THE UNIVERSITY of EDINBURGH

LEXICRUNCH:
AN EXPERT SYSTEM FOR WORD MORPHOLOGY

Andrew R. Golding

M. Phil.
University of Edinburgh
1984

Contents

Abstract

Natural language programs typically store words like **pig** and **pigs** as independent entries in their dictionaries, thus neglecting the obvious morphological relationship between them. Lexicrunch tries to induce such relationships from examples of root forms of words and the corresponding inflected forms.

The program collates the examples into classes according to the difference between the inflected form and its root -- e.g. the classes for the plural noun inflection in English might include "root forms to which an **-s** is added" (**pig**, **apple**, etc.) and "root forms which take **-es**" (**fox**, **box**, etc.). It then characterizes each class using a modified version of Quinlan's ID3 procedure.

The resulting rule will be along the lines of, "If a noun ends in **-x**, form its plural by adding **-es**; otherwise, add **-s**." The program then needs to store only root forms in its dictionary; it can reconstruct plurals on demand by applying its rule. It thereby eliminates redundancy and compacts the lexicon. Lexicrunch's formalism for representing morphological rules was influenced by the Two-level model of Koskenniemi.

The program was tested on the past tense inflection in English, the first person singular present indicative of Finnish, and the past participle in French. It appeared to pick up most of the regularities in the data successfully. However, a meta-level extension to the program is indicated to enable it to capture regularities across its rules.

Chapter 1

Introduction

1.1. The structure of words

The **word** is a rather peculiar level of semantic abstraction. Words are certainly not the "atoms" of meaning, as many multi-word phrases constitute indivisible chunks. **Sweet potato**, for example, is a semantically simple entity -- namely, a yam -- and is by no means a "potato which is sweet." Moreover, clitic words, such as the French **je**, do not even occur in isolation [Matthews 1974, p.168].

On the other hand, widespread structural and semantic regularities within words suggest the existence of a finer type of granule, the morpheme. For instance, in **thicker**, **taller**, and **stronger**, the suffix **-er** appears to carry the independent force of "more."

1.2. The need for a computational model of morphology

Despite the strangeness of the concept, the word is undeniably an integral unit of language. It is the word, and not some simpler component, that is specially demarcated in the written form of most languages. A computer program that processes natural language must therefore have some facility for contending with words. To date, the typical approach has been merely to enumerate every word; only a handful of regular forms are systematically derived from their morphological constituents [Winograd 1983, p.545; Ritchie and Pulman 1984, p.1; Koskenniemi

1983a, p.12].

As the interest in natural language programs grows, it is becoming increasingly apparent that a principled scheme for representing the morphology of words is needed. Such a system would contain a dictionary of morphemes, along with rules for reconstructing words from them. This design would have the obvious virtue of compactness, as it would eliminate a great deal of redundancy. Additionally, it would offer the other classical advantages gained by expressing logical dependencies explicitly: it would facilitate modifications such as spelling changes, and it would potentially be able to predict new words by extrapolating from its rules.

1.3. Producing rules automatically

One practical trouble with such a formalism is creating the desired set of rules in the first place. Even if the native speaker has rules in his head, he normally uses them just to calculate results -- the task of verbalizing the rules themselves is unnatural and difficult. The only alternative is to induce rules of word construction from actual linguistic data. The analysis process demands considerable linguistic expertise, though, and the corpora are generally of formidable proportions.

This is where the work reported here comes in. The Lexicrunch program is designed to automate the laborious business of abstracting morphological rules from examples. It is then able to use its rules to generate and recognize words in the given corpus as required.

1.4. Overview of the program

Lexicrunch is divided into four main modules: word generation, word recognition, data entry, and rule compression.

The first two perform the standard morphological operations of converting back and forth between words and ordered sequences of morphemes. One word can correspond to several morpheme sequences, but the reverse is not true.

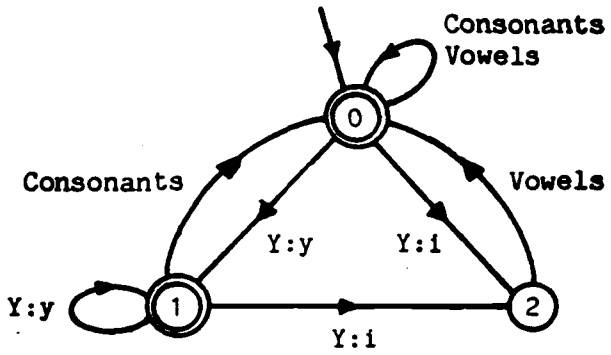
The last two modules are the "meat" of the program: they construct morphological rules from a corpus of examples. The work is split between two modules because the rule-building algorithm is by nature a batch operation, but it is convenient to enter examples incrementally. The batch part of the processing is performed by the rule-compression module; the data-entry module takes care of updating the rule for each example given.

The typical way to build a rule is to type root forms of words and the corresponding inflected forms into the data entry module. For the English plural noun inflection, for example, we would enter pairs like **pig/pigs**, **fox/foxes**, etc. The data-entry routines then do the minimal amount of work necessary to express the list of examples as a rule in Lexicrunch notation. The routines are relatively fast, but their rules tend to be bulky and naive.

Alternatively, the user can bias the program toward a particular formulation by typing in rules directly in Lexicrunch notation. The program compiles such rules into its internal representation. The rule compiler is peripheral to the central

after reading both completely, it ends up in a final state. [Kay 1983, p.100]

A finite-state transducer for the Y rule above might look like:



State 0 is the initial state, and 0 and 1 are final. The label "Consonants" is short for the twenty-one pairs b:b, c:c, ..., z:z; "Vowels" represents the five vowel pairs. Most of the time, the machine matches normal letters (state 0). If it hits a Y on the input tape and an i on the output tape, it makes sure that a vowel follows (state 2). It also checks that there is a consonant, not a vowel after a Y:y pair (state 1). In this manner, the transducer verifies that the desired relationship is maintained between its tapes.

Kaplan and Kay have designed a general procedure for constructing a finite-state transducer from an ordered list of context-sensitive rules. The method involves deriving one transducer for each individual rule, and then merging all the transducers into one large machine, essentially by taking the cross-product of their sets of states. [Kay 1983, p.102]

Chapter 2

Previous work in computational morphology

2.1. The morphemic model

Much of the research in computational morphology has its linguistic roots in the **morphemic model**. Since computational linguistics has generally addressed orthographic phenomena, we will give a graphological account of the model, rather than the more traditional phonological one. The two accounts can be reconciled in light of the indifference between representing words as strings of phonetic symbols or letters of the alphabet.

The central tenet of the morphemic model is that the morpheme is the fundamental unit of grammar; everything else is built up by concatenation. For example, **raised** would be broken down as RAISE* + PAST PARTICIPLE, where RAISE is realized as **raise-** and PAST PARTICIPLE as **-d**. Similarly, **have raised** would be HAVE + (RAISE + PAST PARTICIPLE), and so on, up to the level of the sentence and beyond. The crucial feature of this approach is that it analyzes words in just the same way as it does clauses and other higher-level structures. [Matthews 1974, p.78]

2.1.1. Alternation

In the example above, PAST PARTICIPLE was realized as **-d**. This, of course, is not the only possibility -- in **walked**, e.g., it comes out as **-ed**. The fragments **-d** and **-ed** are said to be

* Following Matthews [1974], we will designate morphemes by writing abstract labels for them entirely in upper case.

alternative realizations or **allomorphs** of the PAST PARTICIPLE morpheme. The morphemic model specifies two sorts of conditions under which a particular allomorph is selected for a given word. [Matthews 1974, p.85]

In the case of **graphologically conditioned** alternation, the choice of allomorph is governed by the grapnemic environment [Matthews 1974, p.92]. The **-d/-ed** distinction above falls into this category, as the morpheme is written as **-d** for verbs ending in **e**; after other final letters it is realized as **-ed**.

Grammatically conditioned changes, in contrast, are triggered by the presence of certain morphemes [Matthews 1974, p.92]. A case in point is the deletion of terminal **n** in Finnish nouns before a possessive suffix, as in

taloon	+	-si	→	taloosi
("to the house")		("thy")		("to thy house").

There is no apparent graphological rationale for the change, as the sequence **nsi** can and does crop up in other Finnish words, such as **kynsi-** ("to scratch," stem form). [Koskenniemi 1983a, p.78]

It frequently happens that the morphemes that control a grammatically conditioned alternation are lexical morphemes; in such cases, we say that the alternation is **lexically restricted** [Matthews 1974, p.92]. For instance, although final **f**'s of English nouns usually change to **v**'s before the PLURAL morpheme -- e.g. **elf** becomes **elves**, **loaf**, **loaves**, and **thief**, **thieves** -- the pathological example **oaf** has **oafs** as its plural. We therefore posit a lexical restriction whereby the alternation is triggered

by the morpnemes ELF, LOAF, and THIEF, but not OAF.

2.1.2. Meta-graphemes

Aside from the regular complement of letters in the alphabet, many linguists include meta-graphemic characters in their specifications of morphemes. Meta-graphemes are special symbols that stand for more than one possible grapheme; the choice is made by realization rules. Meta-graphemes are conventionally denoted by capital letters.

Meta-graphemes come in two varieties. The first, **morphographemes**, are intended to be "specially marked" instances of a particular grapheme [Matthews 1974, p.211]. They are used to specify which morphemes activate a grammatically conditioned alternation. Take the **y** in **berry**. In some sense, it is not an ordinary **y**, because it has the power to change to an **i** if a vowel follows. We can capture this by introducing a morphographeme **Y** and an associated rule:

> **Y** is realized as an **i** before vowels, and as a **y** elsewhere.

We then identify the morphemes that trigger the rule by our distribution of the **Y** morphographeme; that is, we write **berrY**, **flY**, and **dairY**, but **valley** and **toy**.

A closely allied species of meta-grapheme is the **archigrapheme**. An archigrapheme can be thought of as the common denominator of the graphemes it represents. The choice among the graphemes should be graphologically conditioned [Matthews 1974,

p.204]. In English, e.g., the nasal grapheme of the negative prefix **in-/im-** inherits the feature of the following consonant*. Thus it is expressed as **n** before the alveolar consonants **d** and **t**, as in **indefinite** and **intolerable**, but it assimilates to **m** before the bilabial consonants **b** and **p** in **imbalance** and **impartial**. It is convenient to unify the two forms of the prefix by replacing their nasals with a nasal archigrapheme, say **N**. We then need only provide rules for realizing **N** as the relevant grapheme.

2.2. Kay's algorithmic formulation

Kay [1983] has recently proposed an elegant and efficient way to implement the morphemic model algorithmically. The primitive operation in his system is, not surprisingly, concatenation. Thus a form like **replayed** is derived from three morphemes:

re- + play + -ed.

[Kay 1983, p.101]

2.2.1. Representing alternations

Alternations are formulated as ordered lists of context-sensitive rules. This works especially well for graphologically conditioned changes, since the graphemic environment that triggers the change becomes the "context." To illustrate, the rule above for **Y** is straightforwardly written as

Y → i / _ Vowel

Y → y

* This example is due to John Phillips.

[Kay 1983, p.102].

Kay makes grammatically conditioned rules look graphological by tagging the morphemes that activate the change. In the example above of **n** deletion in Finnish, we would distinguish all the possessive suffixes, e.g. by prepending a **$** onto them, yielding **$si**, etc. This allows us to express the deletion in terms of context:

$$n \rightarrow \emptyset \; / \; \_ \; \$$$

$$\$ \rightarrow \emptyset$$

where **∅** stands for the empty string. [Koskenniemi 1983a, p.78]

2.2.2. Compiling rules into finite-state transducers

The traditional interpretation of a rule like

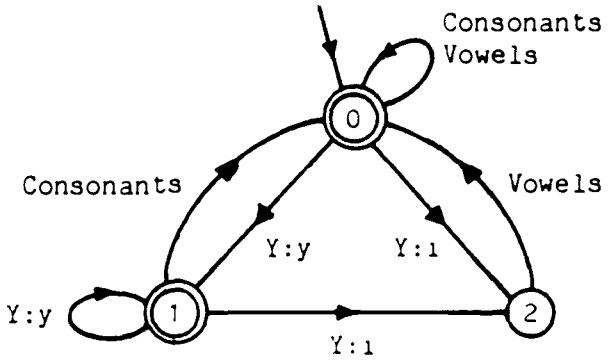
$$\alpha \rightarrow \beta \; / \; \gamma \; \_ \; \delta$$

is that α, if flanked by γ and δ, is rewritten as β. Kay's perspective differs in that he views the rule as a correspondence between α in the input string and β in the output string. He shows how a simple finite-state transducer can be used to verify such a correspondence. [Kay 1983, p.100]

A finite-state transducer is the same as an ordinary finite-state machine, except that it scans two tapes instead of one. Accordingly, its transitions are labelled with pairs of characters, one for each tape. One character in the pair can be ε, meaning that the corresponding tape is not read on that transition. The transducer is said to accept a pair of tapes if,

after reading both completely, it ends up in a final state. [Kay 1983, p.100]

A finite-state transducer for the **Y** rule above might look like:



State 0 is the initial state, and 0 and 1 are final. The label "Consonants" is short for the twenty-one pairs b:b, c:c, ..., z:z; "Vowels" represents the five vowel pairs. Most of the time, the machine matches normal letters (state 0). If it hits a **Y** on the input tape and an **i** on the output tape, it makes sure that a vowel follows (state 2). It also checks that there is a consonant, not a vowel after a **Y:y** pair (state 1). In this manner, the transducer verifies that the desired relationsnip is maintained between its tapes.

Kaplan and Kay have designed a general procedure for constructing a finite-state transducer from an ordered list of context-sensitive rules. The method involves deriving one transducer for each individual rule, and then merging all the transducers into one large machine, essentially by taking the cross-product of their sets of states. [Kay 1983, p.102]

2.2.3. Generation

Once we have the transducer for a set of rules, it is an easy matter to generate inflected forms from lexical entries. First we string together the given prefixes, root, and suffixes -- this gives us the input tape for the transducer. Since the output tape is not available, we guess the characters on it nondeterministically. The transducer tells us when we guess incorrectly. [Kay 1983, p.102]

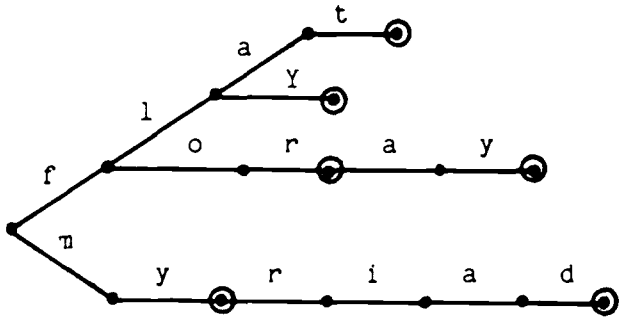
Prima facie, this "trial and error" method may seem inefficient, but in practice there are very few reasonable guesses at each step: most guesses will result in character pairs for which there is no transition in the transducer. A more serious drawback is that the number of states in the transducer grows exponentially with the number of context-sensitive rules, and hence can get out of control. One way to take the edge off the explosion is to split the rule set into clusters, build a transducer for each, and cascade the transducers -- the output of the first serves as the input to the second, etc. Unfortunately, the resulting succession of machines is much less time-efficient than a single large transducer. [Karttunen 1983, pp.174-5]

2.2.4. Recognition

The technique for recovering the prefixes, root, and suffixes from which a word is derived is totally analogous to the inverse process of generation. The only difference is that this time the output tape is provided and we have to guess the input. [Kay 1983, p.102]

Altnough the same formal mechanism suffices for both recognition and generation, there is an important asymmetry between them: and that is that recognition is inherently more ambiguous. Even the simple automaton above accepts several possible lexical strings for, e.g., **myriad** -- **myriad**, **mYriad**, **myrYad**, and **mYrYad** -- but a given lexical string will have only one corresponding output form. The situation improves if we discard nonsense results from the recognition procedure, by allowing it to return only those root forms that are listed in our dictionary. [Kay 1983, p.103]

The root dictionary is stored in tree form, as illustrated by the dictionary below of the words **flat**, **flY**, **for**, **foray**, **my**, and **myriad**.



It can be thought of as a sort of transition network, with its root node as the initial state, and the endings of entries as final states. We can use it to constrain our guesses during recognition by traversing it in parallel with the transducer, feeding it the characters as we guess them. It will prevent us from guessing sequences of characters tnat do not form real words. Note that in the general case we will require a linked chain of lexicons -- one or more for prefixes, one for roots, and one or

more for suffixes. [Kay 1983, p.104]

2.3. Koskenniemi's Two-level morphology

The Kay framework was further developed by Koskenniemi in his Two-level morphology. Koskenniemi implemented the model in Pascal, and demonstrated that it could handle the full spectrum of constructions of Finnish inflectional morphology [Koskenniemi 1983a]. Additional support for the model comes from the KIMMO project, in which it was successfully applied to toy grammars of English [Karttunen and Wittenburg 1983], French [Lun 1983], Japanese [Alam 1983], and Rumanian [Khan 1983].

2.3.1. Enhancements to the Kay framework

A significant hitch in Kay's framework was that for substantial rule sets, the automata turn out to be either prohibitively large or, if they are in series, slow. Koskenniemi overcomes this predicament by ordering the transducers in parallel. Each machine enforces its own particular context-sensitive rule between the lexical string and the inflected form. If any machine fails, the pair of tapes is rejected. The parallel configuration does away with intermediate stages in the analysis and synthesis processes. We are left with only two levels (hence the name of the model): the **lexical** level and the final inflected product or so-called **surface** level. [Koskenniemi 1983a, p.15]

Koskenniemi's other contribution was his implementation of Kay's idea of structuring the dictionary as a network of mini-lexicons. This entails partitioning the lexical entries by

"morphological function." For instance, all root morphemes would go in one lexicon; in French, the present tense **-er** verb endings **-e**, **-es**, **-ons**, **-ez**, and **-ent** would be grouped together; etc. Also, each lexical entry contains a **continuation class**, a list of mini-lexicons whose members can legally follow the entry. The continuation class for French verbs would include mini-lexicons for present tense endings, past tense endings, imperfect endings, and so on. In this way, we can specify which sequences of morphemes are allowed. [Koskenniemi 1983a, p.29]

2.3.2. Rule notation

Koskenniemi adopts a powerful, terse notation for expressing his Two-level rules. For our purposes, though, a simplified variation will do.

Characters are expressed as dotted pairs of the lexical version of the character and the surface version. Hence **(Y.i)** is a pair which is a **Y** in the lexical level, but corresponds to an **i** in the inflected form. The pair can be abbreviated as a single symbol if the lexical and surface characters are identical -- e.g. **y** is synonymous with **(y.y)**. [Koskenniemi 1983a, pp.31-2]

Symbols for sets of characters are permitted in lieu of single characters, though the interpretation is not the obvious one. Say **V** = {**a**, **e**, **i**, **o**, **u**}. **(V.V)**, or simply **V**, does not necessarily represent all twenty-five possible vowel pairs; it stands only for those that appear <u>explicitly</u> in other rules. Thus if **(a.u)** is never mentioned in any rule, it is not included in the interpretation of **V**. The symbol **=** is a predefined set consisting

of all characters in the alphabet. The complement of a set is indicated by putting a minus sign in front of it. Thus **(=.V)** refers to all explicitly appearing character pairs that are realized as vowels at the surface level; **(=.-V)** stands for all such pairs that are realized as surface non-vowels. [Koskenniemi 1983a, pp.31-2]

In our Two-level notation, one way to write the first part of our earlier rule for Y is

(Y.i) <=> _ (=.V)
where V = {a, e, i, o, u}.

The rule is interpreted as follows: lexical **Y** corresponds to surface **i** if and only if there is a vowel after the **i** in the surface string.

In general, the left-hand side of a Two-level rule specifies a correspondence between a lexical form and a surface form. The right-hand side gives the environment of the correspondence. The left and right contexts are regular expressions of character pairs, and can include optional parts, enclosed in square brackets, as well as Kleene closures, indicated by an asterisk superscript. [Koskenniemi 1983a, pp.33-4]

Most rules, like the one above, stipulate that a lexical character is realized in a certain way **if and only if** it is surrounded by the given context. Alternatively, **only if** rules, identified by the => operator, state that a lexical character can be realized as a particular surface form only in the given environment. Lastly, **if** rules, written with a <=, mean that if

tne lexical character occurs in the specified context, then its surface counterpart has to be the character given. [Koskenniemi 1983a, pp.36-7]

2.3.3. Two-level rules for Finnish

Because of its rich morphology, Finnish provides an excellent testbed for programs that recognize or generate word-forms. Below we give examples of how Koskenniemi uses his Two-level formalism to describe representative phenomena in the language.

2.3.3.1. Vowel harmony

There are three types of vowels in Finnish: harmonizing front (**ä, ö, y**), harmonizing back (**a, o, u**), and neutral (**i, e**). We say that Finnish has **vowel harmony** because harmonizing front and back vowels cannot mix in a word. The vowels of suffixes must therefore conform to tne frontness or backness of the root to which they are added. Neutral roots take front vowels. [Koskenniemi 1983a, p.76]

Koskenniemi represents vowels in suffixes by archigraphemes: **A** for **a/ä**, **O** for **o/ö**, and **U** for **u/y**. His rules for vowel harmony are then:

(A.a) => (=.Vb) G* _ (A.Ä) => # G* _ (A.Ä) => (=.Vf) G* _

(O.o) => (=.Vb) G* _ (O.Ö) => # G* _ (O.Ö) => (=.Vf) G* _

(U.u) => (=.Vb) G* _ (U.y) => # G* _ (U.y) => (=.Vf) G* _

where Vb = {a, o, u}
Vf = {ä, ö, y}
G = the set of harmonically indifferent letters; i.e. consonants and neutral vowels
# = the beginning-of-word marker

The first column of rules guarantees that **A**, **O**, and **U** will be expressed as back vowels only if the stem contains a back vowel; the second column checks that the archigraphemes turn into front vowels for neutral stems; and the third column ensures front vowels for front stems. [Koskenniemi 1983a, p.76]

2.3.3.2. Consonant gradation

The term **consonant gradation** refers to the alternation in Finnish words of certain pairs of consonant clusters. One member of each pair is said to be strong, and the other, weak. Listed with the strong grade of the pair first, the alternations are

pp ~ p	p ~ v	mp ~ mm		
tt ~ t	t ~ d	lt ~ ll	rt ~ rr	nt ~ nn
kk ~ k	k ~ Ø/v	nk ~ ng.		

[Lehtinen 1967, p.523]

To predict consonant gradation in general, we need two pieces of information: (1) which instances of the consonant groups above are subject to gradation, and (2) when gradation takes place, which grade is selected.

The rule for (1) is that a consonant group can alternate if

it comes right before the last vowel(s) of a stem, or sometimes a suffix of the stem. Gradation is blocked if the consonant group is preceded by **s** or **t**. Thus the **nk** in **Helsinki-** ("Helsinki," stem form) alternates between **nk** in **Helsinkiin** ("of Helsinki") and **ng** in **Helsingissä** ("in Helsinki"); but the **t** in **posti-** ("mail," stem form) remains **t** in **postia** ("mail," partitive singular), **postin** ("of mail"), etc., as it follows an **s**. [Lehtinen 1967, p.523]

As for (2), the strong consonant grade occurs roughly in open and/or long syllables, the weak grade in short closed syllables. There are several classes of exceptions, however -- e.g. nouns with the plural **-i-** have the same consonant grade as the singular, regardless of syllable length. Despite their complexity, the exception classes are more or less well-defined. [Lehtinen 1967, pp.523-5]

Koskenniemi copes with (1) by flagging all gradable consonants with the morphographemes **K**, **P**, and **T**. For example, he spells **katto-** ("roof," stem form) as **katTo-**, indicating that only the last **T** is mutable. He prefers this technique to a more rule-oriented one because rules like the one given above often fail for neologisms and foreign names. [Koskenniemi 1983a, pp.78-9]

For (2), Koskenniemi can afford to ignore consonant gradation when the strong grade is chosen, since it is just as if gradation did not happen at all. The problem is thereby reduced to detecting when the weak grade is called for. Unfortunately, the "short closed syllable" rule above is unweildy to implement, particularly because of its exceptions. Thus Koskenniemi inserts a **$** morphographeme in all endings which effectively produce short

closed syllables at the end of the stem. Basically, **$** is a grammatical trigger for the weak grade*. It follows that

(K.k) <= _ -$ (P.p) <= _ -$ (T.t) <= _ -$.

That is, if the trigger for the weak grade is absent, we must have the strong grade. Similarly, if there is an inhibitory **s****, the strong grade prevails:

(K.k) <= s _ (P.p) <= s _ (T.t) <= s _.

And if the trigger is present, but not the inhibitor, then the strong grade is prevented:

(K.-k) <= -s _ $ (P.-p) <= -s _ $ (T.-t) <= -s _ $.

[Koskenniemi 1983a, p.81]

The way Koskenniemi set up his morphographemes, however, there are actually <u>multiple</u> weak grades. **P**, for instance, can be realized as **ø**, **v**, or **m**. Thus we have the following rules†:

(P.v) => V [Cl] _

(P.m) => V m _

(P.ø) => V [Cl] p _

where V = {a, e, i, o, u, y, ä, ö}
Cl = {r, l}.

These give the environments that distinguish which weak grade is at issue. Note the contextual **m** and **p**, which we incorporated

* Koskenniemi's actual trigger is somewhat more complicated, but for reasons that need not concern us here.
** Koskenniemi does not give analogous rules for inhibitory **t**, probably because it is quite rare.
† Again, one of the rules is slightly simplified to avoid irrelevant technical detail.

directly in the formulation above as **ap** and **pp**. Koskenniemi gives analogous rules for disambiguating **K** and **T**. [Koskenniemi 1983a, pp.79-80]

2.3.3.3. "Non-natural" alternation

Usually, the inflected forms of a Finnish noun are readily derived from its stem by suffixation. There exist many noun classes, however, for which this is not the case. These classes are said to exhibit "non-natural" alternation patterns. The nouns in these classes typically have a characteristic pattern at the end of their stems. [Koskenniemi 1983a, p.53]

One such class is made up of all Finnish nouns ending in **-nen**, e.g. **hevonen** ("horse"). The singular stem for these nouns is constructed by replacing the **-nen** ending with **-se**, e.g. **hevose-** ("horse," stem form). This **nen-se** alternation is both common and productive in Finnish. [Koskenniemi 1983a, p.55]

Koskenniemi deals with the **nen-se** alternation by stripping the **-nen** from each lexical entry, yielding, e.g., **hevo**. The words are then given a continuation class which points to a special mini-lexicon of suffixes. One suffix restores the **-nen** ending for the appropriate inflected forms, and another gives the **-se** ending. The alternation is thus handled entirely in the lexicon. [Koskenniemi 1983a, p.55]

Chapter 3

The Lexicrunch rule formalism

3.1. Describing morphological changes

The first-order approximation that words are assembled by stringing together prefixes, a root, and suffixes goes a long way in accounting for morphological data, as attested to by the success of Koskenniemi's program. It starts to falter, however, when we try to represent changes that occur inside words. Below we discuss the difficulties that arise and a possible cure for them.

3.1.1. The problem with word-internal changes

Finnish is a relatively well-behaved language in that most of its morphological changes can be explained coherently as affixations. A possible counterexample, though, is the **nen-se** alternation of section 2.3.3.3. Koskenniemi only got away with treating **nen** as a "default suffix" because all the words at issue had a common termination. He could not have done so had the change occurred "deeper" in the words.

English is a nastier case. Consider the verb **drive** and its past tense **drove**. Using the tactic above, we would store **dr** as the lexical entry, **-ive** as a present tense suffix, and **-ove** as the past tense suffix. In like manner, the past tense suffix for **arise** would be **-ose**, that for **write**, **-ote**, and that for **ride**, **-ode**.

This approach amounts to barely more than memorizing each past tense form. In fact, Karttunen prefers to do just that in his Two-level grammar of English; e.g., he lists **sleep** and **slept** as separate entries in the lexicon, despite the existence of the similar verbs **creep**, **keep**, **sweep**, **weep**, **feel**, and **kneel**. [Karttunen and Wittenburg 1983, p.226].

One other option for representing the **drive/drove** alternation is to introduce a meta-grapheme I, grammatically conditioned to appear as i in the present tense and o in the past -- nence **drIve**, **arIse**, etc.

Technically adequate though this solution may be, it goes against the very raison d'être of all morphology programs: to capture systematic regularities in word structure. There are two regularities at issue in the **drive** example. The first is that there is a rule for identifying which words undergo the given change -- this point is taken up in section 3.2.1. The second is that the vowel mutation does not occur at a random position within a word: it consistently takes place at the third-to-last letter. Yet by carefully planting an I at the site of the change in every case, we are "building in" the rule about the antepenultimate letter in a rather ad hoc and underhanded fashion.

One might argue, in defense of the meta-grapheme solution, that it is nevertheless acceptable to "memorize" the answer, as the classes of English words that undergo internal change are unproductive and small.

The claim of unproductivity was addressed in Berko's

psychological experiment [1958] in which she presented children and adults with nonsense words, and asked for their inflected forms. She concludes that:

> Whereas the children all used regular patterns in forming the past tense, we found that for adults strong pasts of the form **rang** and **clung** are productive. Since virtually all English verbs that are in the present of an **-ing** form make their pasts irregularly, this seems a likely supposition. Adults make ***gling** and ***bing** into ***glang** and ***bang** in the past... The productivity of the **-ang** and **-ung** forms proves that new forms are not necessarily assimilated to the largest productive class. [Berko 1958, pp.175-6]

Thus rules for word-internal changes appear to have at least some validity for English, and are not "pure fiction."

As for the smallness claim, it seems expedient to note all regularities in the database, as long as the resulting rule takes up less space than did its data.

While the problems of word-internal change are serious in English, they simply get out of control in Arabic. Arabic verbs have discontinuous stems, e.g. **k-t-b** ("write"). Their inflected forms are produced by inserting vowels in the interstices -- for example, the perfective of **k-t-b** is **katab**, and its imperfective stem is **-ktib** [Matthews 1974, p.131]. To describe this relationship with our current model, we would have to resort to pairs of meta-graphemes -- e.g. **kXtYb** -- as well as complex contextual rules for realizing them appropriately. Such examples appear to defy our paradigm that inflections are "essentially affixations," with a few scattered contextual "side effects" thrown in.

3.1.2. Morphological processes

Koskenniemi himself admits that word-internal changes challenge the morphemic model:

> Only restricted infixation and reduplication can be handled adequately with the present system. Some extensions or revisions will be necessary for an adequate description of languages possessing extensive infixation or reduplication. [Koskenniemi 1983a, p.27]

The difficulty has also been acknowledged in the linguistic literature. There the proposal for **morphological processes** has been raised. [Matthews 1974, p.120]

A morphological process is a general operation which transforms a word into an inflected form. Using this more powerful notion, we can conveniently explain the change from **drive** to **drove** as one of replacing the **i** with an **o** -- the same process works for **arise**, **write**, and **ride**. Note that we do not insert special markers in each word to indicate which letter alternates.

Concomitant with the utility of our new machinery, however, is a Pandora's box of representational issues. In particular, what exactly should morphological processes be allowed to do? If concatenation is not efficacious, what ought to be the primitive operation? Due to a recent neglect of morphological theory, linguists have little to say in this regard [Matthews 1974, p.120].

3.2. Defining rule domains

The shortcoming of Koskenniemi's method of defining rule

domains is, in a nutshell, that

> In the Two-level lexicon, regularities are explicitly marked in the word entries. Some markings are expressed as morphophonemes [sic] in the representation, some as continuation classes. [Koskenniemi 1983a, pp.128-9]

Adding a word to Koskenniemi's dictionary therefore involves embellishing it with a variety of special flags that indicate which alternations it undergoes, and selecting or creating a continuation class that specifies which suffixes it takes. Koskenniemi is willing to enumerate all of this information because the task of building lexical entries is for him a one-shot deal performed by an expert "external component" -- namely, Koskenniemi himself [Koskenniemi 1983a, p.129]. We will be concerned with eliminating redundant lexical markers, substituting rules when possible.

3.2.1. The problem with meta-graphemes

Let us return for a moment to the **drive/drove** example of section 3.1.1. We said that the most likely means of representing the alternation was to introduce an **I** meta-grapheme into the lexical entries of affected words. This effectively lists the words in the alternation class, thereby neglecting the trend that they all end in /rai/* + consonant. While this rule is not perfect -- e.g. it spuriously includes **arrive** -- it still explains the data more compactly than a list would. Note also that even though the **I** meta-grapheme does not lengthen entries directly, it uses up a new symbol, which may raise the number of bits needed per

* Phonemes will be enclosed in slashes.

character.

3.2.2. The problem with continuation classes

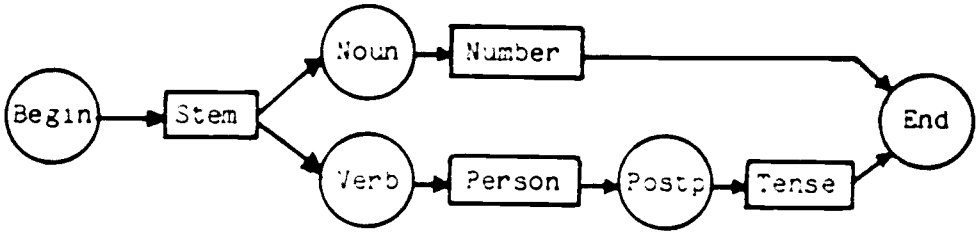
Morphemes in different mini-lexicons only fit together in sequences permitted by their continuation classes [Koskenniemi 1983a, pp.43-4]. Thus the distribution of continuation classes determines the domain of "rules of concatenation." Again, since continuation classes are individually specified for each word, these domains are enumerated by brute force.

The basic job of continuation classes is to define formal categories, e.g. the set of words that take verb endings (namely, verbs). Here it is reasonable to list the domain, since there is presumably no more concise rule to do the equivalent. Various other uses of continuation classes are less well justified, though; a case in point is the **nen-se** alternation of section 2.3.3.3, in which nouns ending in **nen** were enumerated, instead of being selected by the obvious rule.

Apart from listing domains, continuation classes introduce redundancy into the representation in a few other ways.

Following Karttunen, we will diagram the connections between mini-lexicons as a directed graph, where morphemes "flow" along the arcs. Boxes represent mini-lexicons; circles stand for continuation classes. [Karttunen 1983, p.180]

Schematically, a language in which nouns take a number suffix only, and verbs take person followed by tense would be:



Now suppose that there are two sorts of tense endings, **A** and **B**, and the choice depends on the verb stem. We would then have to split the **Verb** continuation class into **VerbA** and **VerbB**, which would lead eventually to the **TenseA** and **TenseB** mini-lexicons, respectively. The consequence is that all mini-lexicons intervening between **Stem** and **Tense** would have to be duplicated so as to "remember" which type of verb stem they are propagating. Long-range dependencies, as between **Tense** and **Stem**, always result in this sort of waste. [Karttunen 1983, p.180]

Prefixes also present special problems for the Two-level framework. If a <u>suffix</u> is allowed only after certain stems, the continuation class of those stems can be suitably modified; but this trick does not work for idiosyncratic <u>prefixes</u>. Take the **re-** prefix in English. Using continuation classes, the only way to indicate that it goes before particular verbs is to establish a purpose-built mini-lexicon of just those verbs, and to give that mini-lexicon as the continuation class of **re-**. The verbs must be enumerated afresh for **mis-**, as the two sets do not coincide exactly -- e.g. **reappear** but ***misappear**.

It is of course preposterous to construct a new mini-lexicon for every prefix. What Karttunen did in his Two-level grammar of English was to give the root lexicon per se as the continuation class for **un-** -- thus his grammar overgenerates, accepting

*unmouse, *unwent, etc. Karttunen proposes extensions to the Koskenniemi framework to enable it to deal with prefixes more gracefully. [Karttunen 1983, pp.221-3]

3.2.3. Characterizing domains via properties

The principal cause for concern in Koskenniemi's use of morphographemes and continuation classes to define rule domains was that it was just like listing the words in the domain. The remedy is to characterize domains by the properties of their members.

The morphemic model recognizes two types of properties that may help describe domains.

For graphologically conditioned alternations, the graphemic environment of the change may be handy. Going back to the **-d/-ed** example of section 2.1.1, we could formulate our rule as:

rule 1: Form the past tense of a verb by adding **-d**.
domain: verbs ending in **e**.

rule 2: Form the past tense of a verb by adding **-ed**.
domain: verbs ending in a letter other than **e**.

Likewise, we should look at the surrounding morphemes in grammatically conditioned alternations. The Finnish rule of **n** deletion from section 2.1.1 can then be stated as:

rule: Delete the final **n** of the stem of a noun.
domain: nouns with a possessive suffix after their stem.

There are certainly other kinds of properties that may be of interest as well. Consider the English **-er** inflection which

produces **teacher** from **teach**, **actor** from **act**, and **sailor** from **sail**. While no apparent grapnological or grammatical property springs to mind to explain the **er/or** alternation, it is still largely predictable by etymological criteria. In fact, the **-or** ending is applied mostly to words which were borrowed directly from Latin; **-er** words came through French, etc. [Jespersen 1946, pp.224-5].

3.3. The structure of rules in Lexicrunch

Two concerns with Koskenniemi's rule formalism have motivated the design of rules in Lexicrunch. The first was the inability of the Two-level model to represent word-internal changes satisfactorily. This is overcome in Lexicrunch by describing inflections in more general terms as morphological processes. Second, Koskenniemi defines domains of rules by enumerating them with meta-graphemes or continuation classes. In Lexicrunch we instead use properties of the words in the domain.

Words are derived, in the Lexicrunch framework, by applying an ordered sequence of rules to a stem, where each rule corresponds to a morpheme. For example,

```
DOG + PLURAL → dogs
MAN + PLURAL → men
MAN + PLURAL + GENITIVE → men's.
```

The morphemic model is like a special case of this, in which the only sort of rules allowed are rules of concatenation. In this section we describe the internal format of Lexicrunch rules, as well as inter-rule relationships.

3.3.1. Changes

The essential component of a Lexicrunch rule is the morphological process which performs the appropriate graphological transformation. Lacking a theoretical groundwork, we will start with as general-purpose a transformation as possible: string substitution. Affixation is a special case, when the string replaced is **Ø**.

We must then specify <u>where</u> in the word the string substitution happens. In **drive**, it was enough to say that "the **i**" changes, but in the Arabic **k-t-b** we would have to talk about inserting vowels between the first or last two letters. We will assume that it is always feasible to describe the positions of strings within words, since native speakers manage to do so -- not that we will try to copy their method.

Lexicrunch, rather arbitrarily, has four ways of identifying change sites: positive/negative offsets, and first/last occurrences of strings. All expressions refer to the left edge of the substring that is replaced. For instance, the **a** in **mailman** that changes to an **e** in the plural can be referenced by five expressions:

mailm·an (the dot marks the change site)

1) +5 (a positive offset)
2) -2 (a negative offset)
3) +an (the first occurrence of a string)
4) -a (the last occurrence of a string)
5) -an (the last occurrence of a string).

As an example of suffixation, consider the addition of **-s** to **frog**:

frog.

1) +4
2) -0.

These expressions for change sites are called **afields**, short for "active fields."

A complete description of a morphological process is given as a list of zero or more **changes**, each consisting of four components:

- **oldstring** -- the substring to be replaced
- **oldfield** -- an afield specifying where the oldstring is located in the untransformed word
- **newstring** -- the substring that does the replacing
- **newfield** -- an afield specifying where the newstring is located in the transformed word (to facilitate the undoing of the change).

A few examples should make the notation transparent:

for frog/frogs:
Ø @ -0 <-> s @ -1
i.e. change a nil at the end of the word to an **s**

for drive/drove:
i @ -1 <-> o @ -o
i.e. replace the last **i** with an **o**

for sleep/slept:
e @ -2 <-> Ø @ -2
Ø @ -0 <-> t @ -1
i.e. delete the **e** two from the end, and add a **t** suffix.

Whether this rendition of morphological processes is universal, in the sense that it can capture with a single formula the transformation for all words "of the same type," is a moot point. The philosophy of the design is that it can always be extended to identify internal positions of words in new and useful

ways. If, say, one wishes to talk about a change at **the nth** syllable, one could envisage a fifth type of afield with the relevant interpretation. In some cases, such extensions would require additional information in lexical entries.

Thus Lexicrunch is more a "linguist's workbench" for experimenting with different types of rules than a definitive answer to the enigmas of morphology. In practice, its simple devices suffice for most exercises. In the worst case, it can decompose rules that are beyond its expressive power into more manageable bits. Take the English rule of doubling the final consonant of certain verbs in the past tense, as in **rib/ribbed, bar/barred, pin/pinned.** Lexicrunch cannot express the rule in its full generality -- it would need three sub-rules:

```
for rib/ribbed:
Ø @ -0 <-> bed @ -3

for bar/barred:
Ø @ -0 <-> red @ -3

for pin/pinned:
Ø @ -0 <-> ned @ -3.
```

3.3.2. The decision tree

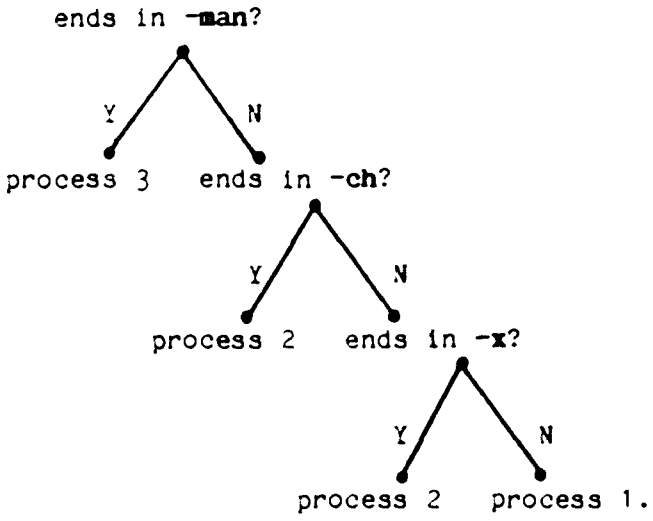
Naturally, one morphological process is not enough to represent a rule like PLURAL. Rules must contain numbered lists of processes -- e.g. for PLURAL:

```
1. Ø @ -0 <-> s @ -1
2. Ø @ -0 <-> es @ -2
3. a @ -2 <-> e @ -2.
```

Process 1 would handle words such as **dog** and **bird**; 2 would take

care of **fox**, **witch**, etc.; and 3, **man**, **woman**, **doorman**, etc. To choose which process to apply to a given word, Lexicrunch has a **decision tree**.

The decision tree has a process number at each of its leaf nodes. An internal node contains a property or test to apply to words; coming out of the node are a Y-branch for words that have the property, and an N-branch for words that do not. A sample decision tree for the PLURAL rule above could be:



To classify a word, we start at the root of the decision tree, applying tests and taking the pertinent outgoing branch, until we reach a leaf node. Henceforth, we will draw decision trees as **if-then-else** statements as below:

```
if "ends in -man" then 3
elseif "ends in -ch" then 2
elseif "ends in -x" then 2
else 1.
```

In the current version of Lexicrunch, only graphological properties are considered. Grammatical properties like the one

above for **n** deletion can be incorporated into the relevant rules directly -- i.e. we would have:

> rule: Form the possessive of a noun by deleting final **n** from the stem and appending a possessive suffix.

Lexical restrictions are handled by the "inclusion" facility described in the next section.

Again, we are not claiming that this range of properties is all-encompassing. The intent is that as the need arises for new kinds of properties, they can be incorporated readily into the existing framework.

The graphological properties or tests in Lexicrunch have the following three components:

- **teststring** -- a string of letters to test for
- **position** -- an afield which, together with the **offset**, describes where in the word to look
- **offset** -- an afield which is added to the position to give a location in the word.

Thus the test **ch @** -0-2 means "ends in -**ch**"; it would select words like **beach** and **match**. The test **ng @** -**i**+1, or "has an **ng** right after its last **i**," picks **ring**, **sing**, and **ingot**, but not **triangle**, **frog**, or **ingrain**. One way (though not the only one) to express our example above is then:

```
if man @ -2-1 then 3
elseif cn @ -2+0 then 2
elseif x @ -1+0 then 2
else 1.
```

3.3.3. Inclusions

When Lexicrunch has trouble finding the right property to characterize a domain, it can always resort to memorizing words. The decision tree above, for instance, is not at all comprehensive, as it misclassifies such words as **Roman** -- the plural should be **Romans**, not ***Romen**. To correct for this, we can explicitly store **Roman** as a member of the domain of process 1. In the notation, words which are specially included under a process are written in curly brackets following its list of changes -- i.e. we get

1. Ø @ -0 <-> s @ -1
   {Roman}
2. Ø @ -0 <-> es @ -2
3. a @ -2 <-> e @ -2.

Now when we classify a word, we first check whether it is an inclusion of any process. If not, we proceed with the decision tree as usual.

3.3.4. Form-class rules

Unlike other morpnemes, stems do not have their own private rules. Rather, all the stems in one form class are listed together in a common rule. We will write form-class rules as demonstrated below:

RULE: noun
{dog bird fox witch man woman,
doorman Roman}.

One might protest that (a) storing words as lists makes the look-up procedure for them inefficient, and (b) it is redundant to

make multiple entries for words in more than one form class.

In response to (a), the notation does not preclude the use of, e.g., lexical trees. As for (b), we should bear in mind that, say, **duck** qua "to lower one's head" and **duck** qua the fowl will require separate semantic entries anyway -- the repetition of the lexeme itself seems negligible. Even in the less extreme example of **telephone**, the meaning of the verb is not trivial to deduce from the meaning of the noun.

3.3.5. Rule ordering

Rules are organized as a directed network: rule **A** can only be applied directly after rule B if there is an arc from B to **A**. The network connections are specified by the list of **predecessors** for each rule. Form-class rules never have any predecessors.

In Lexicrunch notation, our complete PLURAL example would now be:

```
RULE: noun
{dog bird fox witch man woman,
doorman Roman}


RULE: plural > noun
1. Ø @ -0 <-> s @ -1
   {Roman}
2. Ø @ -0 <-> es @ -2
3. a @ -2 <-> e @ -2

if man @ -2-1 then 3
elseif ch @ -2+0 then 2
elseif x @ -1+0 then 2
else 1.
```

The "greater than" symbol indicates a rule's predecessor list. The list can contain more than one element; e.g. if we added a

GENITIVE rule to the example, it might begin with

RULE: genitive > noun plural,

as GENITIVE can be applied directly to nouns -- e.g. **dog's** -- or it can follow the PLURAL morpheme -- e.g. **dogs'**.

Chapter 4

The Design of Lexicrunch

4.1. Generation

Given a stem, optionally its form class, and an ordered list of rules to apply to the stem, the generation routine produces the appropriate inflected form.

4.1.1. The overall algorithm for generation

Should the form class of the stem be omitted, the program first tries to work it out. It considers only those form classes that contain the stem, and that can legally precede the first rule to be applied. If the form class is not thereby determined uniquely, the generation task fails.

Then, for each successive rule in the list of rules to apply, we perform the indicated transformation on the word. The resulting form becomes the input for the next iteration.

To apply a rule to a word we (a) determine which of that rule's morphological processes is relevant for the given word, and (b) apply that process to the word. Procedures (a) and (b) are described in sections 4.1.2 and 4.1.3.

4.1.2. Classifying a word

Here we are given a word and a rule, and we want to decide which of the rule's morphological processes should be applied to the word.

The first step is to check whether the word is listed explicitly as an inclusion to a morphological process; if so, we return that process. Otherwise, we classify the word using the decision tree and the following algorithm*:

```
procedure CLASSIFY (word, dnode):

if dnode is a leaf node then
        return the process number at dnode
elseif word has the property at dnode then
        return CLASSIFY(word, Y-child of dnode)
else
        return CLASSIFY(word, N-child of dnode)
```

The **dnode** in the initial call to CLASSIFY is the root node of the decision tree.

4.1.3. Applying changes

Once we know which morphological process to use on a word, it remains only to effect its list of changes. This is straightforward: for each change, we delete the oldstring at the oldfield, and insert the newstring in its place.

4.1.4. Generation example

Suppose the problem is to generate SING + PAST, where the rule set includes

* We will write algorithms in the Pidgin ALGOL of Aho, Hopcroft, and Ullman [1974, pp.33-9].

```
RULE: verb
{jump bake singe sing ring ping}


RULE: past > verb
1. ∅ @ -0 <-> ed @ -2
   {ping}
2. ∅ @ -0 <-> d @ -1
3. i @ -3 <-> a @ -3

if e @ -1+0 then 2
elseif ng @ -3+1 then 3
else 1.
```

Lexicrunch figures out that SING is a verb, since PAST can only follow verbs. It then applies PAST to SING. SING is not noted as an inclusion -- hence we run the CLASSIFY algorithm, which returns process 3. There is one change to effect:

```
      i @ -3 <-> a @ -3
sing                  sang
```

and so the program generates **sang**. For PING, we would get **pinged**, as this word is explicitly mentioned under the first morphological process.

4.2. Recognition

Recognition is just the opposite of generation: we are given an inflected form, and must recover the stem, form class, and sequence of rules from which it was derived. For words that have more than one derivation, we must return all the possibilities.

4.2.1. The overall algorithm for recognition

The idea of the recognition algorithm is to guess the rules that were applied to the stem, and to "unapply" them in reverse

order.

We start by nondeterministically picking the last rule that was applied to the stem, and unapplying it -- section 4.2.2 explains unapplication. Our guess for the next rule is constrained by the fact that it must be a legal predecessor of the last rule.

The procedure repeats until we reach a form class rule.

4.2.2. Unapplying a rule

In unapplying a rule to a word, we do not know which morphological process of the rule to use. There may even be several correct choices. In French, e.g., **pu** is the past participle of two verbs, **pouvoir** and **paître**.

Thus the algorithm for unapplying a rule is to try uneffecting the changes of every morphological process. A change is uneffected by replacing the newstring at the newfield with the oldstring.

As is, however, this algorithm overgenerates. It would say that one result of unapplying PAST (from section 4.1.4) to **singed** is SING, by uneffecting process 1. This answer is spurious because SING would be classified under process 3, not 1. Such errors are prevented by checking that "de-inflected forms" like SING are classified as assumed.

4.2.3. Recognition example

Referring again to the rule in section 4.1.4, we would analyze **jumped** as follows: say there are just two rules, PAST and VERB. The last rule in the derivation cannot be VERB, because **jumped** is not listed in the VERB form class rule. Hence we proceed to unapply PAST. This gives:

```
                Ø @ -0 <-> ed @ -2
1. jumped            +          jump

                Ø @ -0 <-> d @ -1
2. jumped            +          jumpe

                i @ -3 <-> a @ -3
3. jumped            +          (not applicable)
```

Morphological process 3 cannot be unapplied because there is no **a** to mutate; hence there are two candidate "unpasts": **jump**, produced by process 1, and **jumpe**, produced by process 2. **Jump** and **jumpe** would be classified under processes 1 and 2 respectively, and so both are acceptable de-inflected forms.

The only predecessor of PAST is VERB. **Jump** is indeed listed as a verb, and so this derivation path is JUMP/VERB + PAST. On the other hand, **jumpe** is not a verb, and hence does not lead to a second solution.

4.3. Data entry

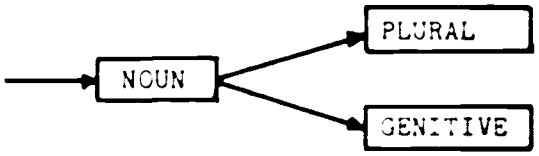
The normal way to present a corpus of data to Lexicrunch is via the set of input routines discussed below. They perform a first pass of analysis on the examples -- just enough to express them in the Lexicrunch rule notation. Further analysis is carried

out by the rule compression module (see section 4.4).

4.3.1. Input routines

There are three input routines. One lets the user establish new form classes; another enters stems into the form classes; and the third is for building predecessor links between rules.

Since the rules in Lexicrunch are arranged as a directed graph, we can think of words as flowing from form classes to other rules, being transformed at each step. Thus in adding a stem to a form class, we are pumping a new word into the network at the source. The new word can ramify through all existing channels to produce new forms. For example, in the following network:



the addition of the noun **frog** gives rise to FROG + PLURAL and FROG + GENITIVE. Likewise, building a link between rules creates a new word-production route. If we added a GENITIVE > PLURAL link above, we would permit for the first time all constructions of the form NOUN + PLURAL + GENITIVE.

Whenever the input routines detect that a new form can be generated, they call another routine (see the next section) which prompts the user for the form in question, and enters his answer into the rule database.

4.3.2. Learning new forms

The task of the routine that learns new forms is to ascertain how to apply a given rule to a given word, and to modify the database accordingly.

First it guesses the answer by applying the existing rule to the word (as in section 4.1). It refrains from guessing if there is no decision tree for the rule*, or if the tree chooses an inapplicable rule -- for instance, it says to change **a @ -2 <-> e @ -2** for **lemon**.

The program displays its guess, and if it is right, no update to the rule set is necessary. Otherwise, it asks for the correct inflected form.

It proceeds to test whether any of the rule's extant morphological processes gives the desired answer. If one does, it stores the given word as an inclusion under that process. In the absence of such a process, it has to invent one, and list the word under it.

4.3.3. Inventing morphological processes

There are, in general, a large number of morphological processes that produce a specified result for a word. To make **read** into **reread**, e.g., we could invoke any of these processes:

* A new rule will not have a decision tree until it is compressed (see section 4.4).

p1. Ø @ +0 <-> re @ +0

p2. Ø @ -4 <-> re @ -6

p3. Ø @ +2 <-> re @ +2

p4. Ø @ +1 <-> er @ +1

p5. r @ +0 <-> rer @ +0

p6. read @ +0 <-> reread @ +0.

To separate the wheat from the chaff we can appeal to a simplicity criterion. We will define the complexity of a process to be the total number of characters it inserts or deletes. P1 through p4 therefore each have complexity 2, whereas p5 has 4, and p6, 10. We will dismiss p5 and p6 from consideration because of their higher complexity.

There is no obvious principle that will select a "best" process from among p1 through p4. It is only when we see other examples of the change -- e.g. type/**retype** -- that we can discard p3 and p4. P2 would be invalidated by **write/rewrite.** Since the input routines only have one example to work with, they pick one of the minimum-complexity processes arbitrarily; their mistakes are rectified later (see section 4.4.1).

The algorithm in Lexicrunch for finding a minimum process is based on techniques from compiler theory for "string-to-string repair" [Backhouse 1979, pp.189-218]. In essence, we scan along the word and result strings simultaneously, trying to match characters as we go. If two corresponding characters differ, we nondeterministically insert or delete a letter to make them the same.

In the Pidgin ALGOL statement of the algorithm below, we refer to the letters in a string as **string**[0], **string**[1], etc. -- strings are assumed to end with a nil. Pointers to strings are passed as integer indices. The procedure returns a minimum list of one-character insertions and deletions, which can easily be spliced into contiguous runs. The initial call would be MINPROCESS(**word**, 0, **result**, 0).

```
procedure MINPROCESS (word, wp, result, rp):

if word[wp] and result[rp] are nil then
        return nil
elseif word[wp] = result[rp] then
        /* match a character */
        return MINPROCESS (word, wp+1, result, rp+1)
else
        if result[rp] is not nil then
                /* try inserting a character */
                insert-solution ← the concatenation of
                        (Ø @ wp <-> result[rp] @ rp) and
                        MINPROCESS(word, wp, result, rp+1)

        if word[wp] is not nil then
                /* try deleting a character */
                delete-solution ← the concatenation of
                        (word[wp] @ wp <-> Ø @ rp) and
                        MINPROCESS(word, wp+1, result, rp)

        return the shorter of insert-solution
                and delete-solution
```

4.4. Rule compression

The compression routines sift through a rule, exploiting its regularities to re-organize it. They restate morphological processes in as general a form as they can, build a decision tree for classifying words by the process they undergo, and finally they tidy up any lingering artifacts of the compression process.

4.4.1. Constructing a set of morphological processes

The input routines are rather haphazard about the morphological processes they invoke to explain inflections. Here we will aim for the most general formulation.

The construction of classes -- morphological processes and their domains -- can be broken down into seven steps, as detailed below.

4.4.1.1. Calculating the domain of the rule

The first step of compression is to figure out the domain of the given rule, i.e. the set of words the rule acts upon. This is accomplished by taking the union of the ranges of all predecessors of the rule. The range of a form class rule is just its list of words. Otherwise the range is calculated from the domain by applying the rule to each word.

We then list each word in the domain as an inclusion of the class to which it belongs. This gives us a complete tabulation of which words are handled by which morphological processes.

To illustrate, say we start out with the following rule set:

```
RULE: noun
{newt tulip apple elbow fox tax box prefix ox switch}


RULE: plural > noun
1. ∅ @ +4 <-> s @ +4
   {newt}
2. ∅ @ +5 <-> s @ +5
3. ∅ @ -0 <-> es @ -2
   {switch}
4. ∅ @ -0 <-> en @ -2
   {ox}
     if x @ -1+0 then 3
     else 2.
```

The domain of PLURAL is just the set of words in the noun form class. After listing the words as inclusions, the classes of PLURAL are:

```
1. ∅ @ +4 <-> s @ +4
   {newt}
2. ∅ @ +5 <-> s @ +5
   {tulip apple elbow}
3. ∅ @ -0 <-> es @ -2
   {fox tax box prefix switch}
4. ∅ @ -0 <-> en @ -2
   {ox}.
```

4.4.1.2. Restating morphological processes

To find the most general formulation of a morphological process, we try expressing its afields in several alternative ways. Each change of a process is varied independently.

Since afields can be written in four different ways, there are in principle at least sixteen ways to express a change -- one for each choice of its oldfield and newfield. The twelve variations in which tne oldfield and newfield are given different types, however, are nighly unlikely candidates for "most general formulation." If, e.g., the oldfield is best expressed as a

negative offset, then we will usually be able to write the newfield as a negative offset calculated by

newfield = oldfield + (length of oldstring)
- (length of newstring).

It is doubtful that we could benefit by writing the newfield instead as a positive offset or a first/last occurrence of a string. Related arguments militate against mixing other types of afields within a change. Hybrid changes may still conceivably have their uses, but too rarely to justify the vast overhead of computing them for every change.

We are left with four variations on changes. The first is to express the oldfield and newfield as positive offsets -- several formulas may result. For the process

i @ -3 <-> ou @ -4
{wind grind find bind},

e.g., we would get two possibilities:

1. i @ +1 <-> ou @ +1
   {wind find bind}
2. i @ +2 <-> ou @ +2
   {grind}.

Second, we try writing the afields as negative offsets, maybe producing multiple answers. Next, if neither the oldstring nor the newstring is nil, we have

oldstring @ +oldstring <-> newstring @ +newstring

for whichever words fit this pattern. Finally, we can do the same with last occurrences of strings. There may again be other

options, such as

i @ -ind <-> ou @ -ound

for our example, but these yield little marginal utility compared to their cost. We will limit ourselves to the aforementioned types, to keep the number of possibilities in check.

Continuing the example of section 4.4.1.1, there will be nine classes after varying the afields:

1. Ø @ +4 <-> s @ +4
   {newt}
1a. Ø @ -0 <-> s @ -1
   {newt}
2. Ø @ +5 <-> s @ +5
   {tulip apple elbow}
2a. Ø @ -0 <-> s @ -1
   {tulip apple elbow}
3. Ø @ -0 <-> es @ -2
   {fox tax box prefix switch}
3a. Ø @ +3 <-> es @ +3
   {fox tax box}
3b. Ø @ +6 <-> es @ +6
   {prefix switch}
4. Ø @ -0 <-> en @ -2
   {ox}
4a. Ø @ +2 <-> en @ +2
   {ox}.

4.4.1.3. Cross-classifying words

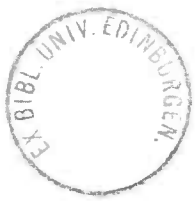
The classes as they stand are incomplete, in that some are missing words which they rightfully deserve. In the example above, class 1a should include **tulip**, **apple**, and **elbow**, since **Ø @ -0 <-> s @ -1** correctly predicts their plurals; also, class 2a should contain **newt**. This is because classes 1a and 2a are not independent: they derive from classes 1 and 2, which both consist of words whose plurals are formed by adding **-s**.

To restore consistency, we must cross-classify words under the appropriate classes outside their **family**, where a family is a group of classes that derive from the same original class, here denoted with the same initial integer.

Cross-classification entails comparing every pair of classe in different families. The morphological process of the firs class is applied to the words in the second: those words that are handled properly are added to the first class. The procedure is repeated for the process of the second class and the words of the first.

Many times, however, it is silly to compare two classes, as there is no hope that the process of one will work for the words of the other. One necessary condition for success is that both processes insert and delete the same characters. Lexicrunch therefore computes two hash values for each process. The first is the sum of the ascii characters deleted; the second, that of those inserted. Pairs of processes with unequal hash values are not compared.

After cross-classification, our running example becomes:



```
1.  Ø @ +4 <-> s @ +4
    {newt}
1a. Ø @ -0 <-> s @ -1
    {newt tulip apple elbow}
2.  Ø @ +5 <-> s @ +5
    {tulip apple elbow}
2a. Ø @ -0 <-> s @ -1
    {tulip apple elbow newt}
3.  Ø @ -0 <-> es @ -2
    {fox tax box prefix switch}
3a. Ø @ +3 <-> es @ +3
    {fox tax box}
3b. Ø @ +6 <-> es @ +6
    {prefix switch}
4.  Ø @ -0 <-> en @ -2
    {ox}
4a. Ø @ +2 <-> en @ +2
    {ox}.
```

4.4.1.4. Simplification

Some of the classes in the cross-classified rule are clearly redundant. Class 1, for instance, can be disposed of, as its word, **newt** -- along with others -- is dealt with adequately by class 1a.

We will say that class **x** subsumes class **y** if the domain of **y** is contained in the domain of **x**; **x** is a more general case of **y**. In such cases, we can get rid of **y** scot-free.

The program simplifies the rule by comparing pairs of classes: if one class is subsumed by the other, it is deleted. If the domains are equal, it is inconsequential which process we get rid of. As a matter of esthetics, the program decides which process is "better," and it eliminates the other one. Basically, the program prefers processes with "better" afields and fewer changes. Numerical afields are "better" than first/last occurrence afields, simply because they are easier to calculate;

between two numerical afields, the one of lower magnitude is preferred; and if both afields are of the first/last occurrence type, the one with the longer string is selected, as it is "more specific." As with cross-classification, the program does not bother comparing two classes with different hash values, as there would be no chance for subsumption.

Following cross-classification, a given word may well be in multiple classes. At the end of the day, though, we ought to decide which of these classes is best, and assign the word to that one only. Simplification is a substantial first step towards this goal, but it does not do away with ambiguous classification entirely. If, say, we are dealing with the past tense rule for English verbs, we could still have these classes after simplification:

1. e @ -2 <-> Ø @ -1
   {breed feed}
2. e @ +3 <-> Ø @ +3
   {breed bite}.

Breed is in both classes, but we have no principled way of picking the "right" one at this stage. Later we will be able to assign it to class 1, on the grounds that this allows the natural characterization of words in class 1 as those that end in **-eed**; there would be no parallel rule had we put **breed** in 2.

Returning to our ongoing example, the simplification process would remove the following classes:

1 -- subsumed by 1a
2 -- subsumed by 1a
2a -- same as 1a, but one of the two must go
3a -- subsumed by 3
3b -- subsumed by 3
4a -- same as 4, but 4 is preferred because its oldfield is of smaller magnitude.

We are left with just three classes:

1a. Ø @ -0 <-> s @ -1
{newt tulip apple elbow}
3. Ø @ -0 <-> es @ -2
{fox tax box prefix switch}
4. Ø @ -0 <-> en @ -2
{ox}.

4.4.1.5. Filtering out insignificant classes

At this point, we are essentially finished constructing the set of classes. This step and the remaining ones merely pave the way for the building of the decision tree.

Since we are creating the decision tree to delineate domains via properties, it makes sense to first throw away those classes that are simply too small to have an interesting characterization. In our example, class 4, which contains only **ox**, is just such a class. **Ox** is a special case -- there is probably no other English word in its class -- and thus there is hardly a basis for extrapolating a rule that defines which nouns take -**en** in the plural.

It is not clear, of course, where to draw the line between these paltry classes and normal ones. Moreover, this issue of judging when a trend is "significant" recurs frequently in Lexicrunch. We resolve the problem somewhat crudely by decreeing

that **m** is a "significant" proportion of **n** if and only if SIGNIF(**m** ,**n**), where

```
procedure SIGNIF (m, n):

if (m > 3) and (m > 10 percent of n) then
        return true
else
        return false
```

The values of the parameters in the function -- "3" and "10 percent" -- were arrived at empirically. Decreasing them would make SIGNIF return "true" more often, and so the program would "cut fewer corners" and run correspondingly slower. If we raised the values, the program would do less work, but its answers would be of a coarser grain. The metric could no doubt be improved by the introduction of more sophisticated statistical methods, but it appears sufficient for the present purposes.

We therefore take to be insignificant those classes of size **n** where SIGNIF(**n**, **n**) is false, i.e. classes such that even if all their members have a certain property, the trend is still not significant. These amount to classes with fewer than four members.

The program withdraws the insignificant classes from consideration -- their words will turn up as explicit inclusions of the rule. If a word is in both significant and insignificant classes, we strike it from the insignificant ones to simplify bookkeeping. This way it retains the opportunity to be classified by the decision tree, and sustains no real loss.

By weeding out insignificant classes, we unclutter our sample

rule by one additional class, yielding:

1a. Ø @ -0 <-> s @ -1
    {newt tulip apple elbow}
3. Ø @ -0 <-> es @ -2
    {fox tax box prefix switch}

4.4.1.6. Devising tests

In order to build a decision tree, we will need some idea of what properties typify the words in each domain of our rule. In this step of the compression procedure, we will comb through the domains, attempting to spot characteristic trends. Specifically, we will look for common strings of letters occurring on either side of a change site -- i.e. the graphological environment of the change (as discussed in section 3.3.2).

Again we must invoke the notion of a "significant" trend, so as to distinguish between useful tests and noise. A "significant" trend is one that holds for **m** out of a domain of **n** words, where SIGNIF(**m**, **n**) (see section 4.4.1.5).

The program searches for trends in a given domain by examining each change of the morphological process in turn. It begins by looking at the first letter of each word to the right of the oldfield of the change. If any letter occurs in a significant proportion of the words, the program goes on to look for a significant following letter. It continues in this manner, finding strings of any length that start to the right of the change site and that turn up in a significant proportion of words. The whole process is repeated for strings that end on the left-hand side of the change site. Finally, if the oldstring of the

change is not nil, the program proposes the very property of having the oldstring at the oldfield.

Consider the class from the English past tense rule for verbs:

```
ea @ -3 <-> o @ -3
Ø @ -0 <-> e @ -1
{swear steal break speak wear tear bear}.
```

We look first at the change **ea @ -3 <-> o @ -3**. To the right of this site, we get **r**'s, **k**'s, and an **l**, but only the **r** is significant, being 4 for 7. There are no more letters after the **r**, and so one property of the class is **r @ -3+2**. We then turn our attention to the left of the change site. The letters are **w**, **t**, **r**, **p**, and **b**, but none is significant. Then, since the oldstring is non-nil, we add the property **ea @ -3+0**.

Having exhausted that change, we try **Ø @ -0 <-> e @ -1**. There are no strings to the right of the change, but to the left are, as above, four occurrences of the letter **r**. Of those four words, all have a preceding **a**; before that they all have an **e**. Beyond this there are no significant letters. Thus our third test is **ear @ -0-3**. This time the oldstring is nil, and so we cannot milk any further properties out of the change.

Note that the properties proposed by our algorithm are not always helpful. In the example above, **ear @ -0-3** was probably the only germane test, though **ea @ -3+0** and **r @ -3+2** could be conjoined to give the same effect. The idea is that these are meant as candidate tests, to be incorporated into the decision tree as the tree-building routine sees fit.

Writing the tests for each domain in parentheses, the running example, after characterization, becomes:

1a. Ø @ -0 <-> s @ -1
    --
    {newt tulip apple elbow}

3. Ø @ -0 <-> es @ -2
   (x @ -0-1)
   {fox tax box prefix switch}

4.4.1.7. Calculating attribute vectors

The last preparatory step before building the decision tree is to calculate which properties each word has. This information will be accessed many times during tree construction.

4.4.2. Building the decision tree

The tree-building routine in Lexicrunch, ID4, is modelled after Quinlan's classification program, ID3 [Quinlan 1979], which is in turn an extension of Hunt et al.'s concept learning system, CLS [Hunt et al. 1966]. The Lexicrunch routine incorporates several enhancements to Quinlan's algorithm.

4.4.2.1. ID3

ID3 takes as input a training set, a partition of the training set into classes, and a list of properties that can be used to characterize each class. From these data, it builds a decision tree which assigns each object in the training set to its proper class. Though Quinlan allows multi-valued properties, we will assume, in the description that follows, that properties are

two-valued, as they are in Lexicrunch -- i.e. an object either has a property or it does not. [Cohen and Feigenbaum 1982, p.407]

The basic ID3 algorithm is:

```
procedure ID3 (set, partition, properties):

if all elements of set are in the same class then
        return a leaf node containing that class
else
        prop ← a property in properties          *
                which is chosen by some heuristic
        setY ← the elements of set that have prop
        setN ← the elements of set that do not have prop
        return a tree whose root contains the property prop,
                whose Y-child is
                ID3(setY, partition, properties - prop),
                and whose N-child is
                ID3(setN, partition, properties - prop).
```

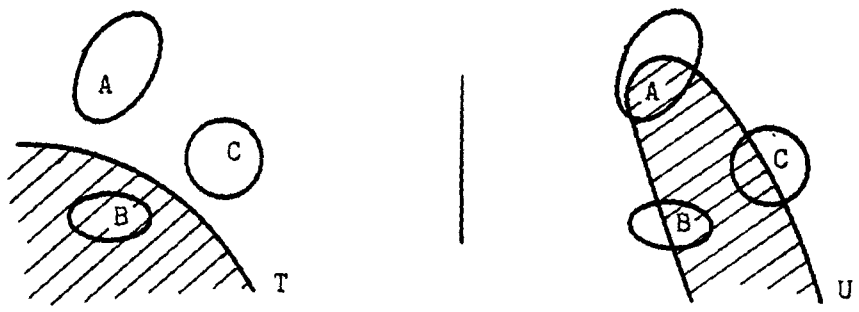
Quinlan also provides a mechanism for sampling the corpus iteratively, in case the whole thing does not fit in core at once. [Cohen and Feigenbaum 1982, pp.407-8]

The heuristic that picks a property to split on should select the test with the greatest discriminatory power -- i.e. it should partition the data as close as possible to class boundaries. [Cohen and Feigenbaum 1982, p.408]

Let us represent the classes pictorially by amorphous shapes, and a property by a line through these shapes. The objects on the shaded side of the line have the property; the objects on the other side do not. Then, for example, test **T** in

---

* The heuristic is discussed below.



is an excellent property to split on, as it allows us to define class **B** as "those objects with property **T**." Test **U**, on the other hand, probably does not help distinguish among the classes.

Quinlan selects properties to split on using an information-theoretic measure, the entropy heuristic. The entropy of a corpus with respect to test **T** is defined as

$$\text{entropy}[T] = \sum_{\text{classes } c} [\, -p_{Tc}\log' p_{Tc} - q_{Tc}\log' q_{Tc}]$$

where

p_{Tc} = proportion of objects in class c that have property T

q_{Tc} = proportion of objects in class c that do not have property T

$\log' x = 0$ for $x = 0$; else $\log' x = \log_2 x$†

Roughly speaking, entropy[T] is the number of binary tests needed in addition to **T** to discriminate among the classes [O'Keefe 1983, p.480]. The entropy heuristic is to choose the test **T** with the lowest entropy[T]. [Cohen and Feigenbaum 1982]

† We can change the base of the logarithm (to numbers greater than 1) if we wish, as this just multiplies the entropies by a constant, but does not affect comparisons between entropy values.

4.4.2.2. ID4

To render ID3 suitable for Lexicrunch, several extensions are indicated. Although the modifications were designed with word morphology in mind, they bear on the classification task in general.

4.4.2.2.1. Providing for exceptions

Implicit in ID3 is the assumption that the given list of properties will always be adequate for explaining why a given object is in the class it is in. If this assumption were violated, there would come a point in the algorithm when a corpus could not be analyzed into its constituent classes -- the program would "break."

In Lexicrunch, we cannot justify the assumption of "omniscient properties." Sometimes the tests needed to discriminate among classes lie beyond the program's grasp, such as in the case of the rule of section 3.2.3 that words borrowed directly from Latin take **-or**. In addition, some morphological distinctions may simply be totally arbitrary, and not expressible in terms of abstract properties. For instance, what property of **ox**, etymological or otherwise, explains why it takes an **-en** plural, but **fox** and **ax** do not?

We therefore need an "escape hatch" to resort to when our properties fail us. If we run out of properties in ID4, we simply approximate the current corpus by a leaf node containing the largest class. Words in minority classes are memorized as

inclusions.

The revised algorithm is shown below. It returns two-part structures written as <X | Y>, where X is the decision tree and Y the list of inclusions. An assignment of the form <A | B> ← <X | Y> means A ← X and B ← Y.

```
procedure ID3.5 (set, partition, properties):

if (all elements of set are in the same class)
        or (properties is nil) then

        class ← the class in set with the most members
        inc ← a list of all objects not in class
        return <a leaf node containing class | inc>
else
        prop ← a property in properties
                which is chosen by some heuristic
        setY ← the elements of set that have prop
        setN ← the elements of set that do not have prop
        <treeY | incY> ← ID3.5(setY, partition,
                properties - prop)
        <treeN | incN> ← ID3.5(setN, partition,
                properties - prop)
        return <a tree whose root contains the property prop,
                whose Y-child is treeY, and whose N-child is
                treeN | the concatenation of incY and incN>
```

4.4.2.2.2. A relevance check

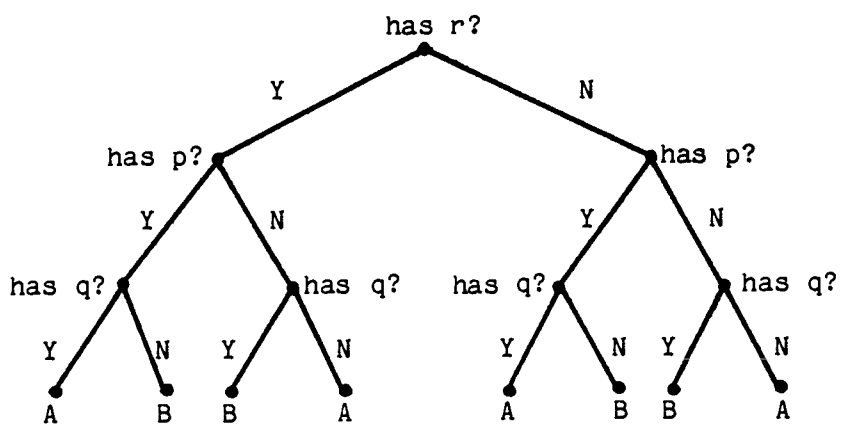
Our motivation for building a decision tree in the first place was to represent as compactly as we could a partition of objects into classes. It follows that tests in our tree that take up more storage than the examples they account for are counterproductive and irrelevant. Such tests nevertheless manage to find their way into our decision trees in two circumstances.

The first is when our heuristic for picking a test to split on backfires. Consider a hypothetical partition of English words into two classes, A and B, defined in terms of properties p and q

by

A = objects with both p and q, or neither
B = objects with p or q, but not both.

Let us say that there is a third property, **r**, which tests whether a word has an even number of letters. Assuming that all three properties bisect **A** and **B** approximately, the entropy heuristic will rate them more or less equally. Just to be perverse, suppose **r** comes out on top* Then the program will build a tree along the lines of



Here **r** is the quintessential irrelevant test: splitting on it contributes absolutely nothing to the task of discriminating between **A** and **B**. This is shown by the fact that the Y-subtree and N-subtree of the **r** node are identical -- we must apply the same classification function to words regardless of whether they have **r**.

The second situation that invites irrelevant tests is when the program simply does not have the property needed to tell certain classes apart. It then tends to claw and scratch its way

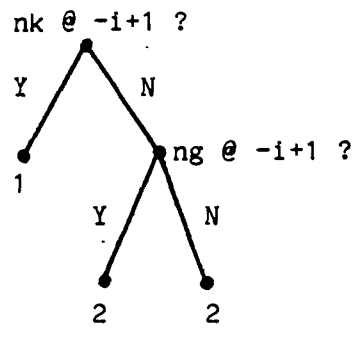
* This will happen if **p** and **q** split **A** and **B** into more equal halves than **r** does.

toward class definitions by applying numerous inappropriate tests.

For instance, say we are faced with these classes:

1. i @ -i <-> a @ -a
   {sink ring sing spring drink}
2. i @ -i <-> u @ -u
   {string cling fling sting stick slink}

and the available tests are **nk @ -i+1** and **ng @ -i+1**. These tests are certainly not sufficient to separate the classes -- indeed it is not clear that any test could do so -- but as long as the program has properties at its disposal, it will persist in growing its tree. It might come up with:



inclusions: ring, sing, spring, slink.

This solution requires a 5-node decision tree and 4 inclusions to remember 11 examples. Surely we would have been better off building a leaf node right at the start -- then there would be 1 node in the tree and 5 inclusions, as below.

•
2

inclusions: sink, ring, sing, spring, drink

Another reason for halting early in examples like that above is that "improving the fit of a rule to the training set (beyond a certain point) can make it perform worse on the rest of the

population" [O'Keefe 1983, p.479].

The customary cure for irrelevant tests -- at least those that arise in the second situation -- is to appeal to a stopping rule that says when classification has gone far enough (see, e.g., Sturt [1981]). The stopping rule evaluates the potential property-to-split-on using some heuristic; if the property fails, we build a leaf node. An example of a heuristic is to check that the entropy of the property is below a certain threshold.

One trouble with stopping rules is that defects in their heuristics tend to have drastic consequences. Sometimes it is necessary to split on a superficially "bad" property, as in the case of inherently disjunctive classes, when each individual property fits the class poorly. If the stopping rule vetoes the split, the decision subtree cannot be built. Stopping rules also magnify flaws in the property-selecting heuristic. Where the poor choice of a property led to redundant subtrees before (as in the example above with properties **p**, **q**, and **r**), once we have a stopping rule, the subtrees may not be created at all.

Corlett [1983] suggests a more robust remedy for irrelevant tests. He first "decompiles" the decision tree into a list of classification rules. There will be one rule for each root-to-leaf path in the tree: it will be of the form

> <u>if</u> an object has the properties corresponding to Y-branches on the path, and it lacks those corresponding to N-branches <u>then</u> the object is in the class given by the leaf node.

The next step is to go through each classification rule and try to

drop each of its conditions in turn. Unless the deletion of a condition would cause the rule to misclassify, it is discarded. [Corlett 1983, pp.139-40]

Applying Corlett's method to our **p**, **q**, and **r** example, we get eight classification rules:

if r and p and q then A
if r and p and not q then B
if r and not p and q then B
if r and not p and not q then A
if not r and p and q then A
if not r and p and not q then B
if not r and not p and q then B
if not r and not p and not q then A

The **r** condition from the first four rules and the **not r** condition from the last four can be deleted with impunity. The eight rules then collapse into the correct four.

Corlett actually addresses a more general problem than the one we stated: he deletes a test from only those rules in which it is extraneous, rather than in "all or none" of them. But Corlett's algorithm suffers from the concomitant problem that its output is a list, not a tree of rules: lists are far less efficient at classification. Moreover, he fails to throw away irrelevant tests that arise in our second situation, as long as they cover at least one example.

In Lexicrunch, a test is deemed irrelevant if its Y-child and N-child "compute the same function"; for if this is the case, then we may as well divert all the data to one of the subtrees -- it is redundant to have both. The way we tell whether one subtree alone suffices is to measure how it performs on the examples normally

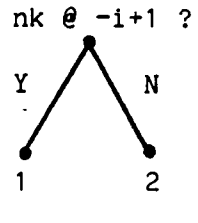
handled by the other subtree. If it does not require significantly more inclusions than the other subtree for any class, then it can be said to subsume the function of the other subtree. We may then eliminate the other subtree, as well as the parent node.

With this relevance check installed, our classification algorithm becomes:

```
procedure ID4 (set, partition, properties):

if (all elements of set are in the same class,
        or (properties is nil) then

        class ← the class in set with the most members
        inc ← a list of all objects not in class
        return <a leaf node containing class | inc>

else
        prop ← a property in properties
                which is chosen by some heuristic
        setY ← the elements of set that have prop
        setN ← the elements of set that do not have prop
        <treeY | incY> ← ID4(setY, partition,
                properties - prop)
        <treeN | incN> ← ID4(setN, partition,
                properties - prop)

        errY ← inclusions produced by classifying setN
                using treeY
        errN ← inclusions produced by classifying setY
                using treeN

        if for all classes c, not SIGNIF(no. of class c
                objects in errY - no. of class c objects
                in incY,  no. of objects in class c) then

                /* Y-subtree subsumes N-subtree */
                return <treeY | concatenation of incY
                        and errY>

        elseif for all classes c, not SIGNIF(no. of class c
                objects in errN - no. of class c objects
                in incN,  no. of objects in class c) then

                /* N-subtree subsumes Y-subtree */
                return <treeN | concatenation of incN
                        and errN>

        else
                /* neither subtree subsumes the other */
                return <a tree whose root contains the
                        property prop, whose Y-child is
                        treeY, and whose N-child is treeN |
                        the concatenation of incY and incN>
```

For the first example above, with properties **p**, **q**, and **r**, ID4 deletes the **r** test, because its Y-subtree and N-subtree behave exactly the same way. The whole tree is replaced by the Y-subtree.

In the second example, the **ng** test will be excised, since its subtrees are identical. This leaves the tree



The analysis for the **nk** test is as follows:

setY = {sink drink slink}
setN = {ring sing spring string cling fling sting stick}
incY = {slink}
incN = {ring sing spring}
errY = {string cling fling sting stick}
errN = {sink drink}

inclusions	setY		setN	
	class 1	class 2	class 1	class 2
treeY	0	1	0	5
treeN	3	0	2	0.

The Y-subtree does not subsume the N-subtree because the N-subtree is better for the class 2 data of **setN**, i.e. SIGNIF(5 - 0, 6). However, the N-subtree does subsume the Y-subtree, as it does not do significantly worse for class 1 -- not SIGNIF(3 - 0, 5) -- nor for class 2 -- not SIGNIF(0 - 1, 6). Thus we end up with the decision tree

•
2.

4.4.2.2.3. Ambiguous word classifications

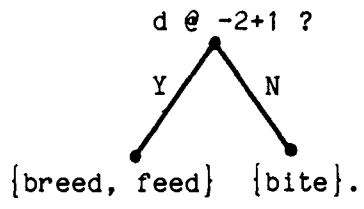
The classification problem in Lexicrunch is a departure from Quinlan's specification in that we are not given a strict

partition of words into classes: some words may be listed in multiple classes. The example given in section 4.4.1.4 was of **breed** in

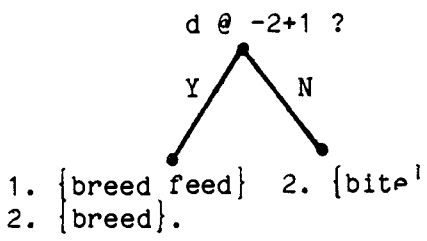
1. e @ -2 <-> ø @ -1
   {breed feed}
2. e @ +3 <-> ø @ +3
   {breed bite}

Let us consider how ID4 would cope with such an ambiguous classification.

Since ID4 classifies words by their properties, not by which class(es) they are in, we can forget for the moment that words are assigned to classes at all. Given the words **breed**, **feed**, and **bite**, and appropriate properties, ID4 might group the words as follows:



Now if we endeavor to account for "class structure," we will observe that all the copies of a word will aggregate together at the same leaf node, as there is no possible test that could split them up. Hence we can redraw the diagram above as



The rule for **breed** is thus computed normally. The only special provision we need to make is to cancel spare copies of a word at a leaf node if one of the copies is in the chosen class -- e.g. if we associate class 1 with the left leaf node above, we must strike **breed** from class 2, rather than listing it as an inclusion. If none of the copies of the word is in the chosen class, we will list it as an inclusion in all of its classes, and resolve the ambiguity later (see section 4.4.3).

4.4.3. Tidying up

Inclusions to our rule come from two sources: the ones returned by ID4, and those set aside earlier in rule compression in "insignificant classes." There may be words of either sort that are listed as inclusions in more than one class. We now assign them to just one of their classes.

We could make the assignments arbitrarily; but with the right arrangements, we may be able to get rid of a few classes. The task is to select as few classes as possible which still contain one copy of every word.

This is known as the minimum set cover problem. Unfortunately, it is NP-complete [Aho, Hopcroft, and Ullman 1974, p.378]. We will have to settle for a greedy heuristic. We start by taking "forced" classes, i.e. those referenced in the decision tree, and those that contain unique occurrences of elements. Then at each iteration, we pick the class that gives us the largest number of new elements. Ambiguously classified words are assigned to the first of their class that we select. Hopefully we will not

have to select every class.

Chapter 5

Performance of Lexicrunch

5.1. Examples

Lexicrunch was tested on three main examples: the past tense of verbs in English, the first person singular present indicative in Finnish, and the past participle in French. The rules produced by the program, after data entry and compression, are given in the appendices in **verbose** form.

The program displays rules in verbose form when trace mode is turned on during compression. Our rule from chapter 4 for the plural inflection could be written in verbose form as follows:

```
Decision tree:

if x @ -0-1 then [2.1]
  2.    {fox tax box prefix}
else [1.1]
  1.    {newt tulip apple elbow}
  2.    {switch}

Rule-defined classes:

1.      Ø @ -0 <-> s @ -1
        [1.1]:  newt tulip apple elbow

2.      Ø @ -0 <-> es @ -2
        [2.1]:  fox tax box prefix
        Inclusions:  switch[1.1]

Inclusion classes:

3.      Ø @ -0 <-> en @ -2
        Inclusions:  ox
```

The decision tree is printed first. Its leaf nodes are specified slightly differently than usual: instead of just supplying the index of a morphological process, we write the

index, a dot, and a small integer which identifies the leaf node, all enclosed in square brackets. Thus in the first line of the decision tree above, we have the leaf node [2.1], which references process 2. If there were subsequent leaf nodes containing process 2, they would be expressed as [2.2], [2.3], etc. Appearing after each leaf node is a tabulation of the words in <u>significant</u> classes which the node must handle. Node [1.1], for example, must handle **newt**, **tulip**, **apple**, and **elbow** from class 1, and **switch** from class 2.

Following the decision tree are the rule-defined classes, i.e. those classes referenced in the decision tree at least once. For each class, the domain is explicitly enumerated. First its rules in the decision tree are given, identified by their leaf nodes, along with the list of words they contribute to the class. Then come the words which are memorized as inclusions. Each word is accompanied by the leaf node that would misclassify it were it not listed as an inclusion. To illustrate, class 2 above is a rule-defined class with one rule in the decision tree, namely **if x @ -0-1 then** [2.1]. This rule contributes the words **fox**, **tax**, **box**, and **prefix** to the class. **Switch** is stored as an inclusion because it would otherwise be classified by the **else** [1.1] rule.

Finally, we have the inclusion classes. They have no rules in the decision tree; their domains are merely lists of inclusions. Words in the inclusion lists from <u>insignificant</u> classes are not printed with a corresponding leaf node. In our example, class 3 is the only inclusion class. It covers just one word, **ox**. Since **ox** is originally from an insignificant class, it

does not have a following leaf node.

5.1.1. English data

The English data consist of 127 verbs and their past tenses. An attempt was made to include at least four verbs from each irregular class, and a few more for the larger regular classes. The data and the rough class definitions come from Quirk and Greenbaum [1973, pp.26-35] and Strang [1962, pp.129-33].

Lexicrunch arrived at 14 rule-defined classes and 35 inclusion classes for the English example (see appendix A). Its rules for the regular classes are sensible; e.g. for class 1 we have, "Add **-ed** to a verb by default or if it ends in **-ay**." It is more difficult to judge other rules, such as, "Change the last **i** to an **ou** in verbs which have an **nd** after their last **i**," as grammar books rarely lay out such patterns. Pragmatically, the rules are sound, as they account for 84 words, memorizing only 17 of them. Three of the inclusions are **-ng** verbs whose **i** changes to an **a**, not a **u**. These are forgivable, as there is no obvious test that distinguishes the two sorts of **-ng** verbs. It also seems reasonable to memorize words like **hold**, as they bear little resemblance to other words of their class -- in this case, **blow**, **grow**, and **throw**.

The inclusion classes handle 43 examples, a suspiciously large number. But 39 of them -- words like **go** and **buy** -- are in classes containing at most two words. Such words are arguably "true anomalies" which must be learned by rote; it just happens that these are common in English. The remaining 4 words of the 43

-- **split, beat, cut,** and **hurt** -- are listed together in the class in which the past tense is the same as the present. This class could have been characterized approximately as "words that end in **-t**"; the reason that Lexicrunch missed this rule is discussed in section 5.2.3.

Rule compression for this example took approximately six minutes of CPU time, with the program running in compiled mode on the departmental VAX 11/750.

5.1.2. Finnish data

Inflected forms of Finnish verbs are derived from **verb stems**, abstract grammatical entities which do not exist in the language per se. Since our data are taken from Whitney [1956], we observe his convention of treating all stems as vowel stems, rather than saying that certain verbs have both a vowel and a consonant stem. To distinguish stems from "normal" words, we write them with a final hyphen [Whitney 1956, p.vi]. Note also that **ä** and **ö** are distinct letters in the Finnish alphabet [Whitney 1956, p.13]. They were represented as special characters in the Lexicrunch data, but appear normally here for the sake of readability.

The first person singular present indicative is derived quite regularly from the verb stem by (a) the addition of **-n**, and (b) any relevant consonant gradation in the stem (see section 2.3.3.2) [Whitney 1956, p.27]. Thus we should expect twelve classes in our rule: one for stems that do not change, and eleven for the eleven types of gradation. This is indeed what the program gets (see appendix B).

A correct rule for predicting which words are in, say, the **lt ~ ll** gradation class, is to check for the **-ltV-** ending, where **V** stands for any vowel. The relevant properties in Lexicrunch are **l @ -3-l** and **t @ -3-0.** We do not need to verify that the word has a hyphen as its last letter or a vowel before that, since all stems do. In fact, the program uses just the **l** property, because this works for most words -- it only misclassifies **alka-**. The program shies away from the conjunction of two properties because that would only explain one additional word.

The rule that a preceding **s** or **t** inhibits consonant gradation is partially expressed by leaf nodes [1.1] and [1.2], which state that stems in **-sty-** and **-sta-** belong in the non-gradating class. Words with related endings, e.g. **-tka-** in **jatka-**, are not numerous enough to warrant rules of their own, and thus are listed as inclusions.

There are also two rules in the decision tree that pick out stems in **-ata-** and **-ätä-**. These handle the so-called **amalgamating** verbs, which drop their **t** in the present tense. They are characterized more completely by a short (orthographically single) vowel plus a **-ta-** or **-tä-** ending. [Whitney 1956, p.28]

All together, the program requires 15 inclusions for a corpus of 81 words. The inclusions are distributed in little bunches, and so it would not be worthwhile to introduce rules to explain them.

A larger dataset would present Lexicrunch's rule with counterexamples, as several of its formulas are cheap extensional

equivalents. To see if additional data would indeed force the program to be more exact, an example with all 711 verbs from Whitney [1956] was run (see appendix C).

The program got the same twelve classes, but modified its decision tree. Where it used **l @ -3-l** to delineate the **lt ~ ll** class before, it now resorts to the proper conjunction,

(**l @ -3-l**) and (**t @ -3+0**),

as the extra condition is needed to handle six non-gradating counterexamples. It induces several unintuitive rules as well, e.g. that verbs in **-ty-** or **-tu-** do not gradate, but the rules cannot be faulted on practical grounds, as they produce very few inclusions.

Lexicrunch's solution incorporates 55 inclusions. As before, most of them can be accounted for by the fact that there are very few of any one kind. The first three inclusions to class 1, for example -- **kynsi-**, **pakinoi-**, and **luennoi-** -- are memorized because they are the only words in the corpus that have **n @ -3-l**, yet do not go in the **nt ~ nn** class. The program could conjoin another property, as it did for the **lt ~ ll** class, if the additional rule were, by its definition, justified.

One might intuitively expect the program to assign **-stä-** verbs to class 1 via a rule; instead, it stores all 12 as inclusions. The justification is that these 12 words are insignificant out of the 311 words in class 1, and thus they do not merit a special rule.

Class 4, the **t ~ d** gradation, also appears to have a hefty number of inclusions. Most of them originate from rule [3.2] -- that is, the program has difficulty distinguishing them from the amalgamating verbs of class 3. In order for it to tell them apart properly, it would need rules saying that verbs with long vowels, i.e. in **-VVtV-**, are in class 4; other verbs ending in **-ta-** or **-tä-** are in class 3. It would take a tremendous number of words to encourage Lexicrunch to build these rules.

The program compressed the shorter Finnish rule in 4.5 minutes of CPU time; the larger rule took 54 minutes.

5.1.3. French data

Le Nouveau Bescherelle [1966] sets out 82 paradigms for French verbs, along with rules which define their domains -- e.g. paradigm 57 illustrates how to conjugate verbs which end in **-eindre**, using the model of **peindre.** With respect to past participles, these reduce to 34 different paradigms.

To test Lexicrunch on the French data, four or more words (if available) from each of the 34 classes were entered -- 129 words in all. A note on the spelling of the words: the French acute accent (´), cedilla(,), circumflex(^), and umlaut(") are true accents, not parts of letters. These were accordingly represented as special characters <u>following</u> the letter they modify.

The program's rule (see appendix D) matches that in the **Bescherelle** quite closely. It has the same 34 classes, and the definitions are very nearly identical. As usual, when the program

could get away with underspecifying a class, it did so. Thus the class of **-er** verbs is characterized by **e @ -1-1.** Still, this approximation makes no errors for the corpus given. Moreover, a few classifications agreed exactly with **Bescherelle**'s; the test for the **-clure** verbs was

(re @ -2+0) and (clu @ -2-3).

Sometimes tests in the tree could be underspecified because all competing words had already been drawn off. This was the case for the rule for the **-enir** verbs,

(ten @ -2-3) or (ven @ -2-3).

All the **-enir** verbs are indeed derived from **venir** or **tenir**; the only problem with Lexicrunch's rule is that it does not mention the **-ir** ending. The rule works, nevertheless, because by the time the rule is given in the tree, only **-ir** verbs are left.

The exceptions in the program's rule also coincide with those given by **Bescherelle.** For example, **rire** and **sourire** appear as inclusions in class 13. **Bescherelle** does no better, listing them by themselves in paradigm 79. A few of the other inclusions, like **absoudre, dissoudre,** and **re´soudre** in class 29, are given in **Bescherelle** by a rule -- "ends in **-soudre**" -- but since the book provides only these three examples in the domain, Lexicrunch treats them as noise to be memorized.

The program's rule accounts for the 129 words using 32 inclusions. Because the rule is so close to the accepted formulation, it should work, practically unchanged, for all 8,000

verbs in the **Bescherelle.**

The rule was compressed in six minutes of CPU time.

5.2. Weaknesses

Lexicrunch suffers from a number of weaknesses, several of which were brought out in the test runs of section 5.1. These are described below, along with outlines of remedies.

5.2.1. Spelling is but a dim reflection of pronunciation

Since speech precedes writing both phylogenetically and ontogenetically, one might expect rules of word construction to be stated most naturally in terms of pronunciation. Orthography is a "second order effect," which was no doubt designed to map straightforwardly onto pronunciation, but falls short of the mark because of dialectical variation, language borrowing, and so forth.

This spells trouble for programs like Lexicrunch, as their vocabulary for expressing changes is orthographic. Sometimes it is beneficial to ignore phonetics -- then, e.g., we do not have to distinguish between the /d/ past tense morpheme in **raised** and the /t/ in **raced.** Just as often, though, it confuses the issue to deal with spelling. This is demonstrated by the English rule for forming the past tense by shortening /i/ to /ɛ/; it is realized in three different orthographic ways in **speed/sped, lead/led,** and **read/read.**

Lexicrunch could capture such trends directly if we were to run it on phonetic rather than graphemic representations of words. We are usually more interested, however, in recognizing and producing normal written words than phonetic transcriptions.

We can retain our graphemic representations but also take advantage of certain phonetic regularities by making a few extensions to the program. First, we would supply a phonetic transcription for each word, as well as the correspondence between phonemes and graphemes. Then we would modify the test-devising routine (see section 4.4.1.6) to take into account the phonetic environment of a change.

For data such as

```
 r     i     s     e
/r/   /ai/  /z/

 d     r     i     v     e
/d/   /r/   /ai/  /v/
```

etc.

we could then construct the past tense rule

```
if /r/ @ -3-1 then
   if /ai/ @ -3+0 then 1

1. i @ -3 <-> o @ -3.
```

This would be more correct than the orthographic test **ri @ -3-1** because it excludes words like **grill.**

We would have to alter the recognition module (see section 4.2) as well. Currently, in unapplying a rule, the program guesses which morphological process to uneffect, and then verifies

that it chose the right one. But now it may not know whether it made the right choice until it reaches a lexical entry, with the requisite phonological information. Thus the routine will have to delay its verification of certain guesses until the end.

For example, in recognizing **drove** and ***groll**, the routine would uneffect process 1 above, yielding **drive** and **grill**. So far, it cannot tell whether these forms have **/r/ @ -3-1** or **/ai/ @ -3+0**. It would then look up the words in the VERB rule, and find that **drive** does indeed have the appropriate /r/ and /ai/, and hence should take process 1. **Grill** lacks the /ai/, and so it was wrong to apply process 1.

5.2.2. Words with a multiplicity of forms

In the overview of the program (section 1.4), we assumed, with Kay [1983, p.94,102], that a morphological rule applied to a word gives only one result. Occasionally, however, there will be several answers.

In some cases, the multiplicity of forms corresponds to a multiplicity of meanings. **Hang**, for example, has the past tense **hung** in the sense of "suspended," but **hanged** for "executed." We could enforce the meaning difference by writing the words as **hang1** and **hang2**, but this is a bit cumbersome and contrived. It would not even work for examples like **appendix**, whose plurals -- **appendixes** and **appendices** -- are not separable on semantic grounds.

A more attractive way to protect the program from

multiplicity would be to hide it within special "**compound**" classes. For instance, we would put **appendix**, along with similar words, in just one such class, which we might represent as

1. Ø @ -0 <-> es @ -2
   OR
   x @ -1 <-> ces @ -3
   {appendix helix matrix radix}.

We would also have to update the generation routine to return a list of all possible answers.

5.2.3. Characterizing words that undergo no change

The routine that devises tests (see section 4.4.1.6) clearly cannot characterize any arbitrary class of words. A flagrant example of its inadequacy arises in the English data of appendix A, where the program fails to come up with any properties for the class

17. --
    {split beat cut hurt}

The problem is that the test-devising routine only looks for common strings around change sites; but the class above has no change sites -- it consists of words whose past tense is the same as the present. Nevertheless, a reasonable characterization exists; namely, **t @ -0-1**. The dental ending makes sense if we hypothesize a derivation like

split + -t → *split-t → split,

where the past tense **-t** is that in **burnt**, **learnt**, etc. [Matthews 1974, p.122].

The analysis above suggests that the change in our class is not null, but invisible, and it occurs at the end of the word. Lexicrunch, however, has no way of knowing this. We could make up for the program's ignorance by having it search for common strings at various places in the word -- including the beginning and the end -- whenever the class has a null list of changes.

5.2.4. Problems with ID4

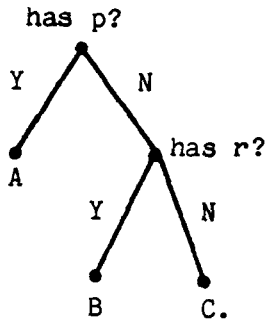
The ID4 algorithm is responsible for several of the program's performance problems. Some of them have to do with aspects of the implementation; others are more fundamental in that they challenge the use of decision trees.

5.2.4.1. Conjunctive "halfway houses"

Ordinary classes are easily teased apart with a decision tree, providing the apposite tests are available. As the tests are applied in turn, the associated classes "peel off" from the pack. Say there are three classes, **A**, **B**, and **C**, defined by

A = objects which have p
B = objects not in A which have r
C = objects not in A which do not have r

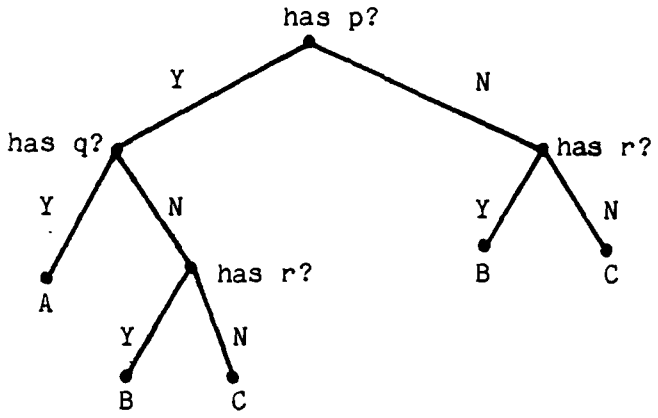
One decision tree to make the proper class assignments would be



Now, however, suppose that **A** is given by

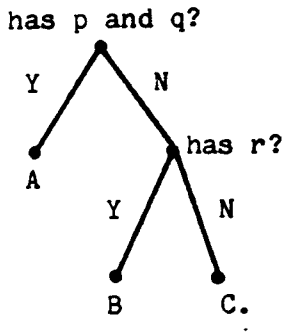
A = objects which have p and q.

To isolate **A**, we then need to apply tests **p** and **q** successively, i.e.



But as far as classes **B** and **C** are concerned, the intermediate test node for **q** is irrelevant. The decision subtree for **B** and **C** is the same, regardless of where in the **p and q** path it diverges from.

In general, a conjunction of **n** properties will necessitate **n** copies of the subtree that distinguishes among the remaining classes. To avoid this redundancy, we could "shrink" the conjunctive succession of properties down to one node. For the example above, we would be left with



This procedure is just like our relevance check (see section 4.4.2.2.2), except that we are conflating not a node's Y- and N-subtrees, but rather its N-subtree and the N-subtree of its Y-subtree.

But the iceberg extends still farther. Redundant subtrees can sprout arbitrarily far apart in the decision tree. If, for example, we defined A by

A = objects which have both p and q, or neither,

then the **not p and q** and **p and not q** subtrees would be the same. We should really compare every pair of subtrees to see if they compute the same function; and this could be prohibitively expensive. Moreover, the topology of the tree prevents the "shrinking" together of widespread nodes. We would have to replace the two subtrees with one common one, and install "links" from the roots of the original subtrees to the new subtree. The resulting structure would only be a decision tree in a very loose sense.

5.2.4.2. Disjunction versus random effects

To restrain Lexicrunch from characterizing wayward classes of

English words by

(a @ -0-1) or (b @ -0-1) or ... or (z @ -0-1),

we imposed the restriction that each disjunct cover at least **m** out of the **n** words in its class, where SIGNIF(m,n). Properties true of fewer examples are not recognized by the test-devising routine.

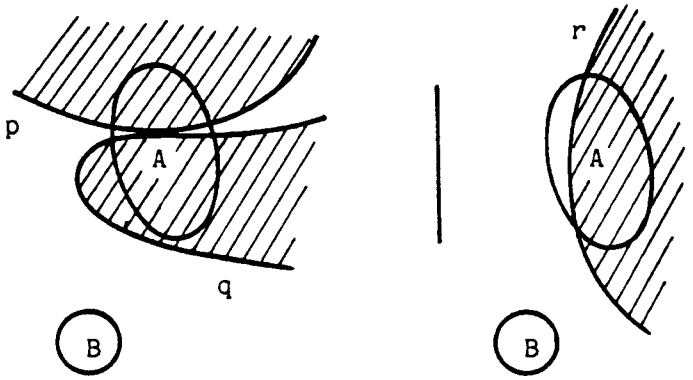
By the definition of SIGNIF (see section 4.4.1.5), this means that properties with 10 percent or less coverage will never be detected. However, some such properties are still "useful," e.g. **stä** @ -1-3 in class 1 of the Finnish example in appendix C.

To fix this, we could loosen the restriction on properties. If we are considering appending a character **c** onto the teststring **s**, we could insist only that **m**, the number of words with a **c**, is significant compared to **n**, the number of words with the string **s**, as opposed to the number of words in the whole class. Then, for instance, the **-stä-** property above would be accepted, so long as a significant proportion of the words in class 1 ended in **ä**, a significant proportion of words in **-ä** had a preceding **t**, and a significant proportion of **-tä** words had an s before that. The tests above for -a, -b, etc. should be judged irrelevant, but for reasons discussed in section 5.2.4.4, the later ones will not be.

We might also try to distinguish more precisely between "true" disjuncts and random effects by refining the SIGNIF function using principles of statistics, but this is beyond the scope of this paper.

5.2.4.3. Lexicrunch's aversion to disjunction

The entropy heuristic (see section 4.4.2.1) favors tests that split the corpus near class boundaries. It follows that disjunctive properties which cleave a class into halves will not be highly regarded. Consider the situation in the diagram:



The formula **p or q** characterizes **A** quite well. Nevertheless, of **p**, **q**, and **r**, it is the last that will be chosen, because it cuts closer to class boundaries than the others. After that, **p** and **q** may not be incorporated into the decision tree if they do not explain enough of the outstanding words.

The only certain way to avoid such unfortunate choices of properties is to "look ahead" to see whether a given test will ultimately lead to a clean, effective characterization. This, however, entails building the rest of the decision tree every time we evaluate a test. It is not clear how to improve the test-selection heuristic without incurring this massive cost.

5.2.4.4. Fragmentation due to irrelevant tests

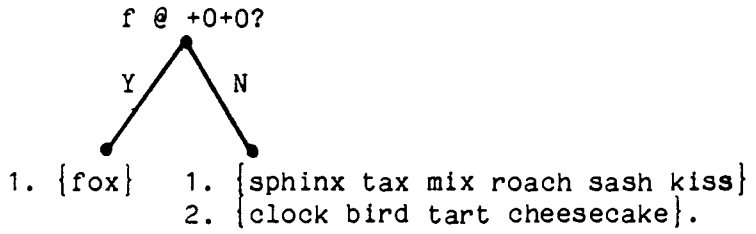
Irrelevant tests should logically never be put in the decision tree. In Lexicrunch, however, it is convenient to build

the whole tree first, including irrelevant test nodes, and to expunge the unwanted tests afterwards. The temporary irrelevant tests tend to break up the classes into incoherent bits, resulting in two deleterious effects.

First, while a whole class may manifest a certain trend clearly, its parts, after fragmentation, may be too small to deserve to be individually characterized. Thus the original trend is lost. Take the following classes from the plural rule for English nouns:

1. {fox sphinx tax mix roach sash kiss}
2. {clock bird tart cheesecake}

and suppose that we have two properties to choose from, **x @ -0-1** and the bizarre **f @ +0+0.** The program will split on the latter, as it slices closer to class boundaries; it will later be thrown out as irrelevant. Meanwhile, we get the tree



The Y-subtree is finished. At the N-subtree, we can apply **x @ -0-1**, but now it only explains three words, and so it too will be considered irrelevant. If we had applied it to the original class 1, it would have been judged worthwhile.

The second problem is the flip-side of the first: tests that cover only a tiny proportion of the words in a class may mistakenly be taken as significant for a fragment of the class.

This circumstance was alluded to in section 5.2.4.2 when we tried to characterize a class of English words by applying 26 irrelevant tests to it:

(a @ -0-1) or (b @ -0-1) or ... or (z @ -0-1).

The first tests we used would correctly be identified as irrelevant, since they would presumably cover an insignificant fraction of the words in the class. These early tests would cull progressively more words out of the class, though, until eventually some of the later tests above would explain a reasonable fraction of the remainder. Such tests would not have stood a chance had they been applied ahead of the other irrelevant tests.

ID4 could be modified to eliminate the side effects caused by irrelevant tests. Instead of picking the one best property to split on, we would rank the properties from best to worst. We would then split on a property and complete the decision tree. If the property proved irrelevant, we would iterate for the next property. Irrelevant tests would thus not affect the construction of the rest of the tree.

This solution smacks of the nondeterminism we were worried about at the end of section 5.2.4.3 -- although here we only have to "look ahead" when we hit an irrelevant test, which should be relatively seldom. But it is still likely that the resulting algorithm would be insufferably slow.

5.2.5. Redundancy at the rule level

While Lexicrunch is reasonably adept at purging redundancies from morphological data, new ones emerge in its rules. These suggest that the program should be extended to capture regularities across rules with meta-rules. We could continue in this manner, with a meta-meta-level, ad infinitum.

5.2.5.1. Disjoining conditions

Given enough English past tenses, the program would eventually derive a rule including a set of highly similar tests:

```
if ay @ -0-2 then 1
if ey @ -0-2 then 1
if iy @ -0-2 then 1
if oy @ -0-2 then 1
if uy @ -0-2 then 1

1. Ø @ -0 <-> ed @ -2.
```

Since these tests differ only in one grapheme of their teststrings, it would make sense to replace that grapheme by a variable. We could then condense the five tests into one:

```
if Vy @ -0-2 then 1
where V ε {a, e, i, o, u}.
```

It pays to merge rules of the form

```
if p then change1
if q then change1      into      if p or q then change1
```

if the disjunction can be expressed moderately compactly. It certainly seems reasonable to combine rules when their tests have the same position and offset, as in the example above. One could

envisage a routine that would "factor out" variables by comparing the two teststrings with a string-to-string repair algorithm like that in section 4.3.3. Recurring variables like the one for vowels would only have to be stored once, resulting in additional space savings.

5.2.5.2. Unifying morphological processes

Lexicrunch may also produce series of related morphological processes, which can profitably be combined if we introduce a new notational mechanism. Consider the changes for consonant doubling in the rule for English past tenses:

```
Ø @ -0 <-> bed @ -3
Ø @ -0 <-> red @ -3
Ø @ -0 <-> ned @ -3
```

These are identical, except for one grapheme in their newstrings. They can be summarized by a single more general expression:

```
Ø @ -0 <-> Xed @ -3
where X @ -1.
```

The "X @ -1" means that **X** stands for the last character of the uninflected word. Had **X** appeared in the oldstring of the change, it would refer to the last character of the inflected word.

A routine to assimilate pairs of changes could look for variables in the same way as the routine sketched in section 5.2.5.1. To figure out an afield that describes where a variable is located, we could simply try all possibilities, as the search space is not that large.

5.2.5.3. Isolating the components of rules

The third person plural present indicative in Finnish is formed by adding **-vat** to the verb stem [Whitney 1956, p.27]. The vowel in this suffix must harmonize with the stem, and so it sometimes appears as **ä**. The stem is also susceptible to consonant gradation.

Lexicrunch would nominally create 24 classes for this rule, each one performing one type of gradation -- there are 12 choices -- and adding one of the two realizations of the suffix -- 2 choices. If, however, we were to analyze the rule into two steps, consonant gradation and vowel harmony, we would need a total of just 14 classes. This wasteful "cross-product effect" will result whenever two or more independent alternations take place within a rule.

Tell-tale evidence of the cross-product effect is when there are pairs of rules of the form

```
if p and q then change1, change2
if p and r then change1, change3.
```

The repeated portion should be extracted, giving two sub-rules:

```
if p then change1          if q then change2
                           if r then change3.
```

It would be feasible but decidedly non-trivial to write an algorithm to decompose rules in this way.

Once a rule has been broken down into independent components, it can be replaced by "sub-rule calls" to them. The same sub-

rule, e.g. that of consonant gradation, could be called by many morphological rules.

5.2.5.4. Shared decision trees

The various inflections on a given part of speech in a language often partition the words in similar ways. Take the first person singular and first person plural for French verbs. The former splits verbs into an **-er** class and an **-ir** class, among others -- these take the **-e** and **-is** endings, respectively. The latter also distinguishes those classes, suffixing them with **-ons** and **-issons**. Grammar texts pick up on this homogeneity by using the same set of verb categories for every inflection. [Bescherelle 1966, p.24]

Lexicrunch would produce two essentially isomorphic decision trees for these French rules. It could therefore economize on storage by recognizing the redundancy and throwing one copy away. One tree may make distinctions that are absent in the other; these would have to be preserved in the resulting tree. Pairs of trees with many partially overlapping classes should not be merged, as their combination would be unacceptably bulky.

Two rules that have had their trees merged would use the same numbering scheme for their classes, and would have pointers to a common tree. We might also wish to put inclusions in the shared area if they are classified the same way for both rules.

5.3. Evaluation

By and large, the troubles cited above are not severe, and can be resolved adequately by the methods sketched. A few, however, are more serious "matters for further investigation."

Computational linguists have in the past put up with the fact that rules of orthography tend to be messier than their phonological counterparts. In any case, we showed how Lexicrunch could be made to describe at least the domains of rules in phonological terms.

The introduction of "compound" classes would enable the program to cope with our second problem, that of words which give more than one answer when a transformation is applied.

The next problem we looked at was that of characterizing words that undergo no observable change. Here it was sufficient to do a "blanket search" for properties; a guided search would necessitate deep linguistic analysis.

The difficulties with ID4 can be ignored to the extent that they arise infrequently, and they introduce redundancy, not inaccuracy into rules. Some of them, however, point up significant flaws in the algorithm. In particular, the repetition of subtrees (see section 5.2.4.1) implies that the decision tree may be an inappropriate representation; a decision "graph," or merely an ordered list of leaf nodes might be better. Lexicrunch's problems with disjunctive rules and, to a lesser degree, irrelevant tests, indicate inherent limitations on ID4's abilities. There is thus room for improvement in the area of

classification algorithms.

The most pressing problem with Lexicrunch is redundancy at the level of its rules. This springs up when there are similar sets of tests, morphological processes, or decision trees, or when there are rules with isolable components. The program needs a meta-level if it is to achieve a reasonable standard of compression when presented with many rules of a language. A rough design of such an extension was given in section 5.2.5.

Chapter 6

Conclusion

The Lexicrunch program abstracts morphological rules from sets of examples. Its primary goal is to compact the data; this it does by capturing many instances of a recurring pattern with a single specification of it.

The linguistic contribution of the work reported here is the Lexicrunch formalism for morphological rules. Its chief merit is its extensibility, deriving from the fully general notions of morphological processes and property-defined domains. The particular implementation can always be adapted within the basic framework to handle new types of alternations.

The program induces a rule by (1) splitting the given corpus of words into classes according to the morphological process they undergo, and then (2) characterizing each class of words by their properties. Step 2 is performed by the ID4 program module, which is based on Quinlan's ID3 procedure. The enhancements to ID3 constitute the other main contribution of this work. They were as follows: to make a provision for exceptions to classification rules; to implement a filter that rejects irrelevant properties; and to let words be assigned initially to more than one class.

The program was tested on examples from English, Finnish, and French: it appeared to pick up most of the significant trends in the data.

Several extensions could be made to the program. The most important would be a meta-level which would enable the program to

recognize regularities in its rules, as it already does for its data.

Appendix A
English example

Decision tree:

```
if ak @ -1-2 then [12.1]
  2.    {bake}
  12.   {take mistake forsake shake}
elseif w @ -o+1 then [13.1]
  13.   {blow grow know throw}
elseif nd @ -i+1 then [5.1]
  5.    {wind grind find bind}
elseif ay @ -0-2 then [1.1]
  1.    {pray stay spray play relay}
elseif y @ -1+0 then [4.1]
  1.    {toy}
  4.    {hurry cry pry carry dally}
elseif nk @ -i+1 then [6.1]
  6.    {drink shrink sink stink}
elseif d @ -1+0 then
  if e @ -2+0 then [9.1]
    9.  {bleed breed feed speed}
  else [16.1]
    1.  {end}
    13. {hold}
    16. {build bend lend spend}
elseif e @ -2+0 then [14.1]
  14.   {creep sleep sweep feel}
elseif p @ -0-1 then [3.1]
  1.    {jump}
  3.    {strip hop strap map}
elseif i @ -i+0 then
  if ng @ -3+1 then [11.1]
    6.  {spring sing ring}
    11. {sting string fling cling}
  else [7.1]
    2.  {like}
    5.  {fight}
    6.  {sit give}
    7.  {drive arise write stride ride win}
    10. {bite hide}
    11. {stick}
    17. {split}
elseif e @ -0-1 then [2.1]
  2.    {hate name race manage}
  15.   {lose}
elseif ear @ -0-3 then [8.1]
  2.    {hear}
  8.    {bear swear tear wear}
else [1.2]
  1.    {clean touch wash work call walk rest}
  8.    {steal break speak}
  17.   {beat cut hurt}
```

Rule-defined classes:

1. Ø @ -0 <-> ed @ -2
   [1.1]: pray stay spray play relay
   [1.2]: clean touch wash work call walk rest
   Inclusions: toy[4.1] end[16.1] jump[3.1]

2. Ø @ -0 <-> d @ -1
   [2.1]: hate name race manage
   Inclusions: bake[12.1] like[7.1] hear[8.1]

3. Ø @ -0 <-> ped @ -3
   [3.1]: strip hop strap map

4. y @ -1 <-> ied @ -3
   [4.1]: hurry cry pry carry dally

5. i @ -i <-> ou @ -ou
   [5.1]: wind grind find bind
   Inclusions: fight[7.1]

6. i @ -i <-> a @ -a
   [6.1]: drink shrink sink stink
   Inclusions: spring[11.1] sing[11.1] ring[11.1] sit[7.1] , give[7.1]

7. i @ -i <-> o @ -o
   [7.1]: drive arise write stride ride win

8. ea @ -3 <-> o @ -3
   Ø @ -0 <-> e @ -1
   [8.1]: bear swear tear wear
   Inclusions: steal[1.2] break[1.2] speak[1.2]

9. e @ -2 <-> Ø @ -1
   [9.1]: bleed breed feed speed

11. i @ -3 <-> u @ -3
    [11.1]: sting string fling cling
    Inclusions: stick[7.1]

12. a @ -3 <-> oo @ -3
    e @ -1 <-> Ø @ -0
    [12.1]: take mistake forsake shake

13. o @ -o <-> e @ -e
    [13.1]: blow grow know throw
    Inclusions: hold[16.1]

14. e @ -2 <-> Ø @ -2
    Ø @ -0 <-> t @ -1
    [14.1]: creep sleep sweep feel

16. d @ -1 <-> t @ -1
    [16.1]: build bend lend spend

Inclusion classes:

10. e @ +3 <-> Ø @ +3
    Inclusions: bite[7.1] hide[7.1]

15. e @ +3 <-> Ø @ +3
    Ø @ -0 <-> t @ -1
    Inclusions: lose[2.1]

17. --
    Inclusions: split[7.1] beat[1.2] cut[1.2] hurt[1.2]

18. Ø @ -0 <-> ted @ -3
    Inclusions: permit

19. Ø @ -0 <-> red @ -3
    Inclusions: bar

20. e @ +1 <-> o @ +1
    Inclusions: get

21. e @ -1 <-> Ø @ -0
    Inclusions: slide

22. ea @ +1 <-> o @ +1
    Inclusions: weave

23. ee @ +2 <-> o @ +2
    Inclusions: freeze

24. o @ +3 <-> Ø @ +3
    Inclusions: choose shoot

25. a @ -a <-> e @ -e
    Inclusions: draw fall

26. a @ +1 <-> u @ +1
    Inclusions: hang

27. u @ +1 <-> a @ +1
    Inclusions: run

28. ay @ -2 <-> ew @ -2
    Inclusions: slay

29. o @ +1 <-> a @ +1
    Inclusions: come

30. e @ +0 <-> Ø @ +0
    Ø @ -0 <-> e @ -1
    Inclusions: eat

31. ie @ +1 <-> ay @ +1
    Inclusions: lie

32. ee @ +1 <-> aw @ +1
Inclusions: see

33. i @ -3 <-> uc @ -3
e @ -1 <-> Ø @ -0
Inclusions: strike

34. y @ -1 <-> id @ -2
Inclusions: say lay

35. el @ +1 <-> o @ +1
Ø @ -0 <-> d @ -1
Inclusions: sell tell

36. o @ +1 <-> id @ +1
Inclusions: do

37. be @ +0 <-> was @ +0
Inclusions: be

38. Ø @ +1 <-> o @ +1
y @ -1 <-> ght @ -3
Inclusions: buy

39. tc @ +2 <-> ug @ +2
Ø @ -0 <-> t @ -1
Inclusions: catch

40. in @ +2 <-> ou @ +2
Ø @ -0 <-> ht @ -2
Inclusions: bring

41. ave @ +2 <-> ft @ +2
Inclusions: leave

42. an @ +2 <-> oo @ +2
Inclusions: stand

43. e @ +1 <-> Ø @ +1
c @ -2 <-> ug @ -4
Ø @ -0 <-> t @ -1
Inclusions: teach

44. eek @ +1 <-> ought @ +1
Inclusions: seek

45. ink @ +2 <-> ought @ +2
Inclusions: think

46. k @ -2 <-> d @ -2
Inclusions: make

47. ve @ -2 <-> d @ -1
Inclusions: have

48. go @ +0 <-> went @ +0
    Inclusions: go

49. y @ -1 <-> ew @ -2
    Inclusions: fly

Appendix B
Finnish example

Decision tree:

```
if p @ -3+0 then [8.1]
  8.    {raapi- viipy- sopi- kylpe- uupu-}
elseif n @ -3-1 then [3.1]
  3.    {myöntä- kokoontu- synty- tunte- käänty- anta-}
elseif l @ -3-1 then [5.1]
  5.    {puhalta- uskalta- viheltä- kieltä- ylty-}
  6.    {alka-}
elseif r @ -3-1 then [4.1]
  4.    {takertu- siirty- murta- kerto- ymmärtä-}
  6.    {kirku-}
elseif k @ -3+0 then [6.1]
  1.    {jatka- kätke- kisko- laske-}
  6.    {nukku- leikki- rikko- haukku- liikku- luke- määki- ,
        näky- aiko-}
elseif ätä @ -1-3 then [2.1]
  2.    {määrätä- värjätä- pelkätä- herätä-}
elseif t @ -3+0 then
  if sty @ -1-3 then [1.1]
    1.  {sivisty- hämmästy- ilmesty- edisty-}
  elseif sta @ -1-3 then [1.2]
    1.  {harrasta- loista- nosta- osta- purista-}
  elseif ata @ -1-3 then [2.2]
    2.  {kelpata- huomata- seurata- pelata- saarnata-}
  elseif t @ -3-1 then [2.3]
    2.  {koputta- peittä- puuttu- moitti- petty-}
  else [7.1]
    1.  {pistä-}
    2.  {hävitä-}
    7.  {huuta- kiirehti- tahto- avautu- pysähty- tietä-}
else [1.3]
  1.    {ui- seiso- ammu- ole- mene- saa- tule-}
```

Rule-defined classes:

```
1.      - @ -1 <-> n @ -1
        [1.1]:  sivisty- hämmästy- ilmesty- edisty-
        [1.2]:  harrasta- loista- nosta- osta- purista-
        [1.3]:  ui- seiso- ammu- ole- mene- saa- tule-
        Inclusions:  jatka-[6.1] kätke-[6.1] kisko-[6.1] laske-,
          [6.1] pistä-[7.1]

2.      t @ -3 <-> ∅ @ -2
        - @ -1 <-> n @ -1
        [2.1]:  määrätä- värjätä- pelkätä- herätä-
        [2.2]:  kelpata- huomata- seurata- pelata- saarnata-
        [2.3]:  koputta- peittä- puuttu- moitti- petty-
        Inclusions:  hävitä-[7.1]

3.      t @ -3 <-> n @ -3
        - @ -1 <-> n @ -1
        [3.1]:  myöntä- kokoontu- synty- tunte- käänty- anta-
```

4. t @ -3 <-> r @ -3
   - @ -1 <-> n @ -1
   [4.1]: takertu- siirty- murta- kerto- ymmärtä-

5. t @ -3 <-> l @ -3
   - @ -1 <-> n @ -1
   [5.1]: puhalta- uskalta- viheltä- kieltä- ylty-

6. k @ -3 <-> Ø @ -2
   - @ -1 <-> n @ -1
   [6.1]: nukku- leikki- rikko- haukku- liikku- luke- , määki- näky- aiko-
   Inclusions: alka-[5.1] kirku-[4.1]

7. t @ -3 <-> d @ -3
   - @ -1 <-> n @ -1
   [7.1]: huuta- kiirehti- tahto- avautu- pysähty- tietä-

8. p @ -3 <-> v @ -3
   - @ -1 <-> n @ -1
   [8.1]: raapi- viipy- sopi- kylpe- uupu-

Inclusion classes:

9. k @ -3 <-> g @ -3
   - @ -1 <-> n @ -1
   Inclusions: tinki-

10. p @ +2 <-> m @ +2
    - @ -1 <-> n @ -1
    Inclusions: ampu-

11. k @ -3 <-> j @ -3
    - @ -1 <-> n @ -1
    Inclusions: sulke- kulke-

12. p @ +2 <-> Ø @ +2
    - @ -1 <-> n @ -1
    Inclusions: oppi- loppu- leppy-

Appendix C
Extended Finnish Example

Decision tree:

```
if p @ -3+0 then [7.1]
  7.    {luopu- leipo- uupu- hiipi- kylpe- vaipu- sopi- ,
        saapu- repi- viipy- raapi-}
elseif n @ -3-1 then [6.1]
  1.    {kynsi- pakinoi- luennoi-}
  6.    {anta- käänty- rakenta- tunte- synty- kiinty- kyntä- ,
        lentä- työntä- kanta- kokoontu- myöntä- kääntä- ,
        paranta- äänt ä- hämmenty- sekaantu- vähenty- tyhjenty- ,
        väänt ä- nääny- asenta-}
elseif k @ -3+0 then [2.1]
  1.    {kätke- koske- tutki- käske- itke- laske- usko- ,
        jatku- kisko- jatka-}
  2.    {uhku- pyrki- purka- puke- loiko- koke- halko- tako- ,
        ruokki- pyyhki- hehku- aiko- kirku- hankki- välkky- ,
        liikku- haukku- alka- rikko- leikki- teke- näke- näky- ,
        nukku- määki- luke- hake-}
elseif r @ -3-1 then [5.1]
  1.    {ympäröi- kärsi- epäröi-}
  5.    {visertä- ymmärtä- kerto- kiertä- kumarta- kumartu- ,
        murta- saarta- siirty- sorta- takertu- piirtä- sinertä- ,
        uurta-}
elseif l @ -3-1 then
  if t @ -3+0 then [8.1]
    8.  {puhalta- uskalta- viheltä- kieltä- sukelta- ,
        vaelta- ylty- kiiltä-}
  else [1.1]
    1.  {lyö- luo- tulvi- salli- valvo- solmi-}
elseif yty @ -1-3 then [4.1]
  4.    {heittäyty- järjestäyty- keräyty- kiteyty- käyttäyty- ,
        kieltäyty- pistäyty- selviyty- tyyty- löyty-}
elseif sta @ -1-3 then [1.2]
  1.    {puhdista- varasta- sisusta- pelasta- korista- ,
        kiinnosta- aavista- tarkasta- muodosta- tunnusta- ,
        omista- reunusta- perusta- koukista- ripusta- painosta- ,
        luista- karista- kuulosta- kalasta- pudista- muista- ,
        matkusta- kirkasta- ratsasta- paista- valmista- ,
        rakasta- purista- osta- harrasta- nosta- loista-}
elseif t @ -3-1 then
  if t @ -3+0 then [3.1]
    3.  {vaikutta- totutta- raoitta- pakotta- yllättä- ,
        voitta- viivyttä- varoitta- tarkoitta- sijoitta- ,
        myöhästyttä- tuotta- saavutta- rauhoitta- noudatta- ,
        nauratta- lähättä- liitty- ilahdutta- upotta- ,
        sytyttä- pystyttä- pettä- nyökäyttä- liikutta- ,
        väittä- vahingoitta- suoritta- saatta- jännittä- ,
        jännitty- vapautta- vakuutta- raivostutta- peitty- ,
        lämmittä- kunnioitta- kauhistutta- häämöttä- välittä- ,
        selvittä- nyökkäyttä- muistutta- yrittä- luotta- ,
        lepuutta- kehittä- hyödyttä- harjoitta- tyynnyttä- ,
        tukahdutta- tomutta- sähköttä- suuttu- poltta- ,
        lahjoitta- kylvettä- kehoitta- katta- hävittä- ,
```

```
      harmitta- väsyttä- taivutta- syyttä- petty- moitti- ,
      kohotta- kengittä- jyskyttä- joudutta- ilmoitta- ,
      huvitta- tottu- painatta- nimittä- muuttu- koetta- ,
      keittä- iljettä- arvelutta- toimitta- selittä- ,
      levittä- kuljetta- kannatta- kammotta- janotta- ,
      inhotta- esittä- erotta- ehdotta- asetta- syöttä- ,
      sammutta- puuttu- lähettä- lopetta- lohdutta- ,
      johdatta- aloitta- värisyttä- virvoitta- säilyttä- ,
      ruskotta- osoitta- muutta- menettä- kirjoitta- ,
      kiinnittä- keskeyttä- hämmästyttä- huomautta- ,
      toivotta- tarttu- sattu- riittä- päätty- näyttä- ,
      kolkutta- kiittä- jättä- irroitta- hengittä- heittä- ,
      asettu- tuijotta- sidotta- pudotta- nautti- käyttä- ,
      viettä- otta- toteutta- soitta- päättä- odotta- ,
      mietti- laitta- herättä- autta- täyttä- peittä- ,
      opetta- koputta- koitta-}
  else [1.3]
    1. {viitsi- hallitse- vangitse- patentoi- havaitse- ,
        harkitse- kutsu- etsi- häiritse- vallitse- valitse- ,
        tuo- tarvitse- merkitse- katso-}
elseif t @ -3+0 then
  if u @ -3-1 then [4.2]
    3. {haluta-}
    4. {huuta- avautu- istuutu- nouta- palautu- pukeutu- ,
        tunkeutu- souta- ilmoittautu- kuto- naulautu- joutu- ,
        toteutu- laittautu- paneutu- riutu- mukautu- sopeutu- ,
        suhtautu- uhrautu- laskeutu- hautautu- ojentautu- ,
        sulkeutu- purkautu- sulautu- painautu-}
  elseif h @ -3-1 then [4.3]
    4. {tohti- pohti- leimahta- läjähtä- jyrähtä- tipahta- ,
        tervehti- johtu- kuihtu- paahta- kiihty- ryhty- ,
        hypähtä- haihtu- ehti- värähtä- mahta- huolehti- ,
        purjehti- käännähtä- perehty- istahta- vaihta- ,
        purskahta- huoahta- erehty- vivahta- seisahtu- lähte- ,
        katsahta- johta- huudahta- unohta- naurahta- välähtä- ,
        tapahtu- pysähty- tahto- kiirehti-}
  elseif y- @ -3+1 then [1.4]
    1. {edisty- ilmesty- hämmästy- sivisty- rypisty- ,
        myöhästy- kyyristy- ällisty- kyllästy- hengästy- ,
        menesty-}
    4. {jäätу-}
  elseif u- @ -3+1 then [1.5]
    1. {punastu- pahastu- alistu- pelastu- valistu- ,
        rakastu- kummastu- kiinnostu- huolestu- harmistu- ,
        tutustu- muodostu- valmistu- ihastu- sairastu- ,
        suostu- muistu- kastu- maistu- poistu- kuvastu- ,
        onnistu- astu- istu-}
    4. {koitu- kaatu- katu-}
  else [3.2]
    1. {niistä- päällystä- yhdistä- ryöstä- kestä- siisti- ,
        veistä- rypistä- estä- virkistä- pyydystä- pistä- ,
        järjestä-}
    3. {tilata- hakkata- varata- uhkata- sieppata- takata- ,
        selvitä- punata- kuiskata- katketa- kampata- uppota- ,
        kiusata- suuntata- salata- kehrätä- nyökkätä- kuvata- ,
        kohtata- lepätä- korvata- erota- rupeta- kultata- ,
```

kokota- kiipetä- huokata- halketa- viittata- vajota- ,
tarjota- poikketa- nojata- laahata- hyppätä- osata- ,
lykkätä- kerätä- arvata- lupata- kohota- katota- ,
kaipata- tuhlata- siunata- paiskata- lakkata- ,
kuivata- korjata- tapata- siivota- vertata- vastata- ,
sahata- putota- palata- pakkata- naulata- määrätä- ,
maalata- kelpata- hävitä- huomata- leikkata- seurata- ,
värjätä- pelkätä- pelata- avata- saarnata- makata- ,
herätä-}
4. {säätä- nito- hoita- sietä- kiitä- kaata- vaati- ,
sito- pyytä- löytä- kyte- pitä- vetä- tietä-}
else [1.6]
1. {riitele- lämpene- lämmittele- louhi- lavertele- ,
hangoittele- valu- tarjoile- säännöstele- läpäise- ,
kohtele- istuskele- väittele- suunittele- palele- ,
onnittele- nuole- lepäile- kykene- kastele- jyrise- ,
juhli- haukottele- epäile- aukene- suosi- suo- sivele- ,
rankaise- otaksu- lekottele- kysele- kirpaise- kipaise- ,
julkaise- hyväksy- piirtele- painu- niele- mutise- ,
menettele- kilpaile- huoli- tottele- pakene- narise- ,
vilise- vanhene- unelmoi- rohkaise- paina- lupaile- ,
kuvittele- katkaise- vähene- lyhene- liho- kuori- ,
hupene- hio- harvene- vieri- säily- sure- puhele- osu- ,
kuiva- ilmaise- väsy- valkene- valaise- urheile- ,
tiedustele- sääli- seulo- salamoi- rukoile- nytkähtele- ,
lähene- kuljeksi- korjaile- kalastele- hyräile- ,
hoitele- alene- työskentele- suurene- selkene- rohkene- ,
lueskele- hälvene- aukaise- arvele- arvaile- voi- veny- ,
vapise- vaikene- tupakoi- toimi- supattele- sipaise- ,
pysy- pese- oikaise- lausu- kulu- kikattele- ,
keskustele- jaksa- varjele- tuoksu- suojele- pure- ,
palvele- maksa- lörpöttele- luule- kuule- kalise- ,
määri- aja- tavoittele- tanssi- suutele- surise- ,
samoile- pääse- näyttele- matkustele- luistele- ,
lakaise- kitise- juokse- hiljene- visertele- vilkaise- ,
toivo- puhaltele- pimene- pane- mumise- mietiskele- ,
kumartele- kaiva- jää- juttele- ihmettele- tarkastele- ,
myy- kuole- vie- sula- suhise- solise- risteile- ,
nielaise- kuohu- kohise- kimaltele- katsele- aterioi- ,
asu- viljele- ui- käy- kävele- kysy- kuuntele- kuulu- ,
kasva- hymyile- tule- syö- soutele- soi- seiso- sano- ,
saa- repäise- puhu- pala- opiskele- nouse- naura- mene- ,
laula- juo- elä- ammu- ajattele- ole-}

Rule-defined classes:

1. - @ -1 <-> n @ -1
[1.1]: lyö- luo- tulvi- salli- valvo- solmi-
[1.2]: puhdista- varasta- sisusta- pelasta- korista- ,
kiinnosta- aavista- tarkasta- muodosta- tunnusta- ,
omista- reunusta- perusta- koukista- ripusta- painosta- ,
luista- karista- kuulosta- kalasta- pudista- muista- ,
matkusta- kirkasta- ratsasta- paista- valmista- ,
rakasta- purista- osta- harrasta- nosta- loista-
[1.3]: viitsi- hallitse- vangitse- patentoi- havaitse- ,
harkitse- kutsu- etsi- häiritse- vallitse- valitse- ,

tuo- tarvitse- merkitse- katso-
[1.4]: edisty- ilmesty- hämmästy- sivisty- rypisty- , myöhästy- kyyristy- ällisty- kyllästy- hengästy- , menesty-
[1.5]: punastu- pahastu- alistu- pelastu- valistu- , rakastu- kummastu- kiinnostu- huolestu- harmistu- , tutustu- muodostu- valmistu- ihastu- sairastu- suostu- , muistu- kastu- maistu- poistu- kuvastu- onnistu- astu- , istu-
[1.6]: riitele- lämpene- lämmittele- louhi- lavertele- , hangoittele- valu- tarjoile- säännöstele- läpäise- , kohtele- istuskele- väittele- suunittele- palele- , onnittele- nuole- lepäile- kykene- kastele- jyrise- , juhli- haukottele- epäile- aukene- suosi- suo- sivele- , rankaise- otaksu- lekottele- kysele- kirpaise- kipaise- , julkaise- hyväksy- piirtele- painu- niele- mutise- , menettele- kilpaile- huoli- tottele- pakene- narise- , vilise- vanhene- unelmoi- rohkaise- paina- lupaile- , kuvittele- katkaise- vähene- lyhene- liho- kuori- , hupene- hio- harvene- vieri- säily- sure- puhele- osu- , kuiva- ilmaise- väsy- valkene- valaise- urheile- , tiedustele- sääli- seulo- salamoi- rukoile- nytkähtele- , lähene- kuljeksi- korjaile- kalastele- hyräile- , hoitele- alene- työskentele- suurene- selkene- rohkene- , lueskele- hälvene- aukaise- arvele- arvaile- voi- veny- , vapise- vaikene- tupakoi- toimi- supattele- sipaise- , pysy- pese- oikaise- lausu- kulu- kikattele- , keskustele- jaksa- varjele- tuoksu- suojele- pure- , palvele- maksa- lörpöttele- luule- kuule- kalise- , hääri- aja- tavoittele- tanssi- suutele- surise- , samoile- pääse- näyttele- matkustele- luistele- , lakaise- kitise- juokse- hiljene- visertele- vilkaise- , toivo- puhaltele- pimene- pane- mumise- mietiskele- , kumartele- kaiva- jää- juttele- ihmettele- tarkastele- , myy- kuole- vie- sula- suhise- solise- risteile- , nielaise- kuohu- kohise- kimaltele- katsele- aterioi- , asu- viljele- ui- käy- kävele- kysy- kuuntele- kuulu- , kasva- hymyile- tule- syö- soutele- soi- seiso- sano- , saa- repäise- puhu- pala- opiskele- nouse- naura- mene- , laula- juo- elä- ammu- ajattele- ole-
Inclusions: kynsi-[6.1] pakinoi-[6.1] luennoi-[6.1] , kätke-[2.1] koske-[2.1] tutki-[2.1] käske-[2.1] itke-, [2.1] laske-[2.1] usko-[2.1] jatku-[2.1] kisko-[2.1] , jatka-[2.1] ympäröi-[5.1] kärsi-[5.1] epäröi-[5.1] , niistä-[3.2] päällystä-[3.2] yhdistä-[3.2] ryöstä-[3.2] , kestä-[3.2] siisti-[3.2] veistä-[3.2] rypistä-[3.2] , estä-[3.2] virkistä-[3.2] pyydystä-[3.2] pistä-[3.2] , järjestä-[3.2]

2. k @ -3 <-> ø @ -2
- @ -1 <-> n @ -1
[2.1]: uhku- pyrki- purka- puke- loiko- koke- halko- , tako- ruokki- pyyhki- hehku- aiko- kirku- hankki- , välkky- liikku- haukku- alka- rikko- leikki- teke- , näke- näky- nukku- määki- luke- hake-

3. t @ -3 <-> Ø @ -2
- @ -1 <-> n @ -1
[3.1]: vaikutta- totutta- raoitta- pakotta- yllättä- ,
voitta- viivyttä- varoitta- tarkoitta- sijoitta- ,
myöhästyttä- tuotta- saavutta- rauhoitta- noudatta- ,
nauratta- lähättä- liitty- ilahdutta- upotta- sytyttä- ,
pystyttä- pettä- nyökäyttä- liikutta- väittä- ,
vahingoitta- suoritta- saatta- jännittä- jännitty- ,
vapautta- vakuutta- raivostutta- peitty- lämmittä- ,
kunnioitta- kauhistutta- häämöttä- välittä- selvittä- ,
nyökkäyttä- muistutta- yrittä- luotta- lepuutta- ,
kehittä- hyödyttä- harjoitta- tyynnyttä- tukahdutta- ,
tomutta- sähköttä- suuttu- poltta- lahjoitta- kylvettä- ,
kehoitta- katta- hävittä- harmitta- väsyttä- taivutta- ,
syyttä- petty- moitti- kohotta- kengittä- jyskyttä- ,
joudutta- ilmoitta- huvitta- tottu- painatta- nimittä- ,
muuttu- koetta- keittä- iljettä- arvelutta- toimitta- ,
selittä- levittä- kuljetta- kannatta- kammotta- ,
janotta- inhotta- esittä- erotta- ehdotta- asetta- ,
syöttä- sammutta- puuttu- lähettä- lopetta- lohdutta- ,
johdatta- aloitta- värisyttä- virvoitta- säilyttä- ,
ruskotta- osoitta- muutta- menettä- kirjoitta- ,
kiinnittä- keskeyttä- hämmästyttä- huomautta- toivotta- ,
tarttu- sattu- riittä- päätty- näyttä- kolkutta- ,
kiittä- jättä- irroitta- hengittä- heittä- asettu- ,
tuijotta- sidotta- pudotta- nautti- käyttä- viettä- ,
otta- toteutta- soitta- päättä- odotta- mietti- laitta- ,
herättä- autta- täyttä- peittä- opetta- koputta- koitta-
[3.2]: tilata- hakkata- varata- uhkata- sieppata- ,
takata- selvitä- punata- kuiskata- katketa- kampata- ,
uppota- kiusata- suuntata- salata- kehrätä- nyökkätä- ,
kuvata- kohtata- lepätä- korvata- erota- rupeta- ,
kultata- kokota- kiipetä- huokata- halketa- viittata- ,
vajota- tarjota- poikketa- nojata- laahata- hyppätä- ,
osata- lykkätä- kerätä- arvata- lupata- kohota- katota- ,
kaipata- tuhlata- siunata- paiskata- lakkata- kuivata- ,
korjata- tapata- siivota- vertata- vastata- sahata- ,
putota- palata- pakkata- naulata- määrätä- maalata- ,
kelpata- hävitä- huomata- leikkata- seurata- värjätä- ,
pelkätä- pelata- avata- saarnata- makata- herätä-
Inclusions: haluta-[4.2]

4. t @ -3 <-> d @ -3
- @ -1 <-> n @ -1
[4.1]: heittäyty- järjestäyty- keräyty- kiteyty- ,
käyttäyty- kieltäyty- pistäyty- selviyty- tyyty- löyty-
[4.2]: huuta- avautu- istuutu- nouta- palautu- pukeutu- ,
tunkeutu- souta- ilmoittautu- kuto- naulautu- joutu- ,
toteutu- laittautu- paneutu- riutu- mukautu- sopeutu- ,
suhtautu- uhrautu- laskeutu- hautautu- ojentautu- ,
sulkeutu- purkautu- sulautu- painautu-
[4.3]: tohti- pohti- leimahta- läjähtä- jyrähtä- ,
tipahta- tervehti- johtu- kuihtu- paahta- kiihty- ,
ryhty- hypähtä- haihtu- ehti- värähtä- mahta- huolehti- ,
purjehti- käännähtä- perehty- istahta- vaihta- ,
purskahta- huoahta- erehty- vivahta- seisahtu- lähte- ,

katsahta- johta- huudahta- unohta- naurahta- välähtä- ,
tapahtu- pysähty- tahto- kiirehti-
Inclusions: jääty-[1.4] koitu-[1.5] kaatu-[1.5] katu-,
[1.5] säätä-[3.2] nito-[3.2] hoita-[3.2] sietä-[3.2] ,
kiitä-[3.2] kaata-[3.2] vaati-[3.2] sito-[3.2] pyytä-,
[3.2] löytä-[3.2] kyte-[3.2] pitä-[3.2] vetä-[3.2] ,
tietä-[3.2]

5. t @ -3 <-> r @ -3
- @ -1 <-> n @ -1
[5.1]: visertä- ymmärtä- kerto- kiertä- kumarta- ,
kumartu- murta- saarta- siirty- sorta- takertu- piirtä- ,
sinertä- uurta-

6. t @ -3 <-> n @ -3
- @ -1 <-> n @ -1
[6.1]: anta- käänty- rakenta- tunte- synty- kiinty- ,
kyntä- lentä- työntä- kanta- kokoontu- myöntä- kääntä- ,
paranta- ääntä- hämmenty- sekaantu- vähenty- tyhjenty- ,
vääntä- näänty- asenta-

7. p @ -3 <-> v @ -3
- @ -1 <-> n @ -1
[7.1]: luopu- leipo- uupu- hiipi- kylpe- vaipu- sopi- ,
saapu- repi- viipy- raapi-

8. t @ -3 <-> l @ -3
- @ -1 <-> n @ -1
[8.1]: puhalta- uskalta- viheltä- kieltä- sukelta- ,
vaelta- ylty- kiiltä-

Inclusion classes:

9. p @ +2 <-> Ø @ +2
- @ -1 <-> n @ -1
Inclusions: oppi- loppu- leppy-

10. k @ -3 <-> j @ -3
- @ -1 <-> n @ -1
Inclusions: kulke- sulke-

11. p @ +2 <-> m @ +2
- @ -1 <-> n @ -1
Inclusions: ampu-

12. k @ -3 <-> g @ -3
- @ -1 <-> n @ -1
Inclusions: tinki-

Appendix D
French Example

Decision tree:

```
if croi^ @ -3-5 then [12.1]
  12.   {recroi^tre de´croi^tre accroi^tre croi^tre}
elseif ai^tre @ -6+0 then [11.1]
  11.   {connai^tre reconnai^tre parai^tre apparai^tre ,
        disparai^tre transparai^tre pai^tre}
elseif re @ -2+0 then
  if in @ -3-2 then [9.1]
    8.  {vaincre}
    9.  {peindre atteindre ceindre feindre geindre teindre ,
        joindre oindre poindre craindre}
  elseif clu @ -2-3 then [13.1]
    13. {reclure occlure inclure exclure conclure}
  elseif i @ -2-1 then [10.1]
    10. {traire dire e´crire confire suffire cuire ,
        conduire produire re´duire se´duire construire luire ,
        nuire}
    13. {rire sourire}
  else [8.1]
    8.  {rendre pendre tendre vendre perdre mordre battre}
elseif oir @ -3+0 then
  if c @ -5-1 then [6.1]
    6.  {percevoir de´cevoir concevoir apercevoir recevoir}
  else [7.1]
    7.  {choir vouloir valoir falloir pourvoir revoir voir}
elseif que´ri @ -1-6 then [4.1]
  4.    {reque´rir reconque´rir que´rir enque´rir conque´rir ,
        acque´rir}
elseif e @ -1-1 then [1.1]
  1.    {abaisser ble´ser coudoyer de´sirer entonner ha^bler ,
        nettoyer rainer rosser terroriser}
elseif ouvrir @ -0-6 then [5.1]
  5.    {rouvrir entrouvrir ouvrir de´couvrir couvrir}
elseif ten @ -2-3 then [3.1]
  3.    {tenir contenir retenir de´tenir}
elseif ven @ -2-3 then [3.2]
  3.    {revenir obvenir devenir venir}
else [2.1]
  2.    {terrir rosir rajeunir nordir hai"r envahir ,
        de´sobe´ir cre´pir blettir abasourdir}
  3.    {secourir courir ve^tir}
  5.    {souffrir offrir}
```

Rule-defined classes:

```
1.      r @ -1 <-> ´ @ -1
        [1.1]:  abaisser ble´ser coudoyer de´sirer entonner ,
          ha^bler nettoyer rainer rosser terroriser

2.      r @ -1 <-> Ø @ -0
        [2.1]:  terrir rosir rajeunir nordir hai"r envahir ,
          de´sobe´ir cre´pir blettir abasourdir
```

3. ir @ -2 <-> u @ -1
[3.1]: tenir contenir retenir de´tenir
[3.2]: revenir obvenir devenir venir
Inclusions: secourir[2.1] courir[2.1] ve^tir[2.1]

4. e´r @ -5 <-> Ø @ -2
r @ -1 <-> s @ -1
[4.1]: reque´rir reconque´rir que´rir enque´rir , conque´rir acque´rir

5. ri @ -3 <-> e @ -3
Ø @ -0 <-> t @ -1
[5.1]: rouvrir entrouvrir ouvrir de´couvrir couvrir
Inclusions: souffrir[2.1] offrir[2.1]

6. evoir @ -5 <-> ,u @ -2
[6.1]: percevoir de´cevoir concevoir apercevoir recevoir

7. oir @ -3 <-> u @ -1
[7.1]: choir vouloir valoir falloir pourvoir revoir voir

8. re @ -2 <-> u @ -1
[8.1]: rendre pendre tendre vendre perdre mordre battre
Inclusions: vaincre[9.1]

9. dre @ -3 <-> t @ -1
[9.1]: peindre atteindre ceindre feindre geindre teindre , joindre oindre poindre craindre

10. re @ -2 <-> t @ -1
[10.1]: traire dire e´crire confire suffire cuire , conduire produire re´duire se´duire construire luire , nuire

11. ai^tre @ -6 <-> u @ -1
[11.1]: connai^tre reconnai^tre parai^tre apparai^tre , disparai^tre transparai^tre pai^tre

12. oi @ -6 <-> u @ -2
tre @ -3 <-> Ø @ -0
[12.1]: recroi^tre de´croi^tre accroi^tre croi^tre

13. re @ -2 <-> Ø @ -0
[13.1]: reclure occlure inclure exclure conclure
Inclusions: rire[10.1] sourire[10.1]

Inclusion classes:

14. avoir @ +0 <-> eu @ +0
Inclusions: avoir

15. ^ @ +1 <-> ´ @ +1
r @ -2 <-> Ø @ -2
Ø @ -0 <-> ´ @ -1
Inclusions: e^tre

16. u @ +2 <-> Ø @ +2
ir @ -2 <-> t @ -1
Inclusions: mourir

17. avoir @ +1 <-> u @ +1
Inclusions: savoir

18. evoir @ +1 <-> u^ @ +1
Inclusions: devoir

19. o @ +1 <-> Ø @ +1
voir @ +3 <-> Ø @ +2
Inclusions: pouvoir

20. o @ +1 <-> Ø @ +1
voir @ +3 <-> ^ @ +2
Inclusions: mouvoir

21. e @ +2 <-> Ø @ +2
voir @ -4 <-> Ø @ -0
Inclusions: pleuvoir

22. eo @ -4 <-> Ø @ -2
r @ -1 <-> s @ -1
Inclusions: asseoir surseoir

23. endre @ +2 <-> is @ +2
Inclusions: prendre

24. ettre @ +1 <-> is @ +1
Inclusions: mettre

25. aire @ +2 <-> u @ +2
Inclusions: plaire

26. ai^tr @ +1 <-> Ø @ +1
Ø @ -0 <-> ´ @ -1
Inclusions: nai^tre

27. oire @ +2 <-> u @ +2
Inclusions: croire

28. re @ -2 <-> s @ -1
Inclusions: clore

29. dre @ -3 <-> s @ -1
Inclusions: absoudre dissoudre re´soudre

30. dre @ -3 <-> su @ -2
Inclusions: coudre

31. dre @ -3 <-> lu @ -2
Inclusions: moudre

32. re @ -2 <-> i @ -1
Inclusions: suivre

33. ivr @ +1 <-> Ø @ +1
Ø @ -0 <-> ´cu @ -3
Inclusions: vivre

34. ire @ +1 <-> u @ +1
Inclusions: lire

Bibliography

Aho, A., Hopcroft, J., and Ullman, J. The Design and Analysis of Computer Algorithms. Massachusetts: Addison-Wesley Publishing Company, 1974.

Alam, Yukiko Sasaki. "A Two-level Morphological Analysis of Japanese," Texas Linguistic Forum, 22(1983), 229-252.

Backhouse, Roland C. Syntax of Programming Languages: Theory and Practice. London: Prentice-Hall International, 1979.

Berko, Jean. "The Child's Learning of English Morphology," Word: Journal of the Linguistic Circle of New York, 14(1958), 150-177.

Bundy, Silver, and Plummer. "An Analytical Comparison of Some Rule Learning Programs." Department of Artificial Intelligence, Edinburgh University, Research Paper no. 215, 1984.

Cohen, P.R. and Feigenbaum, E.A. (editors). Handbook of Artificial Intelligence, volume 3. London: Pitman Books Limited, 1982.

Corlett, R.A. "Explaining Induced Decision Trees," in Expert Systems, 1983. Pp. 136-42.

Hunt,. Marin, and Stone. Experiments in Induction. New York: Academic Press, 1966.

Jespersen, Otto. A Modern English Grammar on Historical Principles. Part VI: Morphology. London: George Allen and Unwin Limited, 1946.

Karttunen, Lauri. "KIMMO: A General Morphological Processor," Texas Linguistic Forum, 22(1983), 165-86.

Karttunen, L. and Wittenburg, K. "A Two-level Analysis of English," Texas Linguistic Forum, 22(1983), 217-228.

Kay, M. "When Meta-rules are not Meta-rules," in Automatic Natural Language Parsing, eds. Karen Sparck Jones and Yorick Wilks. Chichester: Ellis Harwood Limited, 1983. Pp. 94-116.

Khan, Robert. "A Two-level Morphological Analysis of Rumanian," Texas Linguistic Forum, 22(1983), 253-270.

Koskenniemi, Kimmo. Two-level Morphology: A General Computational Model for Word-form Recognition and Production. Department of General Linguistics, University of Helsinki, publication no. 11, 1983a.

Koskenniemi, Kimmo. "Two-level Model for Morphological Analysis," in Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 1983b, volume 2. Pp. 683-5.

Lehtinen, Meri. The Basic Course in Finnish. Indiana: Indiana University Press, 1967.

Lun, S. "A Two-level Morphological Analysis of French," Texas Linguistic Forum, 22(1983), 271-8.

Matthews, P.H. Morphology: An Introduction to the Theory of Word-structure. Cambridge: Cambridge University Press, 1974.

Le Nouveau Bescherelle: L'Art de Conjuguer. Paris: Libraire Hatier, 1966.

O'Keefe, Richard. "AS165 as a Learning Program," in The Mecho Notebook, volume 2. Department of Artificial Intelligence, Edinburgh University, unpublished. Note 156, 4 December 1982.

O'Keefe, Richard. "Concept Formation from Very Large Training Sets," in Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 1983, volume 1. Pp. 479-81.

Quinlan, J. R. "Discovering Rules by Induction from Large Collections of Examples," in Expert Systems in the Micro-electronic Age, ed. D. Michie. Edinburgh: Edinburgh University Press, 1979. Pp. 168-201.

Quirk, R. and Greenbaum, S. A University Grammar of English. London: Longman Group Limited, 1973.

Ritchie, G.D. and Pulman, S.G. "A Dictionary and Morphological Analyser for English Language Processing Systems." Unpublished research proposal, Edinburgh University, 1983.

Stein, Jess (editor). The Random House College Dictionary: Revised Edition. New York: Random House, Inc., 1975.

Strang, Barbara. Modern English Structure. London: Edward Arnold Publishers Limited, 1962.

Sturt, E. "An Algorithm to Construct a Discriminant Function in Fortran for Categorical Data," Applied Statistics, 30(1981), 313-25.

Whitney, Arthur H. Finnish. Suffolk: Hodder and Stoughton (Teach Yourself Books), 1956.

Winograd, Terry. Language as a Cognitive Process. Volume 1: Syntax. California: Addison-Wesley Publishing Company, 1983.