherent equivalence classes. Intuitively, someone who searches the internet or a database for *fishing* may also be interested in documents that only mention *fish*. Replacing words with their "stems" allows retrieval systems to improve generalization and reduce the computing and memory requirements of indexing and retrieval. At roughly the same time, early word processing and typesetting systems were forced to model morphology to compress large lists of words—for spell-checking (McIlroy 1982) and hyphenation (Liang 1983), for example—to fit within limited random access memory of the era's microcomputers.[1] One of the more systematic early computational treatments of English morphology was undertaken by the developers of MITalk, a **text-to-speech synthesis** engine. This system is described by Allen et al. (1987) and Klatt (1987), although work began almost two decades earlier.[2] MITalk includes a module called DECOMP which decomposes complex words into their constituent parts for the purposes of predicting their pronunciations. DECOMP consists of a list of stems and **affixes** (e.g., prefixes and suffixes), and rules governing the spelling of complex words. For example, consider the word *scarcity*, which is composed of *scarce* plus *-ity*, a suf-

[1] As an apocryphal story has it, Bill Gates told the audience of a 1980s trade show that 640 kilobytes of memory "ought to be enough for anybody". For comparison, the roughly 235,000 headwords of the second edition of *Webster's New International Dictionary* (Neilson and Knott 1934) constitute 2.2 MB of ASCII text.

[2] DECTalk, a commercial variant of MITalk put out by the Digital Equipment Corporation, is best known as the adopted voice of late physicist and author Stephen Hawking.

fix forming abstract nouns from adjectives. DECOMP generates this word—and thus recovers its decomposition—by concatenating *scarce* and *-ity* and applying a spelling rule that deletes a word-final *e* before certain suffixes; such rules are available only at the right edge of a stem. At the same time, DECOMP considers, but rejects, an alternative analysis *scar-city* (which, for instance, might refer to an urban area where scars are a common sight). This decomposition would yield an incorrect pronunciation, and various hand-tuned heuristics are used to avoid such incorrect decompositions. The system is able to generate over 100,000 words from its 12,000 lexical entries (Klatt 1987:773).

To the modern reader, the data and memory limitations motivating early work on morphological analysis may seem as distant as the 1980s themselves, but such problems still resonate today. Modern laptops may have several orders of magnitude more memory than the microcomputers used to run MITalk, but text processing systems are also expected to run on mobile devices, including affordable cellular phones, with limited computational resources.

6.1 APPLICATIONS

Many systems that process or generate speech or text may need to be sensitive to morphology. For example, the following chapter (section 7.6) illustrates the generation of weather reports. For English, such a system might use a template like `It's $TEMP degrees and $CONDITIONS in $LOCATION`. While this would correctly generate expressions like `It's twelve degrees and cloudy in Montreal`, it also would produce the ungrammatical *`one degrees` on certain cold days. **Internationalization** of text generation or processing systems originally built for English—or Mandarin—may require a great deal of morphological sophistication, simply because these languages have an uncommonly impoverished morphology. Whereas in English, a noun inflects for **number** (e.g., *one city*, *two cities*), Russian nouns are also inflected for **case**, which indicates the noun's grammatical function in the clause. As Russian has six cases and two numbers, a noun's **paradigm**—its list of inflected forms—may have as many as twelve variants. In practice, although, many forms may appear in multiple **slots** or **cells** in the paradigm, a phenomenon known as **syncretism**. The paradigm of a Russian noun is shown in Table 6.1 below; note the syncretism between nominative and accusative forms is characteristic of inanimate nouns.[3] Russian poses a somewhat greater challenge for the weather generation example above, for one says *odín grádus* 'one degree', *trí grádusa* 'three degrees', and *vósem' grádusov* 'eight degrees', using the nominative singular, genitive singular, and genitive plural forms, respectively.

The above example is an instance of **morphological generation**, and models which perform it are known as **generators** or **inflectors**. In this scenario, one wishes to produce a certain form of a stem, for example, the instrumental plural form of the Russian word *žurnál* 'magazine, journal', having already determined the appropriate form to use in this context. De-

[3] Russian examples have been transliterated from their Cyrillic spellings. Primary stress, marked here with an acute accent, is not part of the standard Cyrillic orthography of Russian, but is indicated here.

Table 6.1: Paradigm for the Russian masculine noun *grádus* 'degree'.

	Singular	Plural
Nominative	grádus	grádusy
Genitive	grádusa	grádusov
Dative	grádusy	grádusam
Accusative	grádus	grádusy
Instrumental	grádusom	grádusami
Prepositional	gráduse	grádusax

termining the proper morphological form to use in a given context is beyond the scope of this chapter. The inverse procedure, **morphological analysis**, recovers the **citation form** or **lemma**—roughly, the form one would expect to find this word listed under in a dictionary—and the **morphosyntactic features** of an inflected form. For instance, *žurnálami* is analyzed as the instrumental plural of *žurnál* 'magazine, journal', and might be represented by the string `žurnál+ami[num=pl][case=ins]` where the boundary symbol '+' separates stems and affixes and the morphosyntactic features are written in square brackets. In some cases, one is not as so interested in morphosyntactic features of the inflected word but simply wishes to recover its lemma, a process referred to as **lemmatization**. Conversely, some applications of morphosyntactic features do not require a decomposition into stem and affixes, a scenario referred to as **morphological tagging**. When applied to documents, morphological analysis may require one to first segment the text stream into sentences and/or words. **Sentence splitting** and **tokenization** also lie beyond the scope of this book, and may be non-trivial for certain scripts, particularly certain scripts of East Asia which do not consistently delimit word boundaries with whitespace or punctuation.

Lemmas and morphosyntactic features are commonly used to provide features to **part-of-speech tagging** (e.g., Denis and Sagot 2009, Hajič 2000, Halácsy et al. 2006) or **syntactic parsing** (e.g., Dehouck and Denis 2018, Dubey 2005, Fraser et al. 2013) systems, particularly in richly inflected languages. Lemmas are also used as a sophisticated alternative to stemming in information retrieval applications.

Computational morphological analysis and generation, including most of the examples in this chapter, often makes use of orthographic inputs and outputs rather than phonemic representations. There are several reasons for this. First, most applications of of morphological analysis or generation naturally take in or and/or produce orthographic forms, so additional effort is required to phonemic representations internally. Second, orthographic representations have little effect in languages like Finnish or Spanish, which have shallow, highly consistent orthographies that are quite close to phonemic or phonetic transcriptions. In contrast, languages like En-

glish and Korean have what Rogers (2005), inter alia, calls **deep orthographies**, meaning that spellings are more abstract. In English spelling, one rarely indicates morphologically conditioned changes in vowel quality, and as a result, related words like *sane* and *sanity* are spelled quite similarity despite the fact that they have different stem vowels (Chomsky and Halle 1968:44f.). One is therefore free to ignore stress shift and vowel reduction processes in English when building an orthographically based morphological analyzer.

6.2 WORD FORMATION

There are many different theoretical frameworks used by linguists studying morphology. One major distinction, is between the family of **item-and-arrangement** theories, which analyze a word like *žurnálami* as the concatenation of two meaningful **morphemes**—e.g., the stem *žurnál* and the instrumental plural suffix **-ami**—and **item-and-process** theories which views affixation as just one type of transformation applied to a base (Hockett 1954). However, Karttunen (2003) and Roark and Sproat (2007: ch. 3) argue that this distinction—as well as the additional distinction between **lexical** and **realizational** theories popularized by Stump (2001)—are essentially computationally equivalent because all the relevant processes under either family of theories are largely equivalent to the rational relations and can be modeled by cascades of finite-state transducers. Furthermore, FSTs designed for generation can be modified for analysis, lemmatization, or tagging depending on one's needs.

Koskenniemi's broad-coverage morphology of Finnish, reviewed in detail by Roark and Sproat (2007: ch. 4), was one of the first attempts to use finite-state automata for morphological analysis and generation. The model Koskenniemi proposed became something of a standard for fieldwork and language documentation of morphologically rich languages (Antworth 1990). The subsequent discovery of algorithms for compiling rewrite rules (chapter 5) in the 1980's, greatly simplifies the process of constructing an analyzer, inflector, lemmatizer, or tagger. Using the tools already discussed, there are numerous ways one might build a finite-state analyzer. Indeed, this "many ways to do it" freedom is one of the more pleasant features of finite-state computing. Perhaps the simplest approach for dealing with Russian nouns, for example, would be to simply construct a transducer from wordforms to their associated features; it would be straightforward to convert this analyzer to a inflector, lemmatizer, or tagger. This is illustrated in the snippet below. If this resulting transducer is called ν, then given μ, an FST mapping from features to human-readable strings, one could extract the analyses of a string s from the lattice $\pi_o(s \circ \nu \circ \mu)$.

```
nouns = pynini.string_map(
    [
        ("žurnál", "žurnál+[num=sg][case=nom]"),
        ("žurnála", "žurnál+a[num=sg][case=gen]"),
        ("žurnálu", "žurnál+u[num=sg][case=dat]"),
        ("žurnál", "žurnál+[num=sg][case=acc]"),
```

```
        ("žurnálom", "žurnál+om[num=sg][case=ins]"),
        ("žurnále", "žurnál+e[num=sg][case=prp]"),
        ("žurnálá", "žurnál+y[num=pl][case=nom]"),
        ("žurnálóv", "žurnál+ov[num=pl][case=gen]"),
        ("žurnálám", "žurnál+am[num=pl][case=dat]"),
        ("žurnálá", "žurnál+y[num=pl][case=acc]"),
        ("žurnáláx", "žurnál+ax[num=pl][case=prp]"),
        ("žurnálámi", "žurnál+ami[num=pl][case=ins]"),
    ]
)
```

Of course, to cover a sizeable chunk of Russian noun morphology would require adding whatever nouns one wanted to cover. The disadvantages of this approach should be clear: the list of forms would be need to be extremely large to obtain broad coverage. While this approach is workable, it is not ideal. One would prefer a method that allows one to

1. share information across multiple stems in the same paradigm,
2. inherit information from related paradigms,
3. conveniently represent morphosyntactic features and feature bundles, and
4. construct analyzers, generators, lemmatizers, and taggers.

The remainder of this chapter introduces the `features` and `paradigms` modules, part of Pynini's extended library (Appendix C), which provide precisely this functionality.

6.3 FEATURES

While the term has a much broader sense in morphological theory, here a **feature** refers to morpho-syntactic property that defines the slots within a given paradigm. For instance, Russian noun paradigms are defined by case and number. Classes in the `features` module can be used to define features and slots. The first argument to the `Feature` constructor is the name of the feature, and the remaining arguments define the names of valid values for that feature. Case and number features are defined in the following snippet.

```
case = features.Feature(
    "case", "nom", "gen", "dat", "acc", "ins", "prp"
)
num = features.Feature("num", "sg", "pl")
```

A `Category` is a combination of features, expressed as an acceptor which accepts any sequence of the feature-value pairs it is constructed from. Thus, the noun `Category` defined in the following snippet will admit feature combinations like {`[case=nom]`, `[num=sg]`}, or {`[case=ins]`, `[num=pl]`}.

```
noun = features.Category(case, num)
```

Finally, a `FeatureVector` represents a valid combination of a `Category` and a sequence of feature specifications. For instance, the following combinations are valid.

```
nomsg = features.FeatureVector(noun, "case=nom", "num=sg")
genpl = features.FeatureVector(noun, "case=gen", "num=pl")
inspl = features.FeatureVector(noun, "num=pl", "case=ins")
```

6.4 PARADIGMS

Having defined the slots in a paradigm, it now is necessary to combine them into a paradigm. This is accomplished using the `paradigms` module. One first selects a boundary symbol to separate wordforms into stems and affixes, conventionally '+'. The function `make_byte_star_except_boundary` is used to construct a definition of a stem, here $(\Sigma - \{+\})^*$ where Σ is the set of bytes.

The `paradigms` module follows Roark and Sproat (2007: ch. 2)—and the spirit of the item-and-process model— in that affixes are introduced via composition rather than concatenation. Thus, a suffix s permitted to attach to any stem corresponds to the rational relation $\zeta = S(\emptyset \times \{s\})$, where S is the set of stems. Then, then $\pi_o[x \circ \zeta]$ where $x \in S$, corresponds to the string xs, and a similar logic applies for prefixes. This may seem like a roundabout way of defining affixation, but it—unlike concatenation—is general enough to allow for additional restrictions on stem shape, to permit the affix to trigger a change to the stem, or even to permit the affix to insert itself into the stem, all of which are seen in the following examples.

6.5 EXAMPLES

The remainder of this chapter sketches morphological analyzers for four languages. We focus here on cases that involve relatively complex paradigmatic relationships, or non-concatenative systems where the affixation is sensitive to the phonological shape of the base. Of course purely concatenative and potentially unbounded morphology of the kind as discussed by Hankamer (1989) for Turkish (see section 5.1) is also important, but then again it is more obvious how one might implement such cases. We leave it as an exercise for the reader to implement a model that can handle examples like *marginalizationalizationalizátion* discussed in section 5.1.

6.5.1 RUSSIAN NOUNS

In the `paradigms` module, a paradigm is primarily defined by slots, and each slot is defined by a pair consisting of an affixation transducer and a `FeatureVector`. For Russian nouns, since the relevant affixes are suffixes that impose minimal constraints on their stems, these affixation transducers are defined using the module's `suffix` function, which constructs a simple suffixation relation ζ given stem shape and suffix acceptors; a `prefix` function is also provided.

A `Paradigm` is constructed from

1. the `Category`,
2. a list of slots, and
3. a `FeatureVector` defining the lemma,
4. a list of stems (strings or acceptors),

as well as several optional fields illustrated below. The `Paradigm` can then lazily construct analyzer, tagger, lemmatizer, and inflector transducers. The following snippets implement the so-called "hard stem masculine accent A", which includes Russian nouns like *grádus* and *žurnál*, using the nominal features and noun category defined above. Note that the lemma is assumed to be the nominative singular form. This is shown in the following snippets.

- Defines the stem shape:

```
stem = paradigms.make_byte_star_except_boundary()
```

- Defines the slots:

```
slots = [
    (stem, nomsg),
    (paradigms.suffix("+a", stem),
     features.FeatureVector(noun, "case=gen", "num=sg")),
    (paradigms.suffix("+u", stem),
     features.FeatureVector(noun, "case=dat", "num=sg")),
    (stem,
     features.FeatureVector(noun, "case=acc", "num=sg")),
    (paradigms.suffix("+om", stem),
     features.FeatureVector(noun, "case=ins", "num=sg")),
    (paradigms.suffix("+e", stem),
     features.FeatureVector(noun, "case=prp", "num=sg")),
    (paradigms.suffix("+y", stem),
     features.FeatureVector(noun, "case=nom", "num=pl")),
    (paradigms.suffix("+ov", stem),
     features.FeatureVector(noun, "case=gen", "num=pl")),
    (paradigms.suffix("+am", stem),
     features.FeatureVector(noun, "case=dat", "num=pl")),
    (paradigms.suffix("+y", stem),
     features.FeatureVector(noun, "case=acc", "num=pl")),
    (paradigms.suffix("+ami", stem),
     features.FeatureVector(noun, "case=ins", "num=pl")),
```

```
        (paradigms.suffix("+ax", stem),
         features.FeatureVector(noun, "case=prp", "num=pl")),
    ]
```

- Constructs the paradigm:

```
masc_accent_a = paradigms.Paradigm(
    category=noun,
    name="hard stem masculine accent A",
    slots=slots,
    lemma_feature_vector=nomsg,
    stems=["grádus", "žurnál"]
)
```

The "hard stem masculine accent B" paradigm can be similarly defined. The only difference between this and the accent A paradigm is that word stress shifts to the suffix in all cases except the nominative and accusative singular, a process which be traced back thousands of years to the accentual system of Proto-Indo-European (Halle 1997). Thus, for instance, the dative singular of *górb* 'hump, hunch' is *gorbú* and the nominative plural of *stól* 'table' is *stolý*. To generate this pattern, one needs only to place an acute accent on the appropriate vowel in the suffix, and then to provide a rule deaccenting the stem. Note also that Σ^* for this rule must include the output features, obtained from the output projection of the noun `Category`'s feature mapper automaton. The optional `parent_paradigm` argument to `Paradigm` allows this to inherit any slots not redefined from another paradigm, here, nominative and accusative singular forms from the accent A paradigm. Finally, the optional `rules` argument specifies an optional list of rules—here, just the deaccentuation rule—to be applied when constructing wordforms. The following snippets demonstrate the construction of the accent B paradigm. Note that the `noun` object provides Σ^* for the rewrite rule.

- Defines the deaccentuation rule:

```
deaccentuation_map = pynini.string_map(
    [
        ("á", "a"), ("é", "e"), ("í", "i"),
        ("ó", "o"), ("ú", "u"), ("ý", "y"),
    ]
)
acc_v = pynini.project(deaccentuation_map, "input")
deaccentuation = pynini.cdrewrite(
    deaccentuation_map, "", noun.sigma_star + acc_v, noun.sigma_star
).optimize()
```

- Defines the slots:

```
slots = [
    (paradigms.suffix("+á", stem), nomsg),
    (paradigms.suffix("+ú", stem),
     features.FeatureVector(noun, "case=dat", "num=sg")),
    (paradigms.suffix("+óm", stem),
     features.FeatureVector(noun, "case=ins", "num=sg")),
    (paradigms.suffix("+é", stem),
     features.FeatureVector(noun, "case=prp", "num=sg")),
    (paradigms.suffix("+ý", stem),
     features.FeatureVector(noun, "case=nom", "num=pl")),
    (paradigms.suffix("+óv", stem),
     features.FeatureVector(noun, "case=gen", "num=pl")),
    (paradigms.suffix("+ám", stem),
     features.FeatureVector(noun, "case=dat", "num=pl")),
    (paradigms.suffix("+ý", stem),
     features.FeatureVector(noun, "case=acc", "num=pl")),
    (paradigms.suffix("+ámi", stem),
     features.FeatureVector(noun, "case=ins", "num=pl")),
    (paradigms.suffix("+áx", stem),
     features.FeatureVector(noun, "case=prp", "num=pl")),
]
```

- Defines the paradigm:

```
masc_accent_b = paradigms.Paradigm(
    category=noun,
    name="hard stem masculine accent B",
    slots=slots,
    parent_paradigm=masc_accent_a,
    lemma_feature_vector=lemma_features,
    stems=["górb", "stól"],
    rules=[deaccentuation]
)
```

To demonstrate the use of the two paradigms, consider the following function, which prints all the forms of a given stem using the `stem_to_forms` transducer to generate the wordform itself, and the `feature_label_rewriter` transducer to produce a human-readable representation of the feature vector.

```
def print_forms(noun: str, pd: paradigms.Paradigm) -> None:
    lattice = rewrite.rewrite_lattice(
```

```
        noun,
        pd.stems_to_forms @ pd.feature_label_rewriter
    )
    for wordform in rewrite.lattice_to_strings(lattice):
        print(wordform)
```

The following interactive session shows the outputs for *grádus* and *stól*.

```
>>> print_forms("grádus", masc_accent_a)
grádus+ami[case=ins][num=pl]
grádus+am[case=dat][num=pl]
grádus+ax[case=prp][num=pl]
grádus+a[case=gen][num=sg]
grádus+e[case=prp][num=sg]
grádus+om[case=ins][num=sg]
grádus+ov[case=gen][num=pl]
grádus+u[case=dat][num=sg]
grádus+y[case=nom][num=pl]
grádus+y[case=acc][num=pl]
grádus[case=nom][num=sg]
grádus[case=acc][num=sg]
>>> print_forms("stól", masc_accent_b)
stol+ý[case=acc][num=pl]
stol+ý[case=nom][num=pl]
stol+ú[case=dat][num=sg]
stol+óv[case=gen][num=pl]
stol+óm[case=ins][num=sg]
stol+é[case=prp][num=sg]
stol+á[case=gen][num=sg]
stol+áx[case=prp][num=pl]
stol+ám[case=dat][num=pl]
stol+ámi[case=ins][num=pl]
stól[case=acc][num=sg]
stól[case=nom][num=sg]
```

6.5.2 TAGALOG INFIXATION

Consider an example where affixation is not merely concatenative. Like many other Austronesian languages, Tagalog, spoken in the Philippines, makes use of **infixes**, affixes which attach to the middle of stems. To form the actor-focus infinitive form, one inserts *-um-* between a word-initial consonant C and the following vowel V, or initially—i.e., as a prefix—if there is

Table 6.2: Tagalog *um*-infixation, after Ramos and Bautista (1986).

Lemma	Actor Focus (-um-)	
bilang	b**um**ilang	'count'
ibig	**um**ibig	'love'
kopya	k**um**opya	'copy'
lipad	l**um**ipad	'fly'
punta	p**um**unta	'go to'

no word-initial consonant. Some examples are given in Table 6.2, and the following snippets implement Tagalog *um*-infixation.

- Constructs the focus feature, the verb category, and the lemma feature vector:[4]

```
focus = features.Feature("focus", "none", "actor")
verb = features.Category(focus)
none = features.FeatureVector(verb, "focus=none")
```

- Defines V, C and the stem shape:

```
v = pynini.union("a", "e", "i", "o", "u")
c = pynini.union(
    "b", "d", "f", "g", "h", "k", "l", "ly", "k", "m", "n",
    "ng", "ny", "p", "r", "s", "t", "ts", "w", "y", "z"
)
stem = paradigms.make_byte_star_except_boundary()
```

- Defines *um*-infixation as stem-form rule:

```
um = pynini.union(
    c.plus + pynutil.insert("+um+") + v + stem,
    pynutil.insert("um+") + v + stem
)
```

- Defines the slots:

```
slots = [
    (stem, none),
    (um, features.FeatureVector(verb, "focus=actor")),
]
```

[4] While omitted here, there are several other types of focus in Tagalog.

- Constructs the paradigm:

```
tagalog = paradigms.Paradigm(
    category=verb,
    slots=slots,
    lemma_feature_vector=none,
    stems=["bilang", "ibig", "lipad", "kopya", "punta"],
)
```

The following interactive session shows the outputs for *bilang* and *ibig*.

```
>>> print_forms("bilang", tagalog)
bilang[focus=none]
b+um+ilang[focus=actor]
>>> print_forms("ibig", tagalog)
ibig[focus=none]
um+ibig[focus=actor]
```

6.5.3 YOWLUMNE ASPECT

An even more complex example comes from Yowlumne (formerly Yawelmani), an endangered language spoken in California. This particular example comes to us from Newman (1944) via Archangeli (1984).[5] In this language, verbal aspect is expressed via suffixation with concomitant reshaping of the verb stem. These shapes can be described in terms of **templates** of vowels and consonants, and thus Yowlumne provides a novel example of **templatic morphology** not dissimilar to better-known examples like Arabic and Hebrew.

Four aspectual suffixes can attach to the verb root: dubitative *-al*, passive aorist *-t*, gerundial *-inay*, and durative *-aa*.[6] The first two suffixes are said to be "neutral" because, unlike the latter two, they do not trigger any stem changes. However, the gerundial, for example, is associated with a template represented by the regular language $CVCC^?$ where C is the set of consonants and V the set of vowels. This does not trigger any changes to *caw* 'shout', because it is a subset of $CVCC^?$, but with the stem *hoyoo* 'name', the gerundial stem is realized as *hoy*, with truncation of the final long vowel. Similarly, the durative is associated with a $CVCVVC^?$ template, where the doubled V indicates a long vowel. Thus, the stems *caw* and *ilk* 'sing' have durative stems *cawaa-* and *ʔiliik-*, respectively. Additional examples are provided in Table 6.3.

Archangeli (1984) describes the conventions necessary to map stems onto templates, but what matters here is that these mappings are also regular relations. The transducers implementing these templates may either insert or delete material, and the paradigm slots are defined

[5] Blevins (2004) and Weigel (2005) note that much of the Yowlumne data used by Archangeli others consist of hypothetical words posited by earlier authors—particularly Kuroda (1967)—on the basis of rules and stems provided by Newman (1944). Therefore, these particular data should be taken with a grain of salt.

[6] In these transcriptions ʔ represents the glottal stop, *c* a voiceless alveolar affricate. Long vowels are indicated by doubled vowels.

Table 6.3: Yowlumne aspectual suffixes, after Archangeli (1984:252). Note that the *-n* suffix after the durative ending *-ʔaa* is a tense suffix: see Newman (1944:97).

Lemma	-al	-t	-inay	-ʔaa(-n)	
caw	cawal	cawt	cawinay	cawaaʔaa-n	'shout'
cuum	cuumal	cuumt	cuminay	cumuuʔaa-n	'destroy'
hoyoo	hoyooal	hoyoot	hoyinay	hoyooʔaa-n	'name'
diiyl	diiylal	diiylt	diylinay	diyiilʔaa-n	'guard'
ʔilk	ʔilkal	ʔilkt	ʔilkinay	ʔiliikʔaa-n	'sing'
hiwiit	hiwiital	hiwiitt	hiwtinay	hiwiitʔaa-n	'walk'

by inserting the appropriate suffixes and composing the stem with the appropriate affix. The $CVCVVC^{?}$ template requires one to copy certain vowels to indicate lengthening. Generally speaking, string relations that copy arbitrary unbounded sequences are not rational relations, but one can simulate this effect using an iterated union over the eligible segments. The following snippets implement the Yowlumne aspectual system as a paradigm.

- Constructs the aspect feature, the verb category, and the lemma feature vector:

```
aspect = features.Feature(
    "aspect", "root", "dubitative", "gerundial", "durative"
)
verb = features.Category(aspect)
root = features.FeatureVector(verb, "aspect=root"))
```

- Defines C, V, and the step shape:

```
c = pynini.union(
    "c", "m", "h", "l", "y", "k", "?", "d", "n", "w", "t"
)
v = pynini.union("a", "i", "o", "u")
stem = paradigms.make_byte_star_except_boundary()
```

- Defines the $CVCC^{P}$ template for *-inay*:

```
cvcc = (
    c + v + pynutil.delete(v).ques +
    c + pynutil.delete(v).star + c.ques
).optimize()
```

- Defines the $CVCVVC^{?}$ template for -ʔ*aa*:

```
cvcvvc = pynini.Fst()
for vowel in ["a", "i", "o", "u"]:
    cvcvvc.union(
        c + vowel + pynutil.delete(vowel).ques +
        c + pynutil.delete(vowel).star +
        pynutil.insert(vowel + vowel) + c.ques
    )
cvcvvc.optimize()
```

- Defines the slots:

```
slots = [
    (stem, root),
    (paradigms.suffix("+al", stem),
     features.FeatureVector(verb, "aspect=dubitative")),
    (paradigms.suffix("+inay", stem @ cvcc),
     features.FeatureVector(verb, "aspect=gerundial")),
    (paradigms.suffix("+?aa", stem @ cvcvvc),
     features.FeatureVector(verb, "aspect=durative")),
]
```

- Constructs the paradigm:

```
yowlumne = paradigms.Paradigm(
    category=verb,
    slots=slots,
    lemma_feature_vector=root,
    stems=["caw", "cuum", "hoyoo", "diiyl", "?ilk", "hiwiit"]
)
```

The following interactive session shows the outputs for *caw* and *ilk*.

```
>>> print_forms("caw", yowlumne)
caw[aspect=root]
cawaa+?aa[aspect=durative]
caw+inay[aspect=gerundial]
caw+al[aspect=dubitative]
>>> print_forms("?ilk", yowlumne)
?ilk[aspect=root]
?iliik+?aa[aspect=durative]
```

Table 6.4: Examples of Latin lemmas and stems for three conjugations.

	Lemma	1st Stem	2nd Stem	3rd Stem	
1st conj.	laudō	laud-	laudāv-	laudāt-	'praise'
2nd conj.	moneō	mon-	monu-	monit-	'warn'
3rd conj.	agō	ag-	ēg-	act-	'drive'

```
?ilk+inay[aspect=gerundial]
?ilk+al[aspect=dubitative]
```

6.5.4 LATIN VERBS

Finally, the Pynini distribution includes a fairly extensive treatment of over 2,500 Latin verbs from three conjugations. Salient properties of Latin verbs include agreement with the subject's **person** (first, second, or third) and **number** (singular or plural), **active** and **passive** voices for most verbs, and three separate stems. Roughly speaking, the first stem is used for present, future and imperfect forms, and the present participle, the second stem for perfect aspectual forms, and the third stem for participial forms and various productive nominalizations. Examples of these stems are given in Table 6.4. In all cases it is assumed that the citation form, the first person singular active indicative present, is the lemma.

The relation between the stems of the first conjugation is largely regular in that most verbs of this conjugation exhibit a pattern similar to that of *laud*, the second and particularly third conjugations are much more irregular. Some of the most complex patterns include **reduplication** (e.g., the second stem of *spondeō* 'promise' is *spopond-*) and **suppletion**, the use of phonologically dissimilar stems within a single paradigm (e.g., the second and third stems of *ferō* 'bring' are *tul-* and *lāt-*, respectively). However, it is notable that no matter the shape of a given stem, that stem is used in the same contexts across all verbs; for example, there are no exceptions to the generalization that passive participles and nominalizations in *-iō* are built from the third stem. While there are some subregularities, the third stem of a verb is not fully predictable from the verb's other stems, and there do not seem to be any particular syntactic or semantic commonalities between the various uses of the three stems. Aronoff (1994) refers to morphological generalizations that are phonologically, syntactically, and semantically arbitrary—like the Latin third stem—as **morphomic**.

FURTHER READING

Aronoff and Fudemann (2011) provide an accessible introduction to the theory of morphology. Sproat (1992) describes the early history of computational morphology.

Many of the computational issues and examples discussed in this chapter are addressed in greater detail by Roark and Sproat (2007); their Chapter 2 reviews many examples of morphological processes, including fragments of Latin, Tagalog, and Yowlumne, Chapter 3 argues that item-and-arrangement and item-and-process theories are computationally equivalent, and Chapter 5 reviews machine learning approaches to morphology up to that date.

Kurimo et al. (2010) reviews the Morpho Challenge shared tasks on unsupervised morphological learning, held 2005–2010. More recently, the Conference on Natural Language Learning (CoNLL) and the ACL Special Interest Group on Computational Morphology and Phonology (SIGMORPHON) have hosted a series of shared tasks on supervised morphological analysis and generation (Cotterell et al. 2016, 2017, 2018, McCarthy et al. 2019, Vylomova et al. 2020). Beemer et al. (2020) use the results of the 2020 SIGMORPHON shared task to compare "hand-written" morphological analyzers to ones based on neural networks.