Morphological Parsing with Two-Level Framework

Simo Härkönen

University of Tampere
Department of Computing Sciences
Master's Thesis
December 2006

University of Tampere
Department of Computing Sciences
Simo Härkönen: Morphological Parsing with Two-Level Framework
Master's Thesis, 51 pages, 4 appendix pages
December 2006

This Master's thesis describes the two-level model, which is one way to do morphological parsing. The basic components of the two-level model are the simplified deep form of the language, and the rules which define the relation between the imaginary deep form and the actual words.

The thesis synthesizes the theoretical background in formal languages and same-length relations and uses them to describe the two-level model formally. It introduces a novel way to compile rules, which is simpler than the previous methods.

The empirical part investigates how determinizing the transducer affects parsing performance and transducer size.

Key words and phrases: computational morphology, two-level model, same-length relation, morphological parsing, determinization

Table of Contents

1. Overview 3
2. Informal Description of the Two-Level Model 5
    2.1 Language Model in Two-Level Morphology 5
    2.2 The Deep Form and the Surface Form 5
    2.3 Two-Level Rules 6
    2.4 Rules and Transducers 7
    2.5 Lexicon and Feasible Pairs 9
    2.6 Parsing in Action 9
3. Formal Language Machinery 11
    3.1 Basic Notations 11
    3.2 Regular Expressions and Regular Relations 11
    3.3 Finite State Machines and Transducers 13
    3.5 Same-Length Languages 15
4. Two-Level Rules and Their Application 20
    4.1 The Rules 20
    4.2 Rules as Transducers 22
5. Compiling the Rules 24
    5.1 Previous Compilation Algorithms 24
    5.2 Compiling Context Restriction Rules 25
        5.2.1 Compiling Single Rules 25
        5.2.2 Compiling Batch Rules 27
    5.3 Compiling Surface Coercion Rules 29
    5.4 Implementing the Algorithms 30
6. The Lexicon 31
    6.1 Continuation Classes 31
    6.2 The Deep Form State Machines 34
    6.3 Removing Semiempty Transitions 39
7. Input Determinization 42
    7.1 The Determinization Algorithm 42
    7.2 Problems with Determinization 44
8. The Effects of Determinization on Size and Performance 46
    8.1 Test Software and Data 46
    8.2 Determinization and the Size of the Transducer 47
    8.3 Determinization and Parsing Performance 48
9. Conclusions 50
Appendix 1. Notation Sheet 52
Appendix 2. Sample of the Lexicon Format 53

Appendix 3. Sample of the Rule Format..54

1. Overview

This thesis describes a morphological parsing algorithm based on two-level model. Two-level model was introduced over 20 years ago by Koskenniemi [1984], and it has inspired numerous publications and applications ever since.

Chapter 2 gives the reader an intuitive idea of morphological parsing and the two-level model. Chapter 3 introduces definitions, which are needed when formalizing these ideas. Chapter 4 describes the parsing algorithm, and Chapter 5 presents the rule compiler. Chapter 6 describes one way to organize the vocabulary in two-level morphology. Chapters 7 and 8 discuss the theoretical and empirical performance of the parsing algorithms.

Morphological parsing is a computation, which takes as input a derived form of a word. It outputs the dictionary form of the word and information about the derivation. If several derivations from different words can produce the same surface form, all possible parsings are listed. Figure 1.A illustrates this.

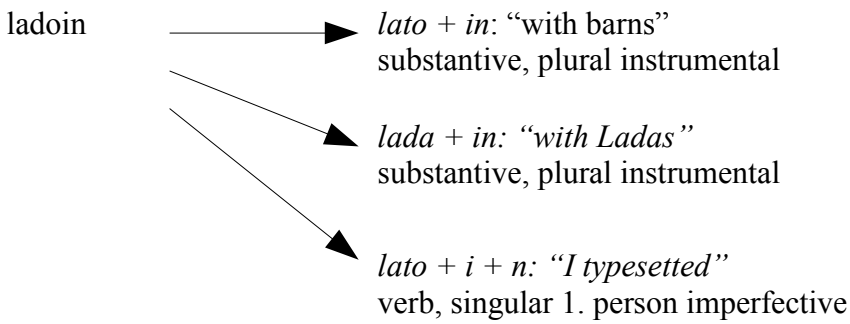

Figure 1.A. Morphological parsing from a word to a cohort.

A proper natural-language parsing pipeline takes a sentence as an input. The output is a tree, where the words are mapped to leaf nodes, while the syntactic structure of the sentence is mapped to the structure of the tree.

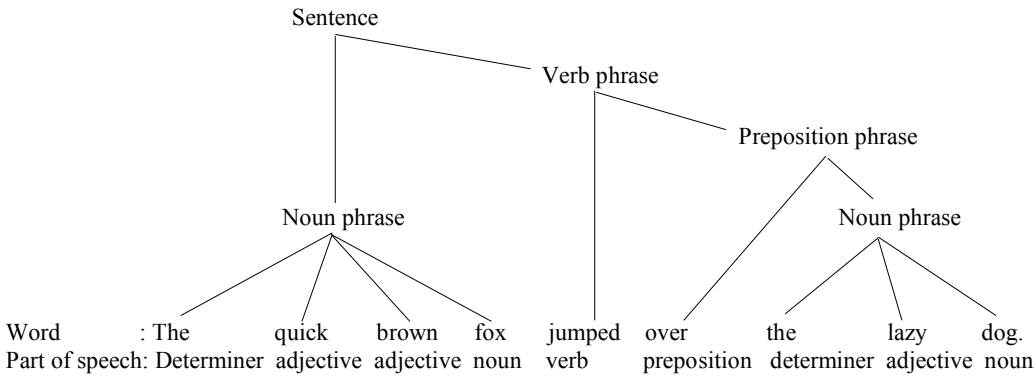

Figure 1.B. Parse tree and the parts of speech for the sentence "The quick brown fox jumped over the lazy dog."

This kind of pipeline has two or three stages. The morphological analysis is the first one. The second phase, called *parts-of-speech tagging*, aims to find out the classes of the words. The third phase constructs the parsing tree. The words are the leaf nodes of the parsing tree, while the inner nodes represent various syntactic structures. The arcs represent relationships: for example a noun phrase being the subject of the sentence. Figure 1.B shows an example of a parsing tree.

In languages with little inflection, like English, parts-of-speech tagging is combined with syntactic analysis. In highly inflected languages like Finnish, morphological analysis also disambiguates the parts of speech for most words.

The most obvious parsing method is to collect a list of acceptable words. Ispell, a free Unix spell checking program, uses this method. Ispell also supports affix files. Affix files specify, which postfixes you can add to a certain class of words. Ispell doesn't support analysis of derivation – it only recognizes ungrammatical words and proposes grammatical ones.

This thesis synthesizes the development of the two-level model after Koskenniemi [1983]. Chapter 5 introduces a novel way to compile rules. The method is in many aspects similar to one introduced by Kaplan and Kay [1994], but simpler. Chapter 6 provides the first mathematical formulation of continuation classes, which is the way vocabulary is expressed in the two-level model. The final chapters provide new information about the interaction between parsing performance and the grammatical structure of Finnish.

2. Informal Description of the Two-Level Model

2.1 Language Model in Two-Level Morphology

Two-level model assumes that morphology is fundamentally concatenative, but various phonological and notational aspects make it seem more complex. Two-level model dinstiguishes between the *surface form* – the form we see in everyday texts – and an artificial *deep from*. In the deep form, parsing and production are a matter of concatenation.

Two-level grammar consists of a set of rules, which define the correspondence between the deep form and the surface form. The rules are *bidirectional*: They define both how to parse the deep form when you know the surface form, and also how to produce the surface form given the deep form.

In the end of parsing, the deep form is converted into *tag form*. Tag form consists of a set of tags that denote the meaning and inflection of the word. Table 2.A illustrates the differences between the surface, the deep and the tag forms.

Word	Surface form	Deep form	Tag form
cats	cats	cat+s	cat, plural
spies	spies	spy+s	spy, plural

Table 2.A. Surface, deep and tag form. Note that deriving a plural in deep form is concatenative, while surface form has nonconcatenative irregularities.

2.2 The Deep Form and the Surface Form

Table 2.B illustrates the Finnish consonant gradation. The letter ”p” is realized as ”v”, when certain inflectional affixes are added to the stem.

Surface Form	Deep Form	In English	Case
papu	paPu	bean	Nominative
pavussa	paPu+$ssa	in a bean	Inessive
pavun	paPu+$n	of a bean	Genitive
papua	paPu+a	bean as an object	Partitive
papujen	paPu+Ien	of beans	Plural genitive

Table 2.B. Deep form and surface form in consonant gradation.

In the deep form, the special character P is used to denote a potentially gradated consonant. We say that P may be *realized* in the surface form as either p or v. There are also non-letter characters in the deep form. ”+” denotes the boundary between a stem and an affix. Whether the gradation happens or not depends on the affix, and ”$” is used as the trigger for gradation.

There is a specific *correspondence relation* between the deep form and the surface form. For each character in the surface form there is a unique counterpart in the deep form. However, some characters in the deep form realize in the surface as nothing. Figure 2.C illustrates this.

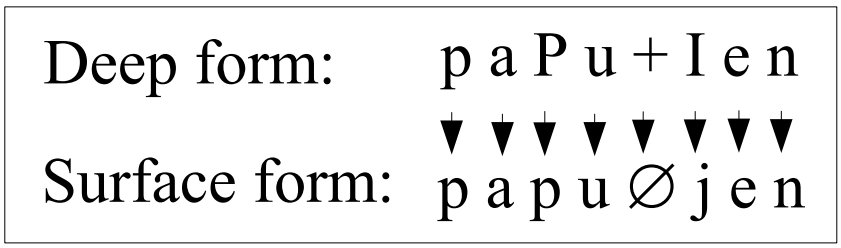

Figure 2.C. Correspondence relation between the deep form and the surface form.

2.3 Two-Level Rules

The rules define the correspondence between the characters in the surface form and the deep form. They may also be intepreted as filters which deny illegal correspondences, or as transformations between the deep form and the surface form. Figure 2.D shows the components of a two-level rule.

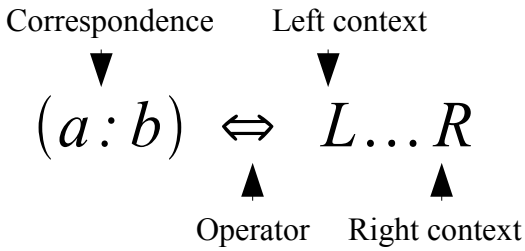

Figure 2.D. The components of a two-level rule. Here "a" is a deep form character and "b" is a surface form character.

Two-level rules either allow or deny the occurence of a correspondence in a specific context. The left context and the right context are regular expressions, except that they are matched against both the deep form and the surface form. There are two main types of rules, and the operator indicates the type.

⇒ : A *context restriction* rule indicates that the given correspondence is possible only in the given context. For example, when modeling irregular plural inflection as in spy-spies, $(y:i) \Rightarrow \ldots (+,e)(s,s)$ indicates that deep y may become a surface i only in the presence of a plural ending.

⇐ : A *surface coercion* rule restricts the possible realizations of a deep form character in the given context. If the correspondence in the rule is $(d:S)$ and the deep form contains a letter d, then it must realize in the surface as some $s \in S$. For example, $(y,i) \Leftarrow \ldots (+, S_{any})$ indictes that y always realizes as i when followed by a

plural marker; that is, it coerces deep y not to realize as surface y in this particular context.

The following additional rule types can be easily translated into context restriction and surface coercion rules.

$\Leftrightarrow$: The effect of a *composite rule* is the same as the effect of a context restriction rule and an identical surface coercion rule combined.

$\not\Leftarrow$: A *negative surface coercion* rule says that in the given context, the deep character d must not realize as any $s \in S$.

The next example relates to the consonant gradation described in Table 2.B. The surface coercion rule, which allows P to realize as either p or v would be

$$(P : pv) \Leftarrow \ldots \quad . \tag{2.3.1}$$

The contexts are empty, since P can't realize as something else in any context. The empty contexts always match.

The choice of p and v would be determined by a composite rule

$$(P : v) \Leftrightarrow \ldots (D_{all}, S_{all})* \ (\$, \varnothing) \quad . \tag{2.3.2}$$

The star marks optional repeating, as usual in regular expressions. The rule says that a gradatable P realizes as v if and only if the gradation trigger ”$” follows. There may be other characters between P and the gradation trigger.

In the absence of a gradation trigger P always realizes as p, since P may only realize as p or v, and it may realize as v only in the presence of a gradation trigger.

2.4 Rules and Transducers

The separation to a deep form and a surface from is a familiar concept from transformational grammar theory [Grinder and Elgin, 1973], and the format of the rules is similar to linguistic notation. The remaining problem is to construct a system, which can execute the rules and actually generate the deep form and the surface form.

For execution, the rules are compiled into 2-tape transducers. If some deep/surface form pair violates the rule, the transducer won't accept the pair.

For example, the surface coercion rule (2.3.1) would be translated as in Figure 2.E.

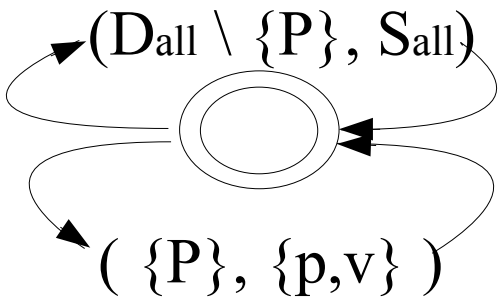

Figure 2.E. Transducer for the surface coercion rule $(P : pv) \Leftarrow \ldots$.

The execution starts from the initial state. There are two transitions – one for each correspondence, where the deep character is P, and another for each correspondence, where the deep character is non-P. If the transducer meets a correpondence with the deep character P and a surface character that is something else than p or v, it stops and rejects the input.

The composite rule (2.3.2) would be translated as two rules. The rules are presented in Figures 2.F and 2.G.

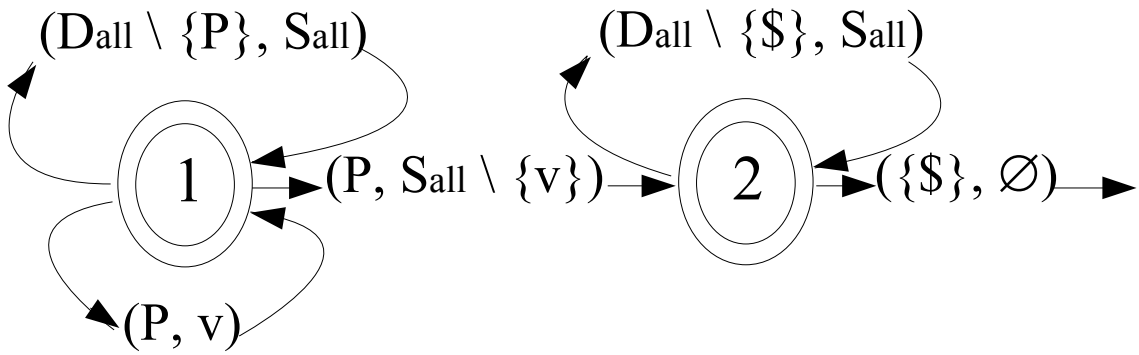

Figure 2.F. Transducer for the surface coercion rule

$$(P:v) \Leftarrow \ldots(D_{all}, S_{all})*(\$, \varnothing) \ .$$

The rule in Figure 2.F says that a deep P must realize as v when followed by the gradation trigger. The rule is violated, if the gradation trigger is present, but P realizes as non-v. The state 1 scans for P realized as non-v. The state 2 scans for the gradation trigger. If both are found, then the input is rejected. The double circles indicate that both states are final states – the rejection is handled by stopping.

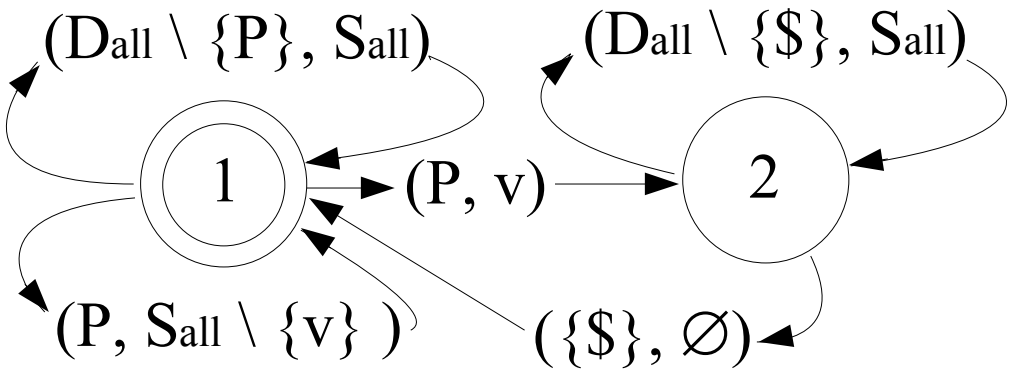

Figure 2.G. Transducer for the context restriction rule

$$(P:v) \Rightarrow \ldots(D_{all}, S_{all})*(\$, \varnothing) \ .$$

The rule in Figure 2.G says that P can realize as v only in the given context. If the transducer finds the correspondence, it has to find the gradation trigger $ before it can accept the word. State 1 is a final state, while state 2 is not.

2.5 Lexicon and Feasible Pairs

Suppose we have a surface form word, which we want to parse. We also have a set of transducers, which filter out the rule-breaking deep forms. How do we generate the hypothesis of the deep form? We can't try all characters from a to z, since the rules ignore most character pairs. For example, the rules in Figures 2.E, 2.F and 2.G accept the surface form ”spy” to become parsed as ”zzz”, simply because they don't say anything about the character z.

The answer two-fold. First, we form a set of *feasible pairs*. Only feasible pairs are allowed to occur in the correspondences. The feasible pairs include

- default character correspondences, like (a:a), (b:b), (c:c),...
- feature-to-null correspondences, like (“+”,∅), (“$”,∅),...
- pairs which are mentioned in the rules like (P,p), (P,v), (I,i), (I,j).

Secondly, we construct the lexicon acceptor, a state machine that accepts all possible deep forms. Figure 2.H gives an example of a lexicon acceptor containing all words in Table 2.B.

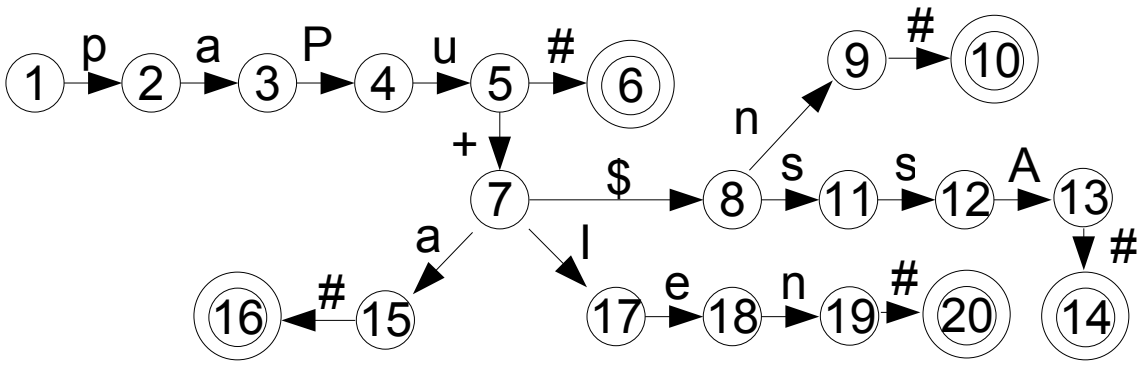

Figure 2.H. Lexicon acceptor containing the deep forms in Table 2.B. The double circles are final states.

Lexicon also solves a problem with characters, which realize as nothing. When we use a lexicon to generate deep form hypothesis, we don't need to worry about situations, where we take reckless amount of empty transitions based on the hypothesis that the following deep form characters are all boundary markers, which realize as empty.

2.6 Parsing in Action

Table 2.I shows how surface form ”papua” gets parsed by the previously mentioned rules and the lexicon in Figure 2.H.

The parsing starts from the initial state. At each state, we make a list of transitions that may correspond to the next surface character. If the surface character is p then possible lexicon acceptor transitions include (1) deep characters that may realize as p, like (p,p) or (P,p), and (2) deep characters that realize as empty, for example (+, ∅).

In the first four transitions, there isn't any choice. The first branching point is at state 5, where also the end symbol (#,#) is accepted by lexicon. However, the next surface character is ”a” and (#, a) is not a feasible pair so this branch is rejected.

In state 7, both ($, ∅) and (a, a) are allowed by the lexicon acceptor, and we try first ($, ∅). At this point, the rule transducer in Figure 2.F stops. It is in state 2, and the transition ($, ∅) is a stopping one. This way, the rule filters out a wrong hypothesis that the lexicon acceptor has generated. Therefore, we have to backtrack back to state 7.

State	Parsed character	Transition	Deep form	Note
1	[p]apua#	1 -> 2	p	Only choice
2	p[a]pua#	2 -> 3	pa	Only choice
3	pa[p]ua#	3 -> 4	paP	Only choice
4	pap[u]a#	4 -> 5	paPu	Only choice
5	papu[Ø]a#	5 -> 7	paPu+	Deep # can't realize as Ø nor ”a”.
7	papuØ[Ø]a#	7 -> 8	paPu+$	Rule rejects $; backtrack
7	papuØ[a]#	7 -> 15	paPu+a	Second choice after $ branch failed
15	papuØa[#]	15 -> 16	paPu+a#	16 is a final state

Table 2.I. Parsing surface form ”papua”. The states refer to the lexicon acceptor in Figure 2.H. Parsed character means the next surface character to be parsed; note that the next character may be either the actual surface character or an imaginary empty character. The deep form is the output of the algorithm.

When the lexicon acceptor reaches state 16, we notice that the input is finished and the lexicon acceptor is at a final state, as is the rule transducer in Figure 2.F. This means that we have found an acceptable deep form, ”paPu+a#”.

3. Formal Language Machinery

This chapter presents the mathematical definitions which we need when formalizing the ideas presented in the previous chapter. Aho and Ullman [1972] is my main source on state machines. Kaplan and Kay [1994] introduce transducers and same-length relations.

3.1 Basic Notations

An *alphabet* Σ is a finite set of *characters*. A *word* is a concatenation of $0 \ldots n$ characters. Let $M = (\Sigma^*, \bullet)$ be the free monoid spanned by Σ and the concatenation operation $\bullet$. Then Σ^* is the *set of all words*. A subset $L \subset \Sigma^*$ is a *language*. I use small letters to denote characters: $a, b \in \Sigma$. The concatenation $a \bullet b$ is written as ab. Capital letters denote sets. Specifically, capital letters $L, M, \ldots$ denote languages.

The empty word $\lambda \in \Sigma^*$ is the identity element of the monoid. The empty language Φ is the empty subset of Σ^*. The set Σ^λ contains the alphabet and the empty word. It is not a subset of Σ, since the empty word is not a character. The length of a word is defined recursively: $Length(\lambda) = 0$, $Length(a) = 1$ and $Length(ax) = 1 + Length(x)$, when $a \in \Sigma$ and $x \in \Sigma^*$.

The main set operations on languages $L, L_1, L_2 \in \Sigma^*$ are

- union: $L_1 + L_2 = L_1 \cup L_2$,
- concatenation: $L_1 L_2 = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$,
- Kleene star: $L^* = \Phi \cup L \cup LL \cup LLL \cup \ldots$, and
- Kleene plus: $L^+ = LL^*$.

3.2 Regular Expressions and Regular Relations

Regular expressions are a shorthand notation for certain formal languages. The regular expressions are defined as follows [Aho and Ullman, 1972]:

1) The empty expression Φ is a regular expression denoting the empty language Φ.
2) Let $a \in \Sigma^\lambda$. Then a is a regular expression denoting the language $\{a\}$.
3) If p and q are regular expressions and P and Q are the corresponding languages, respectively, then
   a) union $p + q$ is a regular expression denoting the language $P \cup Q$,
   b) concatenation pq is a regular expression denoting the language PQ,
   c) Kleene star $p*$ is a regular expression denoting the language P^*.
4) Nothing else is a regular expression.

$Regex(\Sigma)$ denotes the set of regular expressions for the alphabet Σ. If $r \in Regex(\Sigma)$ is a regular expression, then $L(r)$ is the corresponding language. A language is *regular*, if it can be denoted by a regular expression. We use $RL(\Sigma)$ to denote the family of all regular languages over an alphabet Σ. The family is closed under union, concatenation and Kleene star by the definition of regular expressions. It is also closed under complement and intersection [Aho and Ullman, 1972, p.129].

In order to express the left and the right contexts in the rules, which state the correspondence between the surface and the deep form, we need a 2-tape equivalent of regular expressions. The regular relations introduced by Kaplan and Kay [1994] are just that.

From now on, I will use Δ to denote the deep form alphabet, and Σ to denote the surface alphabet. The building blocks of regular relations are *character pairs* $(a,b) \in (\Delta^{\lambda}, \Sigma^{\lambda})$. Sets $D \subset \Delta^{\lambda}$ and $S \subset \Sigma^{\lambda}$ define a *pair set* $(D,S) = \{\ (d,s) \mid d \in D \wedge s \in S\ \}$. *Empty pair* is (λ,λ). A *semi-empty pair* belongs either to the pair set $(\{\lambda\}, \Sigma)$ or $(\Delta, \{\lambda\})$.

We define the pair concatenation operator as $(d_1, s_1) \bullet (d_2, s_2) = (d_1 \bullet d_2, s_1 \bullet s_2)$. We get a monoid by taking the Cartesian product of the component word sets: $(\Delta^* \times \Sigma^*, \bullet)$. The empty pair is the identity element of the monoid. A *2-language* L is a subset of $\Delta^* \times \Sigma^*$. Note that a 2-language is much more than a set of deep words and surface words $(W_d, W_s) \subset (\Delta^*, \Sigma^*)$, since the deep words and surface word are linked. For example, the 2-language $\{(reptile, adder), (mammal, cow)\}$ defines a containment relationship, while two sets $(\{reptile, mammal\}, \{adder, cow\})$ don't. When formulating the two-level model with 2-languages, we link the deep form with the corresponding surface form: For example $\{(spy\#, spy\#), (spy{+}s\#, spies\#)\}$. Parsing becomes a matter of finding the deep form words that correspond to the surface word being parsed. To give another example, the ambiguous words in Figure 1.A would be represented with $\{(lato{+}in\#, ladoin\#), (lada{+}in\#, ladoin\#), (lato{+}i{+}n\#, ladoin\#)\}$.

Regular relations are a shorthand for 2-languages. The following definition is by Kaplan and Kay [1994]:

1) The empty set Φ is a regular relation denoting the empty 2-language $\Phi \subset (\Delta^*, \Sigma^*)$.
2) The pair (a,b) is a regular relation, when $(a,b) \in (\Delta^{\lambda}, \Sigma^{\lambda})$. It denotes the 2-language $\{(a,b)\}$.
3) If r_1 and r_2 are regular relations denoting 2-languages R_1, R_2, then
   - Concatenation $r_1 r_2$ is a regular relation denoting the 2-language $\{\ (d_1 d_2, s_1 s_2) \mid (d_1, s_1) \in R_1 \ \wedge\ (d_2, s_2) \in R_2\}$.
   - Union $r_1 + r_2$ is a regular relation denoting 2-language $R_1 \cup R_2$.
   - Kleene star $r_1 *$ is a regular relation denoting $\Phi + R_1 + R_1 R_1 + \ldots$.
4) Nothing else is a regular relation.

A 2-language is *regular*, if it can be denoted by a regular relation. A 2-language L is *same-length*, if the corresponding surface form and deep form words are equally long: that is, $(w_d, w_s) \in L$ implies $length(w_d) = length(w_s)$. Later we will prove that regular same-length 2-languages are closed under complement and intersection, just like regular languages.

3.3 Finite State Machines and Transducers

A *finite state machine* is a 5-tuple $M = (Q, \Sigma^\lambda, \delta, q_0, F)$, where

- Q is the set of states,
- Σ^λ is the input alphabet extended with the empty word,
- $q_0 \in Q$ is the initial state,
- $F \subset Q$ is the set of final states,
- $\delta \subset Q \times \Sigma^\lambda \times Q$ is the transition relation.

A finite state machine starts from the state q_0 and reads input from the tape. It moves from state q_1 to state q_2 by reading a character a from the tape, if $(q_1, a, q_2) \in \delta$. It is also possible to move between states q_1 and q_2 without reading anything, when $(q_1, \lambda, q_2) \in \delta$.

A state machine accepts input $w \in \Sigma^*$, if it is possible to move from the initial state to some final state $f \in F$ while reading the word character-by-character. The language accepted by the state machine, $L(M)$, is the set of input words it accepts. The set of all finite state machines that use alphabet Σ^λ is denoted by $SM(\Sigma)$.

It can be proven that the family of regular languages equals the family of languages acceptable by finite state machines [Aho and Ullman, 1972, p.119]. The proof is constructive and is based on building a state machine out of a regular expression and the other way around.

If we understand the regular expression as a tree, we can also *reverse* it, so that the language it defines is reversed. This happens by reversing all concatenations. It is tricky to prove, since we have to establish a mapping between the regular expression tree and the word.

A state machine is *deterministic*, if the state and the next character unambiguously define the next state. Determinization is a process, where the result is a deterministic state machine accepting the same language. Subset construction [Aho and Ullman, 1972, p.117] is the classic way to determinize a state machine. In the transition relation of the resulting state machine, the current state and the next character uniquely determine the next state:

$$(q_1, x, q_2) \wedge (q_1, x, q_3) \Rightarrow q_2 = q_3 .$$

The difference between a *finite transducer* and a finite state machine is that a transducer has two tapes. The transition relation has two conditions that must be fulfilled – one for

each tape. For this reason, the transition relation is a subset of $Q \times \Delta^{\lambda} \times \Sigma^{\lambda} \times Q$, where Δ and Σ are the deep form and the surface form alphabets, respectively. The set of all transducers over alphabets Δ and Σ is denoted by $TR(\Delta, \Sigma)$.

The transducer accepts a pair of words (w_1, w_2), if it is possible to move from the initial state to a final state in such a way that all characters of both words have been consumed. When T is a transducer, the 2-language accepted by the transducer is denoted by $L(T)$.

It is also possible to run the transducer so that the other tape is empty, in *transformation mode*. Suppose that only the deep form of a word is given. Then we have only one condition for transitions. When we make a transition, we *write the surface letter to the other tape*. When the input word ends and we are in the final state, we output the surface word. This process transforms a deep form word into a set of surface form words.

It is possible to construct a transducer from a regular relation with an algorithm, which is very similar to the state machine construction algorithm. The state machine construction algorithm can be found for example at Aho and Ullman [1972]. The only difference between the definitions of regular relations and regular expressions is the alphabet where we operate: When state machine definition talks about Σ^{λ}, the transducer definition talks about $\Delta^{\lambda} \times \Sigma^{\lambda}$. The adjustment to the construction algorithm is equally simple, as seen in Algorithm 3.A. Kaplan and Kay [1994] assumed that it is easy to construct a transducer from a regular relation, as it in fact is, but didn't specify the exact algorithm to do so. Therefore I have formalized the algorithm myself.

Algorithm 3.A: Builds a transducer from a regular relation.

Input: A regular relation r, parsed into a tree $T(r)$. Each leaf of $T(r)$ contains a character pair, which may be empty or semiempty. The inner nodes contain unions, concatenations or Kleene stars.

Output: A transducer T accepting the language denoted by r.

Algorithm:

1. Start form the root of $T(r)$.
2. Parse the left and the right subtree using this algorithm, if they exist. We call the resulting left subtransducer $T_1 := (Q_1, \Delta^{\lambda}, \Sigma^{\lambda}, \delta_1, q_1, F_1)$ and the right subtransducer $T_2 := (Q_2, \Delta^{\lambda}, \Sigma^{\lambda}, \delta_2, q_2, F_2)$.
3. If the root is the empty language Φ, we construct a transducer which accepts nothing. We set $T := (\{q_0\}, \Delta^{\lambda}, \Sigma^{\lambda}, \varnothing, q_0, \varnothing)$. Since T has no final states, it clearly accepts nothing.
4. If the root is a word $(d, s) \in (\Delta^{\lambda}, \Sigma^{\lambda})$, we construct a transducer with a single transition (d, s) from the initial state to the final state. We set

$T := (\{q_0, q_1\}, \Delta^\lambda, \Sigma^\lambda, (q_0, d, s, q_1), q_0, \{q_1\})$. Clearly this accepts only the language $\{(d, s)\} \subset \Sigma^*$.

5. If the root is the union of regular relations accepted by the transducers T_1 and T_2 , we make a new indeterministic transducer, where we can move from the initial state q_0 to either T_1 or T_2 by using an empty transition. As a result, we set

$$T := (q_0 \cup Q_1 \cup Q_2,\ \Delta^\lambda,\ \Sigma^\lambda,$$
$$\{(q_0, \lambda, \lambda, q_1), (q_0, \lambda, \lambda, q_2)\} \cup \delta_1 \cup \delta_2 ,$$
$$q_0,\ F_1 \cup F_2).$$

6. If the root is the concatenation of the subrelations accepted by the transducers T_1 and T_2 , we build a transducer, where all paths from the initial state to a final state pass through both T_1 and T_2 . The new transducer starts from the initial state of the first transducer T_1 . From all the final states of the first transducer we add an empty transition to the initial state of the second transducer T_2 . The set of final states of the new transducer is the set of final state of T_2 . As a result, we set

$$T := (Q_1 \cup Q_2,\ \Delta^\lambda,\ \Sigma^\lambda,$$
$$\delta_1 \cup \delta_2 \cup \{ (f_1, \lambda, \lambda, q_2) \mid f_1 \in F_1 \}$$
$$q_1,\ F_2).$$

7. If the root is the Kleene star of the left subrelation, and T_1 is the transducer accepting it, we modify T_1 to accept the empty language and to loop in order to accept several repetitions. First, we make the inital state a final state. Secondly, we add empty transitions from all the final states to the initial state. As a result, we set

$$T := (Q_1, \Delta^\Lambda, \Sigma^\lambda,$$
$$\delta_1 \cup \{(f, \lambda, \lambda, q_1) \mid f \in F_1\},$$
$$q_1, F_1 \cup q_1).$$

3.5 Same-Length Languages

The similarity between transducers and state machines can be characterized by an isomorphism. The formulation of the isomorphism is my own, since Kaplan and Kay [1994, p.343] consider it trivial enough to merit only a passing mention in one sentence. We interpret the two tapes as one tape, where each character has two components. Let $\Gamma = \Delta^\lambda \times \Sigma^\lambda$ be the state machine alphabet. Then $f_{alphabet} : \Delta^\lambda \times \Sigma^\lambda \to \Gamma^\lambda$ is the character isomorphism:

$f_{alphabet}(x, y) = (x, y)$, *for all* $x \in \Delta, y \in \Sigma$,

$f_{alphabet}(\lambda, \lambda) = (\lambda, \lambda)$.

The word mapping $f_{word} : \Delta^* \times \Sigma^* \to \Gamma^*$ reduces a series of characters:

$f_{word}(\lambda, \lambda) = (\lambda, \lambda)$,

$f_{word}(d \bullet d_{rest}, s \bullet s_{rest}) = f_{alphabet}(d, s) \bullet f_{word}(d_{rest}, s_{rest})$.

The transducer isomoprhism $f_{transducer}: TR(\Delta, \Sigma) \to SM(\Gamma)$ switches the alphabet of the transitions:

$$f_{transducer}((Q, \Delta^{\lambda}, \Sigma^{\lambda}, \delta, q_0, F)) =$$
$$(Q, \Gamma^{\lambda}, \{ (q_1, f_{alphabet}(d, s), q_2) \mid (q_1, d, s, q_2) \in \delta \}, q_0, F)$$.

The only problem concerns semiempty transitions. They don't have a natural interpretation in state machines. Figure 3.B illustrates this. Suppose there are 3 transitions from the current state – one with transition condition (d, s), another with (d, λ), and yet another with (λ, s). The next pair of input characters is (d, s). In the realm of transducers, it is clearly possible to take any of the three transitions. In the realm of state machines, (d, s), (d, λ) and (λ, s) represent different members of the alphabet Γ. The fact that any of these transitions is possible would mean that the same input can denote three different elements of the input alphabet Γ.

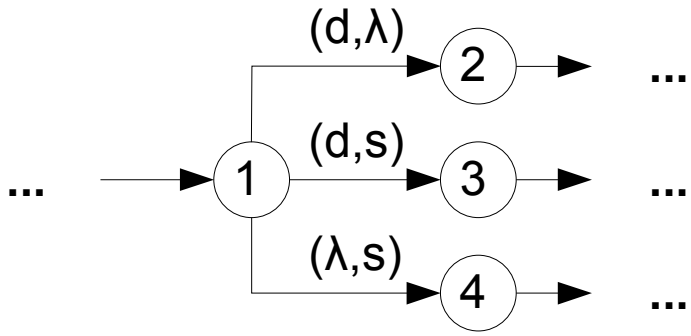

Figure 3.B. A situation, where semiempty transitions make the isomorphism fail.

The answer is to scrap semiempty transitions. We restrict ourselves to *same-length languages*. A 2-language L is same-length, if the words in each word pair $(x, y) \in L$ are equally long, that is, $Length(x) = Length(y)$. We denote the family of same-length languages with deep alphabet Δ and surface alphabet Σ with $SameLength(\Delta, \Sigma)$.

If L is a same-length language, and a transducer T accepts L, and T has no semiempty transitions, then the state machine isomorphism preserves the set of accepted words: $(w_1, w_2) \in L(T) \Leftrightarrow f_{word}(w_1, w_2) \in L(f_{transducer}(T))$.

The state machine mapping guarantees that transducers without semiempty transitions are isomorphic to finite state machines. Next, we prove that all regular same-length 2-languages are accepted by some transducer without semiempty transitions. This perfects the link between regular same-length languages and state machines.

Before presenting the proof as Theorem 3.C we need to define the concept of *imbalance*. Suppose that there is a path $Path$ from the initial state q_0 to state $State$. Now

$Imbalance(q_0, State, Path) = deep\ form\ length - surface\ form\ length$.

In this case, the imbalance is calculated along a specific path. However, if some path leads from $State$ to a final state, then the path must correct the imbalance.

Suppose that the imbalance is k . Then all paths from $State$ to any final state must make a similar $-k$ correction, since a same-length transducer can't accept unbalanced input. Therefore, imbalance does not depend on the path. If p_1 and p_2 are two paths from the initial state to the state $state$, then

$Imbalance(q_0, State, p_1) = Imbalance(q_0, State, p_2)$.

This means that given a specific transducer $T = (Q, \Delta^{\lambda}, \Sigma^{\lambda}, \delta, q_0, F)$ we can define a function $Imbalance : Q \to \mathbb{N}$ so that

$Imbalance(State) = Imbalance(q_0, State, Path)$,

where $Path$ can be any path between q_0 and $State$.

Theorem 3.C [Kaplan and Kay 1994, p.343]**:**

A 2-language S is a same-length regular 2-language if and only if it is accepted by a transducer, which has no semiempty transitions.

Proof:

$\Leftarrow$: If the transducer has no semiempty transitions, then each transition reads 0 or 1 characters from both tapes. Therefore, no transition creates imbalance between the surface word length and the deep word length.

$\Rightarrow$: First of all, the language is accepted by some transducer T , since Algorithm 3.A can be used to build a machine for any regular 2-language. We prove that if a transducer T accepts the same-length language S , then T can be transformed into a transducer with no semiempty transitions.

We prove that if the maximal imbalance of a machine is k , we can reduce the imbalance to $k-1$. We show that we can remove any state, which has imbalance k . By removing all of them, we get a machine with imbalance $k-1$. We assume that the transducer has no (λ, λ) -transitions. They are easy to remove by an obvious generalization of the algorithm that deletes such transitions from a state machine.

Suppose a transducer has maximal imbalance $k > 0$ at state s . Now, all incoming transitions are either of the form (v, λ) or (x, y) , since otherwise some previous state would have bigger imbalance. Similarly, all outgoing transitions are of the form (λ, w) or (x, y) . Figure 3.D illustrates this.

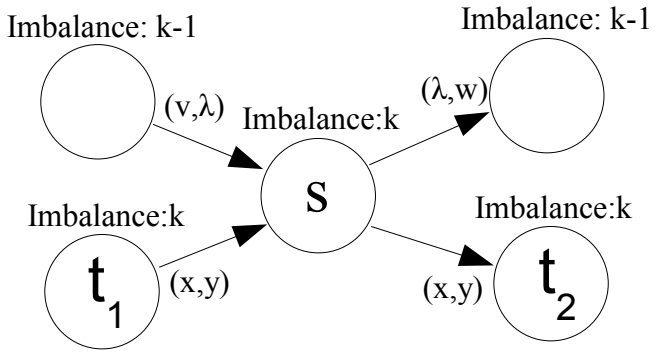

Figure 3.D. A state with maximal imbalance.

Next, we remove transitions of the form (x, y), so that all incoming transitions are of the form (v, λ) and all outgoing transitions are of the form (λ, w). Figure 3.E illustrates this. Suppose there is an incoming transition of the form (x, y) from state t_1 to s. We split the transition (t_1, x, y, s) into two transitions, which use a new intermediate state u_1. The new transitions are (t_1, λ, y, u_1) and (u_1, x, λ, s). Now u_1 is a new state with imbalance $k-1$. The same process can be applied to outgoing transitions of the form (x, y). This way, we remove all transitions of the form (x, y).

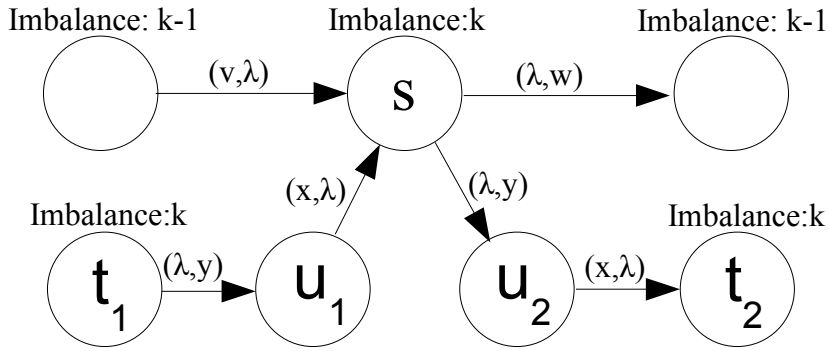

Figure 3.E. Removing transitions of the form (x,y) from the state with maximal imbalance.

Now we are ready to smite the rogue state s. If there are n incoming transitions of form (x, λ) and m outgoing transitions of form (λ, y), we can replace s by $n \times m$ transitions of the form (x, y). Figure 3.F illustrates this.

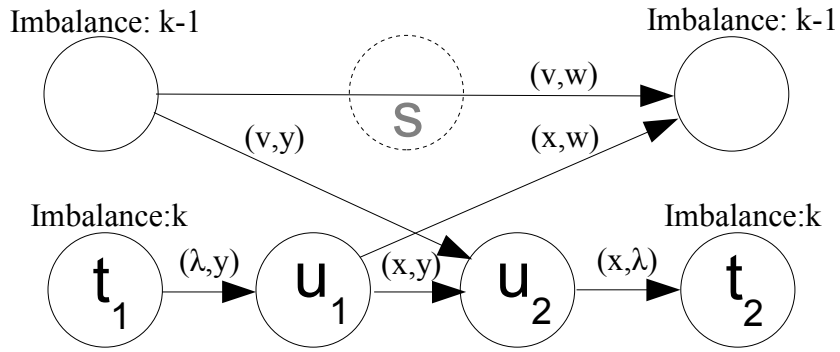

Figure 3.F. Removing the state with maximal imbalance.

We can continue the process until maximal imbalance has been reduced to 0, and generalization to negative imbalance $(-k)$ is obvious.

Hence, the theorem follows.□

We have constructed an isomorphism, which converts between state machines and same-length transducers while preserving the accepted language. To put it formally, when $(w_{1,}w_2)$ is an element of a same-length language, $(w_1,w_2)\in L(T)\Leftrightarrow f_{word}(w_1,w_2)\in L(f_{transducer}(T))$. Now we import some state machine theorems into same-length relations.

Since the regular languages are closed under

- union,
- concatenation,
- complement,
- intersection,
- Kleene star and
- reversal,

also the regular same-length 2-languages are.

However, the imported *same-length complement is not the same as native complement*. This is because of the nature of the inverse mapping. Suppose we have a state machine S in $\Gamma=(\Delta\times\Sigma)\cup(\lambda,\lambda)$, which accepts the language Γ^* . When we use the isomorphism to obtain the transducer, *it only accepts the set of same-length word pairs*. The full language of same-length pairs is Γ^* , while without the same-length restriction the full language is $\{(w_1,w_2)\mid w_1\in\Delta^*,w_2\in\Sigma^*\}$.

4. Two-Level Rules and Their Application

4.1 The Rules

The following definitions has been adapted from Ritchie [1992]. Unless mentioned otherwise, all words belong to the regular same-length 2-language $\Gamma^* = (\Delta, \Sigma)^*$.

A 3-tuple (x, a, y) is a *partitioning* of a word w, if $w = xay$, where $a \in \Gamma$ and $x, y \in \Gamma^*$. A *context pair* (l, r) consists of a left word $l \in \Gamma^*$ and a right word $r \in \Gamma^*$. Either or both of them may be empty.

A context pair (l, r) *matches* to a partitioning $(x, (s, d), y)$, if there exist $x_{start}, y_{end} \in \Gamma^*$ such that $x = x_{start} l$ and $y = r y_{end}$. In this case, the word $x(s, d)y$ can be expressed as $x_{start} l (s, d) r y_{end}$. A set of context pairs matches a partitioning if one of them does.

For example, the word $(fly+s, flies) \in \Gamma^*$ can be partitioned as $((fly, fli), (+, e), (s, s))$. The context pair $((y, i), (\lambda, \lambda))$, which only inspects the left context, matches to the partitioning.

A *two-level rule* $(Pair, Contexts)$ consists of a correspondence pair set $Pair = (\{d\}, S)$ where $d \in \Delta$ and $S \subset \Sigma$, and a set *Contexts* of context pairs. The number of context pairs is often infinite.

In a *regular two-level rule*, the context set can be denoted by a same-length regular relation. Regular two-level rules can be written as $(Pair, L, R)$, where *Pair* is a correspondence pair, and L and R are regular relations. If $(Pair, Contexts)$ is the "plain" rule and $(Pair, L, R)$ is the same regular rule, then the set *Contexts* is equivalent to the set of context pairs implied by the regular relations L and R, which is $\{(l, r) \mid l \in L \wedge r \in R\}$.

A two-level rule $((\{d\}, S), Contexts)$ *contextually allows* a word w, if any character pair $(d, s) \in (\{d\}, S)$ appears only within the given context. To put it formally, for each partitioning $P = (Left, (d, s), Right)$, where $s \in S$, the context set *Contexts* matches P. Especially, if the pairs in $(\{d\}, S)$ are not present in the word, the rule trivially allows the word.

For example, the context restriction rule in Chapter 2 that controls consonant gradation, $(P:v) \Rightarrow \ldots (D_{all}, S_{all}) * (\$, \varnothing)$, would be written as $((\{P\}, \{v\}), (\lambda, \lambda), (D_{all}, S_{all}) * (\$, \varnothing))$. The rule contextually allows $(paPu\$+n, pavu\varnothing\varnothing n)$, since it contains the gradation trigger \$. However, it doesn't allow $(paPu+a, pavu\varnothing a)$, since the partitioning $((pa, pa), (P, v), (u+a, u\varnothing a))$ contains the correspondence pair but the right side doesn't match to the regular relation $(D_{all}, S_{all}) * (\$, \varnothing)$, as it lacks the gradation trigger \$.

A two-level rule $((\{d\},S),Contexts)$ *coercively allows* a word w, if the deep form character d always realizes in the surface as some $s \in S$ in the given context. To be formal, for each partitioning $P=(Left,(d,x),Right)$ that matches to $Contexts$ it holds that $x \in S$. The rule trivially allows all words, where either d is not present in the deep form, or no partitioning matches any context in $Contexts$.

For example, a rule that coerces the + in "y+s" to realize as "ies" would be written as $((\{+\},\{e\}),(y,i),(s,s))$. However, the rule doesn't suffice alone: it allows $(fly+s, fly \varnothing s)$ since the context doesn't match to the only relevant partitioning: $((fly,fly),(+,\varnothing),(s,s))$ doesn't have left context (y,i). For this reason, we need another surface coercion rule $((\{y\},\{i\}),(\lambda,\lambda),(+,S_{any}))$, which signals that a deep form + on the right side makes y realize as i. This second rule rejects the partitioning $((fl,fl),(y,y),(+s,\varnothing s))$.

A set of rules contextually allows a word w, if for each partitioning $(L,(d,s),R)$ either (d,s) is not in the correspondence of any rule, or some rule with correspondence (d,s) contextually allows w. Note that if there are two context restriction rules, which concern the same pair, it is enough for one of them to accept w.

A set of rules R coercively allows a word w, if all rules in R allow w. This is consistent with the idea that the rules are filters, and if one filter is violated, then the input is rejected.

The next example demonstrates interaction between a set of rules. It is about consonant gradation, where gradated K realizes as an apostrophe between two identical vowels, as in *riuku - riu'un*. The "linguistic" notation for this is

$$
\begin{aligned}
(K:') \Leftrightarrow\ & Vowels(a,a)\ \ldots\ (a,a)(\$,\varnothing)\ \| \\
& Vowels(e,e)\ \ldots\ (e,e)(\$,\varnothing)\ \| \\
& Vowels(i,i)\ \ldots\ (i,i)(\$,\varnothing)\ \| \\
& Vowels(o,o)\ \ldots\ (o,o)(\$,\varnothing)\ \| \\
& Vowels(u,u)\ \ldots\ (u,u)(\$,\varnothing)\ \| \\
& Vowels(y,y)\ \ldots\ (y,y)(\$,\varnothing)\ \| \\
& Vowels(\ddot{a},\ddot{a})\ \ldots\ (\ddot{a},\ddot{a})(\$,\varnothing)\ \| \\
& Vowels(\ddot{o},\ddot{o})\ \ldots\ (\ddot{o},\ddot{o})(\$,\varnothing)\ .
\end{aligned}
$$

In mathematical notation this composite rule would be expressed as a set of 8 rules, which are enforced both as context restriction rules and as surface coercion rules. The first rule would be $(\ (\{K\},\{'\}),\ Vowels(a,a),\ (a,a)(\$,\varnothing)\ )$. Now we see why sets of context restriction rules have to treated with "one rule accepts, all rules accept" policy. Otherwise the perfectly valid *riuku – riu'un* inflection would be rejected by the other 7 rules. After all, the partition $((riu,riu),(K,'),(u\$+n,u\varnothing\varnothing n))$ is not allowed by the rule that concerns vowel "a", when the rule is considered as a single rule.

A two-level grammar (CR, SC) consists of a set CR of context restriction rules, and a set SC of surface coercion rules. The set of *feasible pairs* is the union of *default pairs* and *special pairs.* In a default pair, the character realizes as itself. To put it formally, $DefaultPairs = \{(a, a) \mid a \in \Delta \cap \Sigma\}$. A *special pair* is a pair which is mentioned in the rules: $SpecialPairs = \{(d, s) \mid ((\{d\}, S), X) \in CR \cup SC, s \in S\}$.

A two-level grammar (CR, SC) allows a word w , if w is a concatenation of feasible pairs, CR allows w contextually, and SC allows w coercively.

Ritchie's [1992] definition of feasible pairs also included any pairs that are mentioned in the context sets of the rules, but there are practical reasons not to do so. There are unfeasible pairs, which are nevertheless very useful. For example, the pair set (Δ, Σ) can be used to skip any single character. The second difference is that Ritchie [1992] didn't define default pairs and special pairs. Thirdly, in his definition the correspondence part of a two-level rule was a mere pair, rather than a pair set, making rules like $(P : pv) \Leftarrow \ldots$ impossible.

4.2 Rules as Transducers

We consider the rules as filters. Each coercion rule forms one filter. Each set of context restriction rules concerning the same correspondence pair forms one filter. Any input/output pair must pass through all filters simultaneously.

In order to apply the rules to a pair of strings we compile them into transducers. The compilation can be done by hand [Koskenniemi, 1983], or by the method presented later in the thesis, or by method devised by Kaplan and Kay [1994].

A correct compilation of a rule or a set of rules produces a transducer, which accepts a 2-language word if and only if the rule allows the word contextually or coercively, depending on the type of the rule.

The rules are executed in parallel: we take one input character at a time, and feed it to all transduers. Since the transducers are same-length ones, they can be intersected in order to produce a single transducer, even if it may not be economical in terms of memory.

The basic parsing algorithm is the following:

Algorithm 4.A: Parses a word.

Input: A set T of rule transducers. A set F of feasible pairs. A deep form acceptor S . A word w to be parsed.

Output: The possible deep forms of the word.

1. Set the input cursor to the beginning. Set the lexicon acceptor to the initial state. Set all rule transducers to the initial state.
2. Take a character of surface input. Let it be c .

3. Make a list of feasible pairs, where the surface character is *c* , or the surface character is empty.
4. Make a list of possible lexicon transitions. A lexicon transition is possible, if the transition condition is the deep character for some feasible pair listed in 3.
5. For each possible transition:
   5.1. Push (1) the lexicon acceptor state, (2) the rule transducer states and (3) the input cursor to the stack.
   5.2. Go to the next state in lexicon acceptor and rule transducers.
   5.3. Advance input cursor: Read the next character.
   5.4. *If the input is finished and a new character could not be read,* check if all transducers are at a final state. If so, output the lexicon path as one possible parsing result.
   5.5. *If there is an input character,* read forward recursively. Go to step 2.
   5.6. Pop the lexicon state, rule states and the input cursor from the stack.

Note that we don't know, whether the next input character is 'really' an empty character or the naïve *c* we read from the string. At step 3, we must generate all feasible pairs, where the surface is *c or empty*. This means that also more lexicon transitions are possible.

5. Compiling the Rules

Compilation is a process, where the input is a set of two-level rules, as defined in Chapter 4. The output is a transducer, which accepts the input if and only if the rules allow the input contextually/coercively. Past compilation efforts have concentrated on regular two-level grammars. Also in this thesis all rules are assumed to be regular. After reviewing past algorithms, a novel algorithm is presented. It differs from the previous ones in ease of implementation. It works entirely within the same-length framework, and therefore doesn't need full, variable length transducer support.

5.1 Previous Compilation Algorithms

First, I'll review the method of Kaplan and Kay [1994], which is based on composition of several transducer. We proved in Chapter 4, that for any regular same-length 2-language it is possible to create a transducer, which doesn't have semiempty transitions. This means that we can compose several variable-length transducers into a single same-length one, provided that the language they accept is same-length.

The method of Kaplan and Kay [1994] is based on first introducing context markers, then placing the restriction on the core correspondence, and then removing the context markers. The result is highly indeterministic and contains semiempty transitions, but is guaranteed to be reducible to a same-length transducer. Table 5.A illustrates their method.

Phase	Input	Output
Introduce context markers	fly+s	<f<>l<>y<>+<>s>
Restrict left and right context	<f<>l<>y<>+<>s>	<f<l<y><+<s
Replace	<f<l<y><+<s	<f<l<i><+<s
Remove context markers	<f<l<i><+<s	fli+s

Table 5.A. The phases of rewriting rule $y \rightarrow i \ / \ \ldots +$ *in Kaplan and Kay's method.*

The example in Table 5.A. is a rewriting rule rather than a two-level rule. The first phase introduces context markers to the word. The second phase removes those context markers that don't match the right context. The left context is empty, therefore it always matches. The third phase does the replacement to the location, which is surrounded by both the left and the right context marker. The final phase removes the context markers. Kaplan and Kay [1994] also discuss compiling two-level rules with similar methods.

The Xerox FST library (XFST) implements a large set of state machine algorithms, including two-level rule compilation. There is every reason to believe that it implements the compilation method of Kaplan and Kay, since they work for Xerox.

The second known two-level compilation algorithm is part of PC-KIMMO [Antworth, 2004] software. PC-KIMMO is a free, open-source implementation of the two-level model. It includes a rule compiler. It is the only rule compiler implementation with a freely available source code.

The documentation is not too specific about the compilation algorithm. I inspected the source code, and failed to find a proper finite state transducer library. Therefore I believe that the compilation method in PC-KIMMO is heuristic and limited to basic cases only.

5.2 Compiling Context Restriction Rules

First, I'll present two same-length operators that are heavily utilized in the compilation algorithm. These are from Kaplan and Kay [1994]. *IfLeftThenRight* transformation is based on two regular relations, one left and one right. It accepts the input if the beginning doesn't match the left regular relation, or the beginning matches the left relation and the end matches the right relation. Let $\Gamma = (\Delta, \Sigma) \cup (\lambda, \lambda)$ be our alphabet, and let $\sim C$ denote the complement of C. Then

$$IfLeftThenRight(L, R) = \Gamma^* - L(\sim R) = \sim(L(\sim R)) \quad .$$

The "reverse" operation is *IfRightThenLeft*. It is defined as

$$IfRightThenLeft(L, R) = \Gamma^* - (\sim L)R = \sim((\sim L)R) \quad .$$

Two-level rules can be considered as filters, which reject illegal strings. This suggests the following method of checking context restriction rules. When you notice a 'restricted' correspondence, check that it has a legal context on the left and the right side.

Single context restriction rules contain only one context, where the correspondence(s) are allowed:

$$(d:S) \rightarrow L \ldots R \quad .$$

Batch context restriction rules contain several possible contexts:

$$(d:S) \rightarrow L_1 \ldots R_1 \ \vee \ \ldots \ \vee \ L_n \ldots R_n \quad .$$

The difference between single and batch rules that makes it difficult to compile batch rules is the interaction between the contexts. Some contexts may match to both L_1 and L_n.

5.2.1 Compiling Single Rules

This compilation method is presented in Kaplan and Kay [1994], but in much less detail. First we prove that a single rule with a left and a right context can be divided into two simpler rules: One including only the left context, and another including only the right context. To put it formally,

$$(d:S) \Rightarrow L \ldots R$$
$$\Leftrightarrow$$
$$(d:S) \Rightarrow L \ldots \ \wedge \ (d:S) \Rightarrow \ldots R \quad .$$

Theorem 5.B:

A context restriction rule $(d:S)\Rightarrow L\ldots R$ is equivalent to two separate rules, one of them containing only the left context and the other containing only the right context.

Proof:

We prove that given a word $w\in\Gamma^*$ of input, the original rule $(d:S)\Rightarrow L\ldots R$ allows it if and only if the left subrule $(d:S)\Rightarrow L\ldots$ and the right subrule $(d:S)\Rightarrow\ldots R$ allow it. The original rule accepts w, if and only if for an arbitrary relevant partitioning $P=(l,(d,s),r)$, where $s\in S$, P matches the context pair set (L,R). Now

P matches (L,R)

$\Leftrightarrow$ $l\in xL\wedge r\in Ry$ for some $x,y\in\Gamma^*$. The equivalence holds by the definition of partition matching a set of context pairs.

$\Leftrightarrow$ $(l\in xL\ \wedge\ r\in(\lambda,\lambda)y')\ \wedge\ (l\in x'(\lambda,\lambda)\ \wedge\ r\in Ry)$ for some $x,y,x',y'\in\Gamma^*$. We have introduced two extra conditions, which can be always fulfilled by choosing $y'=r$ and $x'=l$.

$\Leftrightarrow$ $(L,(\lambda,\lambda))$ matches P $\wedge$ $((\lambda,\lambda),R)$ matches P. This is true by the definition of a partition matching a set of context pairs.

$\Leftrightarrow$ $(d:S)\Rightarrow L\ldots$ $\wedge$ $(d:S)\Rightarrow\ldots R$. The equivalence holds because we chose the partitioning P and the word w arbitrarily.

Hence, the theorem follows.□

Next, we show that the rule $(d:S)\Rightarrow L\ldots$ is equivalent to $\mathit{IfRightThenLeft}(\Gamma^*L,(d,S)\Gamma^*)$. We already know how to reduce the *IfRightThenLeft* operator into basic 2-language operations which we know how to compute, so this finishes the compilation of the left subrule.

Theorem 5.C:

A context restriction rule $(d:S)\Rightarrow L\ldots$ with only the left context is equivalent to $\mathit{IfRightThenLeft}(\Gamma^*L,(d,S)\Gamma^*)$.

Proof:

Let $w\in\Gamma^*$ be an arbitrary word. The left subrule $(d:S)\Rightarrow L\ldots$ allows w, if and only if for each relevant partitioning $P=(l,(d,s),r)$ where $s\in S$ it is true that (L,λ) matches P. Now firstly, we'll rephrase the left context condition:

(L,λ) matches $P=(l,(d,s),r)$

$\Leftrightarrow$ $l\in xL$, for some $x\in\Gamma^*$,

$\Leftrightarrow$ $l\in\Gamma^*L$.

Secondly, we can express the presence or absence of (d, s) where $s \in S$ in another way. It is equivalent to say that w can be partitioned into $P = (l, (d, s), r)$ for some $l \in \Gamma^*$, and to say that $w \in l(d, S)\Gamma^*$ for some $l \in \Gamma^*$. Therefore

w is accepted by $(d : S) \Rightarrow L \ldots$,

$\Leftrightarrow$ for all partitionings $P = (l, (d, s), r)$ it is true that (L, λ) matches P. This is true by definition of a rule allowing a word contextually.

$\Leftrightarrow$ For all partitionings $P = (l, (d, s), r)$ it is true that $l \in \Gamma^* L$. The equivalence is true, because we showed earlier that (L, λ) matching P is equivalent to $l \in \Gamma^* L$.

$\Leftrightarrow$ If $w \in l(d, S)\Gamma^*$ for some $l \in \Gamma^*$, then $l \in \Gamma^* L$. This simply expresses the presence and absence of (d, S) in another way.

$\Leftrightarrow$ If the right side of w matches to $(d, S)\Gamma^*$, then the left side matches to $\Gamma^* L$.

$\Leftrightarrow$ $w \in IfRightThenLeft(\Gamma^* L, (d, S)\Gamma^*)$.

Hence, the theorem follows.□

Similarly, the right subrule is equivalent to $IfLeftThenRight(\Gamma^*(d, S), R\Gamma^*)$. To sum up,

$$(d : S) \Rightarrow L \ldots R$$
$$= IfRightThenLeft(\Gamma^* L, (d, S)\Gamma^*) \cap IfLeftThenRight(\Gamma^*(d, S), R\Gamma^*).$$

5.2.2 Compiling Batch Rules

The difficulties with batch rules like $(d : S) \Rightarrow L_1 \ldots R_1 \parallel \ldots \parallel L_k \ldots R_k$ are:

- Splitting the rules to the left subrule and the right one requires some preparation, since we don't want to allow $L_i(d, S)R_j$ where $i \neq j$.
- The left contexts or the right contexts of the subrules may intersect.

Our solution is to first split the k original rules into n subrules, where the left contexts don't intersect. This disintersection process may change the number of rules. Let $(d : S) \Rightarrow L_1' \ldots R_1' \vee \ldots \vee L_n' \ldots R_n'$ be the disintersected rules. The fact that left contexts are independent means that $\Gamma^* L_i' \cap \Gamma^* L_j' = \emptyset$ when $i \neq j$. Compiling the disintersected rules is relatively straightforward, and a method to achieve it will be presented after the disintersection algorithm.

Algorithm 5.D splits the subrules so that their left contexts don't intersect.

Algorithm 5.D: Convert a batch rule into a format where the left contexts don't intersect.

Input: A batch rule $(d : S) \Rightarrow L_1 \ldots R_1 \vee \ldots \vee L_k \ldots R_k$.

Output: A batch rule $(d:S) \Rightarrow L_1' \ldots R_1' \vee \ldots \vee L_n' \ldots R_n'$, where the left contexts do not intersect: $i \neq j \Rightarrow \Gamma^* L_i' \cap \Gamma^* L_j' = \varnothing$.

Algorithm:

1. *Unprocessed* is an array of input subrules with Γ^* prefix.
   $Unprocessed := \{ (\Gamma^* L_i, R_i) \mid ((\{d\}, S), L_i, R_i)\ is\ an\ input\ subrule \}$.
2. *Processed* is an empty array of disintersected subrules.
3. While *Unprocessed* is not empty:
   3.1. $Current := Unprocessed[1]$. Let $(L, R) := Current$.
   3.2. *IntersectionLanguage* will be the left context in a new, disintersected rule. It is formed with a series of intersections. *IntersectedRules* is a list of original rules that are combined to form *IntersectionLanguage*. Initialize $IntersectionLanguage := L$, $IntersectedRules := \{ (L, R) \}$.
   3.3. For all subrules $(L_i, R_i) \in Unprocessed$:
      3.3.1. If the left contexts intersect, that is, $L_i \cap IntersectionLanguage \neq \varnothing$:
         3.3.1.1. $IntersectionLanguage := IntersectionLanguage \cap L_i$.
         3.3.1.2. $IntersectedRules.add((L_i, R_i))$.
   3.4. Subtract *IntersectionLanguage* from the other left contexts. For each rule $(L_i, R_i) \in IntersectedRules$ do:
      3.4.1. $L_i := L_i \cap \sim IntersectionLanguage$. This change must be propagated to *Unprocessed*.
      3.4.2. If L_i becomes an empty language: Remove (L_i, R_i) from *Unprocessed*.
   3.5. Create a new, disintersected rule and add it to *Processed*:
      3.5.1. $L_{new} := IntersectionLanguage$.
      3.5.2. Create the right context. Initialize $R_{new} = \varnothing$. For each rule $(L_i, R_i) \in IntersectedRules$ do:
         3.5.2.1. $R_{new} := R_{new} \cup (R_i \Gamma^*)$.
      3.5.3. $Processed.Add(L_{new}, R_{new})$.

To see how Algorithm 5.D works, let's look how it handles a subrule (L, R) where left context doesn't intersect with any other subrule. At some iteration of the loop at step 3, the algorithm is going to take $(\Gamma^* L, R)$ as the *Current* rule, and initialize *IntersectionLanguage* as $\Gamma^* L$. The loop at step 3.3 finds out that only $(\Gamma^* L, R)$ itself intersects with $(\Gamma^* L, R)$. As a result, after the loop at step 3.3 $IntersectedRules = \{(\Gamma^* L, R)\}$. At step 3.4, *IntersectedRules* has exactly one element. When the left context of this element $(\Gamma^* L, R)$ is substracted from itself, the result is an empty language and the rule $(\Gamma^* L, R)$ is removed from *Unprocessed*.

At step 3.5, the new rule that is added to *Processed* is $(\Gamma^* L, R)$. The nonintersecting left context passes through the algorithm unscathed.

What guarantees that the algorithm stops? The loop at step 3.3 collects a subset of rules to *IntersectedRules* and calculates the intersection of their left contexts. Later, loop 3.4 substracts the resulting *IntersectionLanguage* from the left contexts at the *Unprocessed* . After this, the "diminished" rules in the array *IntersectedRules* no longer intersect. If there are n rules, there are $2^n - 1$ possible nonempty subsets of rules. Therefore, *IntersectedRules* can have at most $2^n - 1$ different values during the execution of the algorithm. After they are exhausted, the algorithm stops. It is easy to construct an example, where full $2^n - 1$ iterations are needed. For example, rule $(x : y) \Rightarrow (\{abde\}, \{a\}) \ldots (r, r) \;\|\; (\{abcf\}, \{a\}) \ldots (s, s) \;\|\; (\{acdg\}, \{a\}) \ldots (t, t)$ is split to full $2^3 - 1$ rules, since all combinations of left contexts intersect in a unique way. This also means that the worst-case performance of the algorithm is exponential, but that would require a very special grammar to cause any trouble in practice.

What guarantees that the resulting left contexts do not intersect? In short, the loop at step 3.4. The *IntersectionLanguage* is used as the left context of any new rules, and since it is subtracted from the existing rules, they can't intersect with it.

After the disintersection has been done, the rule can be compiled into $n+1$ transducers: One of them guarantees proper left context and the other n guarantee proper right context. The transducer ensuring that the left context is correct is

IfRightThenLeft($\Gamma^*(L_1' + \ldots + L_n')$, $(d, S)\Gamma^*$) .

The remaining n transducers check the right contexts. Since the left contexts do not intersect, only one of them is triggered for each instance of $c \in (d, S)$ in a word.

IfLeftThenRight($\Gamma^* L_1'(d, S)$, $R_1'\Gamma^*$) ,

...

IfLeftThenRight($\Gamma^* L_n'(d, S)$, $R_n'\Gamma^*$) .

This method of compiling a batch of context restriction rules is novel.

5.3 Compiling Surface Coercion Rules

This method is from Kaplan and Kay [1994]. The surface coercion rule $(d : S) \leftarrow L \ldots R$ means that in the given context, a deep form character d must realize as some surface character $s \in S$. In other words, we want to reject strings which contain

$L(d, \sim S)R$.

When we modify this expression to ignore characters in the beginning and in the end, it becomes

$\Gamma^* L(d, \sim S)R\Gamma^*$.

Since this is what we want to reject, the positive filter is

$$\sim(\Gamma^* L(d:\sim S) R \Gamma^*) \quad .$$

5.4 Implementing the Algorithms

The rule compilation algorithm is a series of operations on same-length 2-languages. However, the practical implementation is done with finite state transducers. The following standard state machine operations are used in the compilation algorithm:

- union,
- concatenation,
- intersection,
- complement,
- determinization.

It is possible to implement also *IfLeftThenRight* and *IfRightThenLeft* with the operations listed above. Regarding left and right contexts, Algorithm 3.A showed a way to build a transducer from a regular relation.

One operation that still needs to be mentioned is checking whether a machine accepts the empty language. It is needed when disintersecting the left contexts of a batch context restriction rule. We know that the smallest machine that doesn't accept anything contains only one state; a state that is initial but not final. Therefore, if we minimize the transducer as if it was a state machine, we can deduce from the minimized transducer if it accepts the empty language or not. Minimizing is also useful for decreasing the sizes of the transducers for other reasons.

6. The Lexicon

In Chapter 2 we saw how parsing uses lexicon. When the parsing algorithm reads a surface form input character, it uses the lexicon to generate a list of possible deep form output characters. Rules filter out the wrong characters from this list.

In Chapter 5 we compiled rules into transducers. In this chapter, we compile the vocabulary into a transducer and combine it with the rule transducers to provide one big transducer for parsing from the surface form to the deep form.

This entire chapter is the first mathematical formalization of continuation classes [Koskenniemi, 1983]. The developers of two-level morphology probably didn't see the topic as worth exact formalization. After two-level morphology, morphological tools have developed towards general finite state libraries [Beesley and Karttunen, 2003], where there is no reason to restrict oneself to continuation classes.

6.1 Continuation Classes

In Chapter 2 we said that the two-level model assumes inflection to be fundamentally concatenative. This means that in the deep form, words are inflected by attaching prefixes and suffixes. We also know that many words have similar inflection. We want to group related inflection suffixes into classes; this way we can denote the inflection of a word by a reference to the class of continuations.

Before defining continuation classes we need to discuss *tags*. They are the final result of parsing and the raw material for producing words. For example, the surface form “going” should be parsed into tag list (“go”, “progressive”). When producing words, for example the tag list (“go”, “imperfect”) should produce the surface form “went”. Tags can denote words or grammatical categories. Sometimes it is possible to split the deep form into “tag-size” pieces so that each tag corresponds to a part of the deep form string. Sometimes a deep form string corresponds to several tags but can't be analysed into subcomponents.

Let Δ be the deep form alphabet and T the tag alphabet. The smallest unit of vocabulary is a *word triple* $(Stem, Continuation, Tags)$ where $Stem \in \Delta^*$ is the string that characterises the word or inflection suffix, $Continuation$ is the name of the continuation class that can be appended after the stem (we define continuation classes in the next paragraph), and $Tags \in T^*$ is a string of tags that identify the meaning of the stem. For example $(“go”, IrregularVerbInflection, “go”)$ or $(“ing”, End, “progressive”)$ may be word triples in an English grammar.

A *continuation class* groups related words or inflection suffixes together. It is a 4-tuple $(Name, Triples, \Delta, T)$ where $Triples$ is a set of word triples. The word triples in $Triples$ use Δ as their deep alphabet and T as their tag alphabet. When a word triple references to a continuation class, it uses $Name$ as the handle. A continuation

class is *empty* if it contains no word triples. Continuing the previous example, a continuation class $IrregularVerbInflection$ could contain word triples for

- forming a present tense with no suffix,
- forming a singular 3rd present tense with -s or -es and
- forming a progressive form with -ing.

A *union* of two continuation classes $A=(Name_a, Triples_a, \Delta, T)$ and $B=(Name_b, Triples_b, \Delta, T)$ denotes the union of their word triples: $A \cup B = Triples_a \cup Triples_b$.

Since continuation classes refer to one another, we need a container structure for them. A *lexicon* is a 5-tuple $(\Delta, T, Classes, Root, End)$ where $Classes$ is a set of continuation classes where stems belong to Δ^* and tags are strings in T^* . The continuation class $Root$ starts the word-forms. The continuation class End signals the end of a word-form. Continuation class references in a lexicon must be consistent: their names have to be unique, and all continuation classes mentioned in the word triples must exist.

The following set of tables describes the inflection of singular Finnish nouns. It is based on the two-level Finnish grammar of Koskenniemi [1983]. The example is quite long, but it demonstrates many different things in this and the next chapter. The crude formula for noun inflection is

stem [+case] [+possessive suffix] [+clitic].

This means that each word starts with the stem. After the stem, 0 – 3 optional suffixes can follow. Cases denote mainly various prepositions, for example, the phrase "in a house" is expressed with the inessive case in Finnish. Possessive suffixes denote ownership: "my house" is expressed with singular 1st person possessive suffix. Clitics have various meanings that depend heavily on context. Table 6.A lists two noun stems. Table 6.B lists singular case inflection. However, in any real-life grammar also the plural noun inflection has to be included. Table 6.D lists the possessive suffixes, and Table 6.C lists the clitics. Table 6.E and Table 6.F list *Root* and *End* continuation classes, respectively. They are required for a valid two-level lexicon. Finally, Table 6.G lists the continuation classes, which are unions of other continuation classes.

Stem	**Continuation**	**Tags**
talo	noun	"house"
öljy	noun	"oil"

Table 6.A. Example of a continuation class. The continuation class is named "stem".

Table 6.B lists the cases. The non-alphabet character "+" denotes the boundary between the stem and case suffix and is necessary for some rules. The character "$" denotes consonant gradation. Chapter 1 had an example of consonant gradation, but this example does without. The last two lines with "@" denote compound words. In Finnish,

it is possible to form compound words by attaching substantives or adjectives after the nominative form or the genitive form of a word.

Stem	Continuation	Tags
	clitic_end	singular, nominative
$+n	clitic_end	singular, genitive
+n	possessive	singular, nominative
+n	possessive	singular, genitive
+nA	possessive_clitic_end	singular, essive
+A	possessive_clitic_end	singular, partitive
$+ksi	clitic_end	singular, translative
$+kse	possessive	singular, translative
$+ssA	possessive_clitic_end	singular, inessive
$+stA	possessive_clitic_end	singular, elative
+:n	possessive_clitic_end	singular, illative
+h:n	possessive_clitic_end	singular, illative
$+llA	possessive_clitic_end	singular, adessive
$+ltA	possessive_clitic_end	singular, ablative
$+lle	possessive_clitic_end	singular, allative
$+ttA	possessive_clitic_end	singular, abessive
@	stem	singular, nominative
$+n@	stem	singular, genitive

Table 6.B. Continuation class "case". It represents the Finnish singular noun case inflection. The last two lines represent compound words.

Stem	Continuation	Tags
_hAn	end	hAn
_kA:n	end	kAAn
_kin	end	kin
_kO	end	kO
_pA	end	pA
_kinkO	end	kinkO
_kA:nkO	end	kAAnkO
_kOhAn	end	kOhAn
_kOs	end	kOs
_pAhAn	end	pAhAn
_pAs	end	pAs

Table 6.C. Continuation class "clitic". Denotes clitics, which serve various grammatical roles that are highly context-dependent.

Stem	Continuation	Tags
/ni	clitic_end	Singular 1st
/si	clitic_end	Singular 2nd
/nsA	clitic_end	Singular/Plural 3rd
/:n	clitic_end	Singular/Plural 3rd
/mme	clitic_end	Plural 1st
/nne	clitic_end	Plural 2nd

Table 6.D. Continuation class "possessive". Denotes possessive suffixes that indicate ownership. Possessive suffixes can be appended after stem or after case inflection.

It is useful to start and end the word with a special character, since some rules need to refer to the beginning or end of a word. We use # as the special character. Table 6.E and Table 6.F introduce this special character. Table 6.G, defines the continuation classes that are combinations of existing classes.

Stem	Continuation	Tags
#	stem	

Table 6.E. Continuation class "Root". All word-forms start from this class.

Stem	Continuation	Tags
#		

Table 6.F. Continuation class "End". All word-forms end in this class.

Continuation Class	Definition
possessive_clitic_end	$possessive \cup clitic \cup end$
clitic_end	$clitic \cup end$
noun	$case$

Table 6.G. Continuation classes that are combinations of other continuation classes. Union means the union of word triples rather than that of continuation classes.

6.2 The Deep Form State Machines

Algorithm 6.H constructs a state machine that accepts all valid deep form words and none of the invalid ones. The state machine is not useful in itself, but the construction algorithm can be modified to produce transducers that convert between the surface form, the deep form and the tag form.

Algorithm 6.H: Builds a lexicon acceptor from a lexicon.
Input: A lexicon $L=(\Delta, T, Classes, Root, End)$.
Output: A state machine $M=(Q, \Delta^{\lambda}, \delta, q_0, F)$ that accepts the deep form of L .
Algorithm:

1. Create an entry state for each continuation class $c \in Classes$. At this point, the entry states are isolated states with no transitions. We denote the entry state of a continuation class $Name$ by $Entry[Name]$. Add entry states to Q .
2. Set the initial state: $q_0 := Entry[Root]$.
3. For all continuation classes $(Name, S, \Delta, T) \in Classes$:
   3.1. For all triples $(Stem, Continuation, Tags) \in S$: Create the transition chain:
      3.1.1. $CurrentState := Entry[Name]$.
      3.1.2. For $k := 1$ to $Length(Stem)$ do:
         3.1.2.1. Create a state and store it to variable $NewState$. Add it to Q .
         3.1.2.2. Add transition $(CurrentState, Stem[k], NewState)$ to δ .
         3.1.2.3. $CurrentState := NewState$.
      3.1.3. Create the transition to the next continuation class:
         3.1.3.1. If $Name$ is End , make $CurrentState$ final.
         3.1.3.2. If $Name$ is not End , add a transition $(CurrentState, \lambda, Entry[Continuation])$ to δ .

Figure 6.K illustrates what kinds of machines this algorithm produces. The lexicon transducer in Figure 6.K converts between the deep form and the surface form. It is produced by a modified algorithm. The only difference to deep form acceptor is that the transducer accepts pairs instead of characters.

We can modify the algorithm to produce a transducer that maps between the deep form and tags. The modification in Algorithm 6.I is restricted to the part where we create the transition chain for a stem. After the transition where tape 1 accepts the last letter of the stem we put transitions that have the tags on tape 2. The transitions up to the last letter don't write any tags. This way, if the transducer reads the deep form it writes appropriate tags. When producing deep forms from tags, at the time the transducer reads a tag it must have gone through a long chain of semiempty transitions that produce the appropriate deep form.

Algorithm 6.I: Creates a transducer that converts between the deep form and the tag form.

Input: A lexicon $L=(\Delta, T, Classes, Root, End)$.

Output: A transducer $M=(Q, \Delta^{\lambda}, T^{\lambda}, \delta, q_0, F)$ that converts between the deep form and the tag form defined by L.

Algorithm:

1. Create an entry state for each continuation class $c \in Classes$. At this point, the entry states are isolated states with no transitions. We denote the entry state of a continuation class $Name$ by $Entry[Name]$. Add the entry states to Q.
2. Set the initial state: $q_0 := Entry[Root]$.
3. For all continuation classes $(Name, S, \Delta, T) \in Classes$ do:
   - 3.1. For all triples $(Stem, Continuation, Tags) \in S$ do: create the transition chain.
     - 3.1.1. $CurrentState := Entry[Name]$.
     - 3.1.2. For $k := 1$ to $Length(Stem)$ do:
       - 3.1.2.1. Create a state and store it to variable $NewState$. Add it to Q.
       - 3.1.2.2. Add transition $(CurrentState, Stem[k], \lambda, NewState)$ to δ.
       - 3.1.2.3. $CurrentState := NewState$.
     - 3.1.3. Add the tags: For $k := 1$ to $Length(Tags)$ do:
       - 3.1.3.1. Create a state and store it to variable $NewState$. Add it to Q.
       - 3.1.3.2. Add transition $(CurrentState, \lambda, Tags[k], NewState)$ to δ.
       - 3.1.3.3. $CurrentState := NewState$.
     - 3.1.4. Create the transition to the next continuation class:
       - 3.1.4.1. If $Name$ is End, make $CurrentState$ final.
       - 3.1.4.2. If $Name$ is not End, add a transition $(CurrentState, \lambda, \lambda, Entry[Continuation])$ to δ.

The second modification, Algorithm 6.J, generates the surface-deep transducer. We have said that rules can be considered as filters that permit or deny certain correspondences between the deep form and the surface form. The role of the surface-deep transducer is to generate the raw data that is filtered by the rules. When parsing, it ensures that the parsed deep forms are in the vocabulary.

Algorithm 6.J: Creates a transducer that converts between the surface form and the deep form, ignoring the rules.

Input: A lexicon $L=(\Delta, T, Classes, Root, End)$. A surface alphabet Σ. A set $F \subset (\Delta, \Sigma)$ of feasible pairs (see chapter 4.1).

Output: A same-length transducer $M=(Q, \Delta^{\lambda}, \Sigma^{\lambda}, \delta, q_0, F)$ that converts between the deep form defined by L and the surface form.

Algorithm:

1. Create an entry state for each continuation class $c \in Classes$. At this point, the entry states are isolated states with no transitions. We denote the entry state of a continuation class $Name$ by $Entry[Name]$. Add entry states to Q.
2. Set the initial state: $q_0 := Entry[Root]$.
3. For all continuation classes $(Name, S, \Delta, T) \in Classes$ do:
   - 3.1. For all triples $(Stem, Continuation, Tags) \in c$, create the transition chain:
     - 3.1.1. $CurrentState := Entry[c]$.
     - 3.1.2. For $k := 1$ to $Length(Stem)$ do:
       - 3.1.2.1. Create a state and store it to variable $NewState$. Add it to Q.
       - 3.1.2.2. Use feasible pairs to find out the set of possible surface characters: $SurfaceSet := \{s \mid (Stem[k], s) \in F\}$.
       - 3.1.2.3. Add transition $(CurrentState, Stem[k], SurfaceSet, NextState)$ to δ.
       - 3.1.2.4. $CurrentState := NewState$
     - 3.1.3. Create the transition to the next continuation class:
       - 3.1.3.1. If $Name$ is End, make $CurrentState$ final.
       - 3.1.3.2. If $Name$ is not End, add transition $(CurrentState, \lambda, \lambda, Entry[Continuation])$ to δ.

In Chapter 2, we ran the rules and the lexicon transducer separately for reasons of conceptual clarity. Now we see that both the surface-deep transducer and the rules are same-length transducers. Running transducers in parallel and rejecting input when one of the transducers rejects it produces the same result as intersecting the separate transducers into one transducer and using it to convert between the deep form and the surface form. It is faster to intersect the lexicon transducer with the rule transducers than to intersect the rule transducers with one another, since many complex interactions between the rules never appear in real-life vocabularies. Algorithm 6.L combines the rules and the surface-deep transducer.

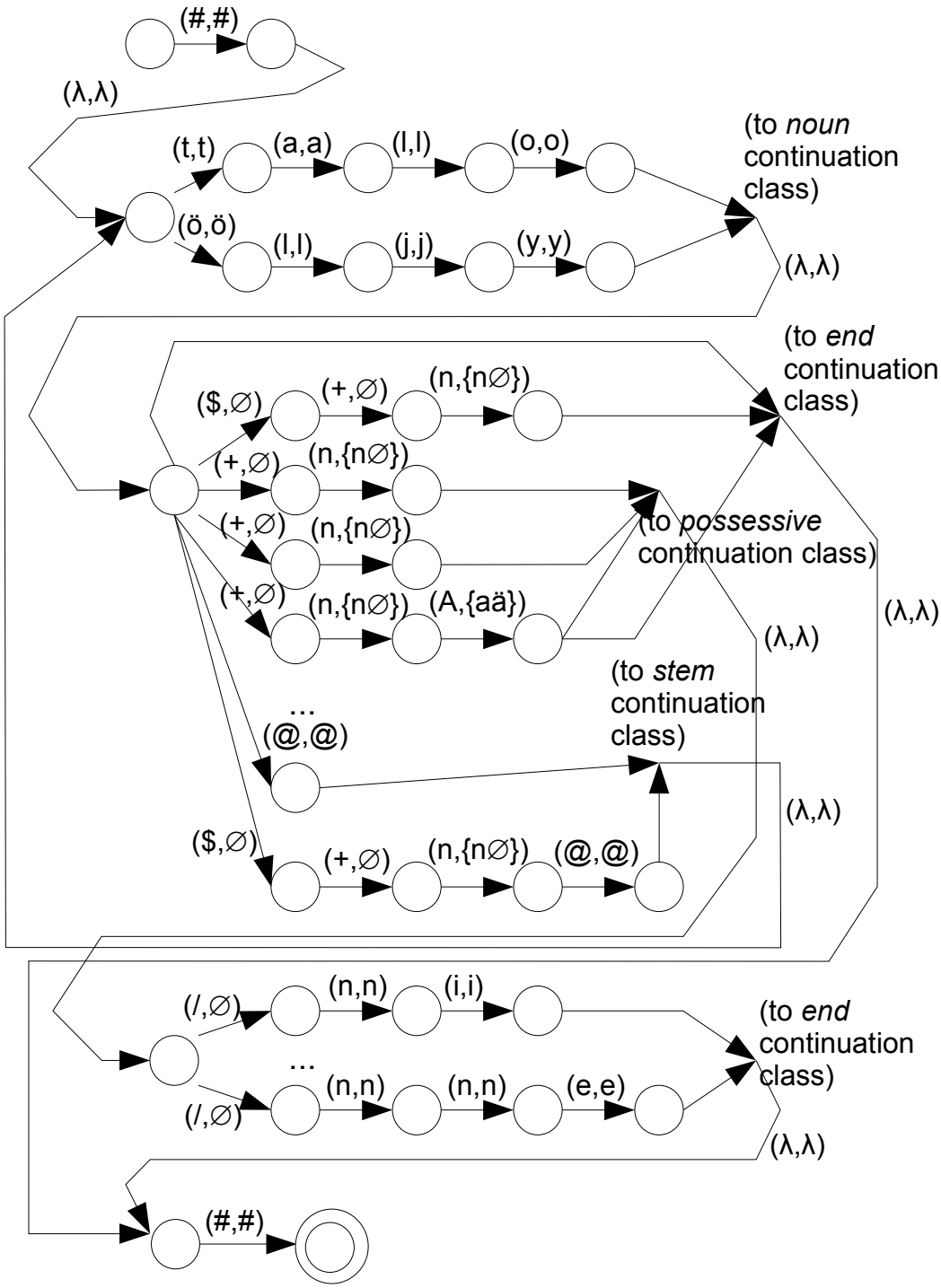

Figure 6.K. Surface-deep transducer generated from the Finnish example. Clitics have been omitted as well as parts of the case and possessive suffix continuation classes.

Algorithm 6.L: Combines the lexicon-based surface-deep transducer and the rules into one transducer.
Input: A surface-deep same-length transducer T . A vector V of same-length rule transducers.
Output: A single transducer that corresponds to running the transducers in parallel.
Algorithm:

1. Initialize: $Result := T$.
2. For all rule transducers $R \in V$ do:
   2.1. $Result := Result \cap R$.

6.3 Removing Semiempty Transitions

In Section 6.2 we created two useful transducers in order to convert between the surface form, the deep form, and the tag form. Both transducers have plenty of semiempty transitions.

The surface-deep transducer creates an illusion of same length between the deep form and the surface form with a dedicated empty character. Special deep characters like inflection boundary markers in the Finnish example realize as empty characters in the surface form. This creates complications in parsing: When we get a real-life surface word, it doesn't have empty characters. When parsing a surface form, we must guess at each point whether the next character is an empty character or the actual character. In this section we treat these transitions as semiempty transitions.

There are two reasons why we need a new type of transducer to express the algorithm that removes semiempty transitions. Firstly, the algorithm is based on concatenating the outputs of the transitions where the input tape is semiempty. Therefore the transducer must be able to write several characters in one transition. Secondly, we must take direction into account and treat the input tape and the output tape differently. It is normal that the output tape has empty transitions: it doesn't create any extra ambiguity when transforming input into output. Empty output has valid uses, for example when converting deep form words to surface form words, where some characters realize as empty. On the other hand, empty transitions in the input tape force us to branch the transduction to a scenario where something is read and another scenario where no input read, slowing down conversion.

The definition also introduces *output function* [Mohri 1996] as preparation to the next chapter about determinization. If the transduction ends at some final state, the output of the output function is written to the output tape. Output function enables us to

get rid of semiempty transitions and have limited amounts of indeterminacy without branching.

A *word transducer with p outputs* $T=(Q, I^{\lambda}, O, \delta, \sigma, q_0, F)$ is a 7-tuple where

- Q is the set of states, q_0 is the initial state and F is the set of final states,
- I^{λ} is the input alphabet extended with the empty word,
- O is the output alphabet,
- $\delta \subset Q \times I^{\lambda} \times O^{*} \times Q$ is the transition relation. Note that the input condition is always a single character or an empty transition while the output is a string.
- $\sigma : F \rightarrow (O^{*})^{p}$ is the output function. If the transduction stops in a final state $q \in F$, the results of the output function $output \in \sigma(q)$ are written to the output tape. Note that the output function can always be replaced with a set of semiempty transitions $(q, \lambda, output, q')$. In the process at most p new states q' are created and q becomes nonfinal.

Converting a regular transducer into a word transducer is almost trivial. The only complication arises from the direction of transduction. One regular transducer splits into two separate word transducers depending on which alphabet we choose to be the input alphabet. After some algorithm modifies the word transducer to have several output letters in one transition, it is no more possible to use the transducer in both directions. Therefore, the plain surface-deep transducer splits into two word transducers: surface-to-deep and deep-to-surface ones.

We say that a transition is *input empty*, if it the input condition is λ. Algorithm 6.M removes input empty transitions from a word transducer.

Algorithm 6.M: Removes input empty transitions from a word transducer.
Input: A word transducer $T=(Q, I^{\lambda}, O, \delta, \sigma, q_0, F)$.
Output: A word transducer $T'=(Q', I^{\lambda}, O, \delta', \sigma', q_0, F')$ without input empty transitions.
Algorithm:

1. Initialize $T' := T$.
2. For each state in $q \in Q'$:
   2.1. While q contains an input empty transition: Let $t := (q, \lambda, Output, q_2)$.
      2.1.1. For each transition in $t' \in q_2$: Let $t' := (q_2, Input_2, Output_2, q_3)$:
         2.1.1.1. Add transition $t_{new} := (q, Input_2, Output \bullet Output_2, q_3)$ to δ'.
      2.1.2. If q_2 is final: Make q final. Since there is no longer a transition from q to q_2, also the output function of q_2 must be copied:
         2.1.2.1. If the output function $\sigma'(q_2)$ is empty, add $Output$ to $\sigma'(q)$.
         2.1.2.2. If $\sigma'(q_2)$ is not empty, add the strings $Output \bullet FuncOutput$ to $\sigma'(q)$, where $FuncOutput \in \sigma'(q_2)$.
      2.1.3. Remove transition t.

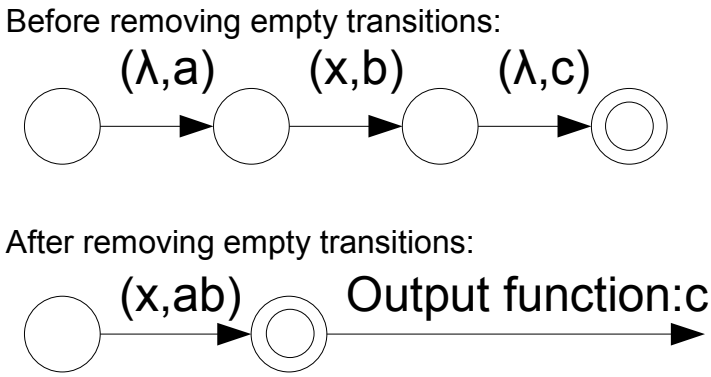

Figure 6.N. An example of using concatenation and output function to get rid of empty transitions.

Figure 6.N illustrates how the algorithm removes semiempty transitions. The algorithm doesn't always terminate: for example if a state q contains an input empty transition to itself. Each iteration of the loop 2.1 removes one input empty transition from the state q by replacing t with t_{new}. However, if $Input_2$ in t_{new} is empty, the iteration doesn't change the number of input empty transitions. Figure 6.O illustrates this.

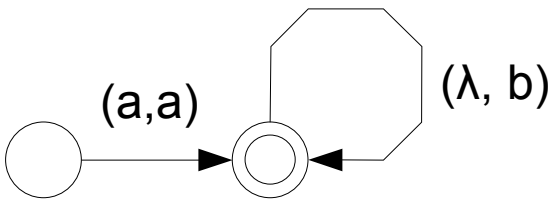

Figure 6.O. An example of a transducer, where it is impossible to remove a semiempty transition.

7. Input Determinization

7.1 The Determinization Algorithm

Given a string of input, a deterministic state machine accepts or rejects the string in linear time, since the next transition is always unique. Determinization is a guarantee of linear performance.

Transducers can be "deterministic" in two different ways. As we saw in Section 3.5, same-length transducers are isomorphic to state machines. We can determinize the transducer as if it were a state machine. When the transducer is used as an acceptor and both tapes are input tapes, this kind of determinization guarantees linear performance. However, if one of the tapes is an output tape that is being written, we can no longer expect linear performance.

The second type of determinization, *input determinization*, guarantees linear performance when the transducer transforms input into output. The trick is that we don't write output before we know exactly what to write. This method is illustrated in Figure 7.A. Note that the input determinized transducer has words as transition conditions, and therefore it is a word transducer as defined in the previous chapter, instead of being a plain transducer as defined in Chapter 3.

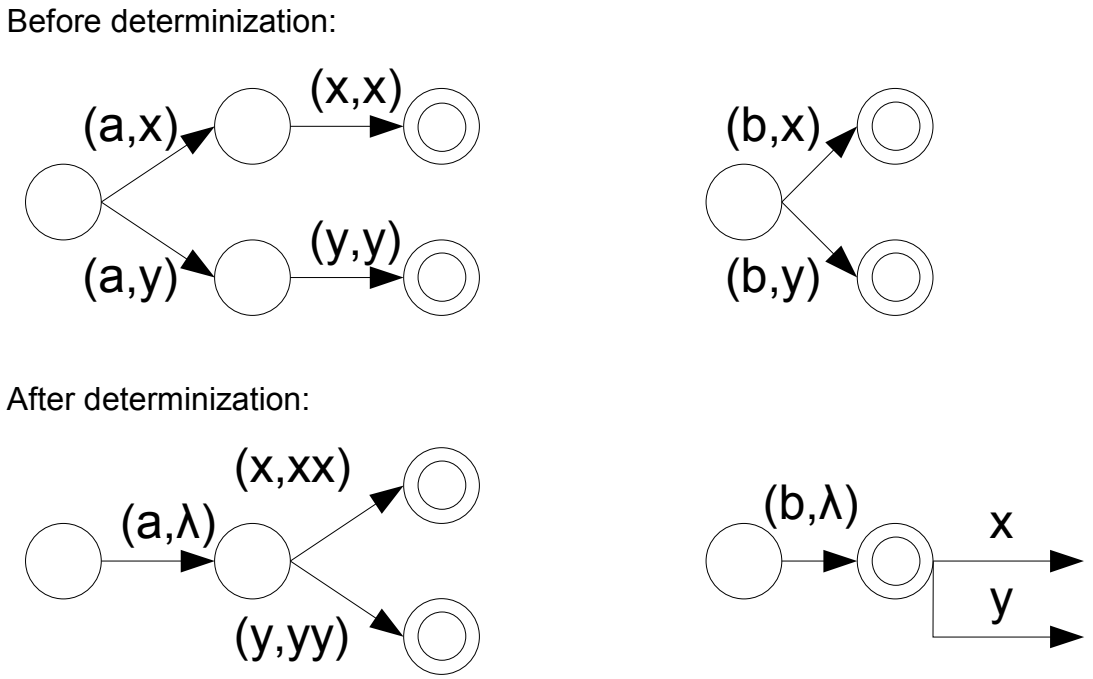

Figure 7.A. Using output function and empty output transitions to make transducers input deterministic.

Mohri [1994] defines three classes of input deterministic transducers:

- *Sequential* transducers are such that at each state, the input letter determines the next state. Sequential transducers don't have an output function.

- *P-subsequential* transducers can have an output function. The output function can write at most p different strings at each final state. The transitions are under the same restrictions as in sequential transducers. P-subsequential transducers run in linear time but can still handle limited amounts of ambiguity.
- *Subsequential* transducers can have at most one output string in each final state. Subsequential transducers can be represented by sequential ones almost always: Only when the initial state is also a final state it is impossible to use a sequential transducer instead.

Algorithm 7.B converts a word transducer into a P-subsequential transducer. The basic idea is to write the output only after we know what to write. Ambiguous output is implicitly stored in "state memory". The algorithm is loosely based on subset construction. In the subset construction, let Q be the set of states of the nondeterministic transducer. Then the states of the deterministic transducer are subsets of Q. In this algorithm, the states of the resulting transducer are sets of $(q, String)$ pairs, where $q \in Q$ and $String \in O^*$ is one possible unwritten output. We call a set of $(q, string)$ pairs a *superstate*.

Not all transducers can be determinized, and sometimes deterministic transducers are much bigger than nondeterministic ones. We'll look at the downsides of determinization later.

Algorithm 7.B [Mohri, 1996]: Determinizes a word transducer.
Input: A word transducer $T = (Q, I^{\lambda}, O, \delta, \sigma, q_{0,} F)$ without empty transitions.
Output: A p-subsequential word transducer $T' = (Q', I^{\lambda}, O, \delta', \sigma', q_0', F')$.
Algorithm:

1. Let *Queue* be the temporary storage for superstates of the resulting transducer. Queue contains *superstates*; that is, sets of $(q, String)$ pairs.
2. Let *StateHash* be a hash table that maps between superstates and states of the resulting transducer: If x is a superstate, then $StateHash(x) \in Q'$.
3. Add initial superstate to queue: $Queue.Push(\{(q_0, \lambda)\})$.
4. Repeat while *Queue* is not empty:
   4.1. Get the next superstate:
      4.1.1. $SuperState := Queue.Head()$.
      4.1.2. $CurrentState := StateHash(SuperState)$.
   4.2. For all final substates (q, Str) in *SuperState* where $q \in F$.
      4.2.1. Make *CurrentState* final: $F' := F' \cup CurrentState$. That is, if there is at least one final substate, then also the superstate becomes final.
      4.2.2. Add *Str* to the output function: $\sigma'(CurrentState) := \sigma'(CurrentState) \cup Str$.

4.3. For all elements of the input alphabet $i \in I$ do:

4.3.1. Make a list of substates with an i-transition: $ISubstates := \{(q, Str) \mid (q, Str) \in SuperState, (q, i, o, q') \in \delta \text{ for some } o, q'\}$.

4.3.2. Make a list of output strings that might be written when reading i: $IOutputs := \{Output \mid (q, Str) \in ISubstates, Output = Str \bullet Common(q, i)\}$ where $Common(q, i)$ is the longest common output prefix when we move from state q with various transitions which read i: $Common(q, i) := LongestCommonPrefix(\{o \mid (q, i, o, q') \in \delta\})$.

4.3.3. Make a list of reachable substates: $TargetStatesStrings := \{(q, Str, o, q') \mid (q, Str) \in SuperState, (q, i, o, q') \in \sigma \text{ for some } o \in O\}$.

4.3.4. Calculate the transition output: $TransitionOutput := LongestCommonPrefix(IOutputs)$.

4.3.5. Calculate the target superstate of the determinized transition:

4.3.5.1. $TargetState := \varnothing$.

4.3.5.2. For each $(q, Str, o, q') \in TargetStatesAndStrings$: create the string part of the (state, string) pair.

4.3.5.2.1. Concatenate "state memory" and normal output: $StateString := Str \bullet o$.

4.3.5.2.2. Remove the output that we write during transition $StateString := StateString.DeleteFromStart(TransitionOutput)$.

4.3.5.2.3. Add (state, string) pair to the target superstate: $TargetState := TargetState \cup (q', StateString)$.

4.3.6. If the $TargetState$ superstate is new:

4.3.6.1. Create $StateHash(TargetState)$.

4.3.6.2. $Queue.Push(TargetState)$.

4.3.7. Add transition from $CurrentState$ to $TargetState$: $\delta := \delta \cup (CurrentState, i, TransitionOutput, StateHash(TargetState))$.

4.4. Remove the $CurrentState$ from $Queue$, as we have finished processing it.

7.2 Problems with Determinization

The positive effects of determinization include linear performance of the resulting transducer and the simplification of other operations like computing the composition of two transducers. Two kinds of adverse effects surface when determinizing natural language transducers.

Suppose that we have a word, which has two interpretations. For example the word "married" be both an imperfective form of verb "marry" and an adjective that has

its own definition in the dictionary. Suppose we construct a surface-to-tags transducer that includes the word “married” and determinize it. The word “married” is represented by a chain of letters. The first state has a transition labeled with 'm' and after the 'd' transition comes the last state. The last state outputs two different interpretations: “married” for the adjective and “marry + imperfective” for the verb. Since there is ambiguity, we must output it in a final state's output function. We can't output it in transitions, since they can give only one interpretation.

The problem surfaces, if the word with many interpretations has inflection. Take the Finnish word “varattu” (reserved) for example. It can be either an adjective or a participle form of a verb. There is a set of suffixes named clitics in the Finnish grammar. There are 11 different clitics, and they are 2-6 characters long. They are listed in Table 6.C in Chapter 6. Clitics can be attached to almost any word, including “varattu”. Normally in a surface-to-tags transducer the clitic string outputs only its own grammatical tag. However, if we add a clitic to the word “varattu”, the end of clitic must output the two different interpretations, since only final states can handle ambiguity. For this reason, determinization makes a dedicated copy of the states and transitions that represent clitics in order to be able to output the two interpretations in the end. Representing the clitics takes much more states and transitions than representing the core word “varattu”. Therefore, determinization increases the size of the transducer, since you need to make dedicated copies of inflection endings. There are also other features in the Finnish grammar that have the same effect as clitics.

The other problem is caused by the fact that adding one word can add an infinite amount of word-forms because of the compound word mechanism. The example in the Chapter 6 illustrates this. After a nominative or a genitive form you can concatenate other words in order to form compound words. This is visible in Table 6.B, where strings that end with @ have continuation class “stem”, which refers back to the table where new words start. Suppose you have multiple interpretations in a genitive or nominative form of a word. The consequence is that adding a new word adds an infinite amount of compound word-forms.

Now, suppose you have a word with two interpretations that can be used as a basis of a compound word. The determinization algorithm is unable to push them to the final states, since the number of required final states is infinite. As a consequence, the machine is undeterminizable.

The next chapter examines empirically just how nonlinear the parsing and production performance is without determinization, and how serious these two drawbacks of determinization are in practice.

8. The Effects of Determinization on Size and Performance

This chapter quantifies the effects of determinization. The rule compiler and the two-level parser/producer were implemented with Java. Also, a 1000-word Finnish vocabulary was collected for testing. Since state machines are fast and Java is slow, it is somewhat pointless to measure time. Therefore, the performance was measured by counting the number of states visited while parsing.

8.1 Test Software and Data

A finite state library is the workhorse of the two-level software package that was used for the experiments. It can handle finite acceptors, same-length transducers and word transducers. It can compute the regular language algebra operations like union and complement, determinize acceptors and same-length machines, and input determinize word transducers. It also contains the special operations needed for rule compilation: constructing machines that accept regular relations, and computing the *IfRightThenLeft* operator. The rule compiler implements exactly the algorithm described in the earlier chapters.

The vocabulary module inputs the data. It reads a set of continuation classes and builds the transducers. First it builds the surface-deep transducer and intersects it with the rule transducers. Then it splits it to surface-to-deep and deep-to-surface transducers. Then it builds the deep-to-tags and tags-to-deep transducers. If configured to do so, it determinizes some of the transducers and possibly combines the surface, deep and tag levels to a single surface-to-tags transducer.

The dictionary and annotator modules work on the level of meanings and tags. They read a dictionary and produce a word list for the vocabulary module without concerning themselves with the details of parsing and production. The annotator can annotate text by performing morphological analysis on words and attaching meanings to them. The dictionary module can also run test cases that ensure that the parsing or production results in the expected surface word-form or tag list, respectively.

The test vocabulary contains 1114 Finnish words, which all belong to a single essay. Therefore, it contains a realistic mix of nouns, verbs and particles. The inflection and rules have been adapted from Koskenniemi [1983]. Altogether, it contains 88 rules. This is somewhat more than what Koskenniemi had, since he had a special notation for grouping some very similar rules together.

8.2 Determinization and the Size of the Transducer

Since possible explosion in size is the main drawback of determinization, a test was performed that compared the size of the 4 transducers before and after determinization. The size was measured in number of states and transitions.

The results are in Table 8.A. The increase in sizes of the transducers reflects the number of processes in Finnish grammar that create ambiguity. The size of the surface-to-deep transducer increases more than tenfold because of the processes described in the previous chapter. In tags-to-deep transformation, there are only two processes that create ambiguity. It is possible to generate the partitive form and the genitive form with several alternative endings, but the resulting words are synonymous. Since tag form only knows that the user wants a "partitive form", producing all partitive or genitive deep forms generates ambiguity. The result is a fivefold increase in states, which is offset by a decrease in the number transitions. Deep-to-surface transformation doesn't really have any significant ambiguity-producing features, so the transducer stays small.

Transducer	Before determinization	After determinization
Surface-to-deep	3190 states, 10683 transitions	103588 states, 112518 transitions
Deep-to-tags	12930 states, 56966 transitions	16226 states, 27690 transitions
Tags-to-deep	4341 states, 41713 transitions	24681 states, 34281 transitions
Deep-to-surface	3190 states, 7054 transitions	3282 states, 7776 transitions

Table 8.A The effect of determinization of transducer size.

Acceptors can also be minimized. The simplest possible minimization algorithm, although by no means newest, fastest nor best, is Brzozowski minimization [Brzozowski 1962]. In this algorithm, the transducer is reversed, determinized, reversed, and determinized again. This implies that also transducers may become smaller if reversed and determinized twice. For transducers, we can apply Brzozowski-style "minimization" where we run the determinization and reversal operations, but use input determinization instead of the state machine determinization. Naturally there is no reason to believe that this "minimization" results in any theoretically significant canonical form. This "minimization" was run on the transducers of Table 8.A. Table 8.B shows the results. The algorithm didn't finish for the last two transducers. For the surface-to-deep transducer the minimizing effect was negligible, probably because the transducer was already minimized as a same-length transducer before conversion into a word transducer. The process has significant effect only on the deep-to-tags transducer.

Transducer	Before minimization	After minimization
Surface-to-deep	103588 states, 112518 transitions	100514 states, 110331 transitions
Deep-to-tags	16226 states, 27690 transitions	12875 states, 15814 transitions
Tags-to-deep	24681 states, 34281 transitions	Couldn't compute
Deep-to-surface	3282 states, 7776 transitions	Couldn't compute

Table 8.B The effect Brzozowski minimization -style transformation on the size of the transducers.

8.3 Determinization and Parsing Performance

The parsing performance was tested with all words in an essay. The essay is the same on which the 1114-word vocabulary is based. Therefore, it presents a realistic annotation scenario, where easy, difficult and ambiguous words occur with real-life frequency. The results are in Table 8.C. In the first line, both transducers are deterministic. In the second line, both transducers are nondeterministic. In the third line, the determnistic surface-to-deep and deep-to-tags transducers were combined into a surface-to-tags transducer; ease of composition is one motivation behind input determinization. The last result is the most interesting: the surface-to-deep transducer was left nondetermistic while the deep-to-tags transducer was determinized. The result is a massive decrease in branching with a negligible increase in size. Once again, input determinization produces little improvement on the surface-to-tags transducer, which was determinized before conversion into a word transducer, while “virgin soil” deep-to-tags transducer does show improvement.

Transducers	States visited	Transducer Size
S2D determinized, D2T determinized	53105	119814
S2D undeterminized, D2T undeterminized	506870	16120
S2D and D2T determinized and combined	23121	
S2D undeterminized, D2T determinized	58368	19416

Table 8.C The effect of determinization of parsing performance. S2D mean surface-to-deep, D2T means deep-to-tags. States visited means how many states were visited while parsing. Transducer size means the number of states in the transducers.

To summarize the results of the empirical part, input determinization does increase the size of the transducers 6-fold and cuts the parsing time to 1/20. With careful selection of deterministic and nondeterministic transducers it is possible to get 10x speed-up with

only a slight increase in transducer size. The effects of determinization vary very much between languages, and it is not possible to draw conclusions that would be valid for all languages. For example, Mohri [1996, p.14] reports that determinization actually compresses French, English and Italian morphological vocabularies.

9. Conclusions

This thesis has summarized the development of the two-level model after Koskenniemi [1983]. Significant development has happened in three areas. Firstly, instead of compiling linguistic rules into transducers by hand, nowadays linguists can use automatic rule compilers. Secondly, all parts of the two-level framework have been expressed in the formal language of transducers and strings, including the lexicon. These two developments were discussed in the Thesis.

The final development has been a shift from two-level morphology to general finite state transducer tools. This development has been spearheaded by Kay and Kaplan and their Xerox Finite State Transducer library. It has succeeded to the point of making two-level morphology obsolete. This development is outside the scope of the Thesis.

List of References

[Aho and Ullman, 1972] Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation and Compiling - Volume 1: Parsing*. Prentice Hall, 1972.

[Antworth, 1990] Evan L. Antworh, *PC-Kimmo, A Two-Level Processor for Morphological Analysis*. Summer Institute of Linguistics, Dallas, Texas, 1990.

[Antworth, 2004] Evan L. Antworth, PC-Kimmo, a morphological parser. Available as http://www.sil.org/pckimmo/ (5.10.2004).

[Beesley and Karttunen, 2003] Kenneth R. Beesley and Lauri Karttunen, *Finite State Morphology*. Leland Stanford Junior University, CSLI Studies in Computational Linguistics N:o 3, 2003.

[Brodda and Karlsson, 1981] Ben Brodda and Fred Karlsson, *An Experiment with Automatic Morphological Analysis of Finnish*. University of Helsinki, 1981.

[Brzozowski, 1962] J. A. Brzozowski, Canonical regular expressions and minimal state graphs for definite events. In: *Mathematical Theory of Automata* (1962), 529-561.

[Grinder and Elgin, 1973] John T. Grinder and Suzette H. Elgin, *Guide to Transformational Grammar*. Holt, Rinehart and Winston Inc., 1973.

[Kaplan and Kay, 1994] Ronald M. Kaplan and Martin Kay, Regular models of phonological rule systems, *Computational Linguistics* **20**, 3 (1994), 331 – 378.

[Koskenniemi, 1983] Kimmo Koskenniemi, *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production.* University of Helsinki, Publications of Department of General Linguistics N:o 11, 1983.

[Koskenniemi, 1985] Kimmo Koskenniemi, Compilation of automata from morphological two-level rules. In: *Papers of the Fifth Scandinavian Conference of Computational Linguistics* (1985), 143-149.

[Mohri, 1996] Mehryar Mohri, On some applications of finite-state automata theory to natural language processing. *Journal of Natural Language Engineering* **2** (1996), 1-20.

[Reape and Thompson, 1988] Mike Reape and Henry S. Thompson, Parallel intersection and serial composition of finite state transducers. In: *Proc. of the 12th International Conference on Computational Linguistics* **2** (1988), 535-539.

[Ritchie, 1992] Graeme Ritchie, Languages generated by two-level morphological rules. *Computational Linguistics* **18**, 1 (1992), 41-59.

Appendix 1. Notation Sheet

These notations are used through the whole thesis. They are explained in detail, when they first appear in the text.

Σ	Generic alphabet; surface alphabet.
Δ	Deep alphabet.
Γ	Same-length alphabet $(\Delta, \Sigma) \cup (\lambda, \lambda)$.
$L, M, \ldots$	Languages.
$S, T, \ldots$	State machines.
$a, b, c, \ldots$	Charaters.
$x, y, z, w, \ldots$	Words.
λ	Empty word.
Φ	Empty language.
$\emptyset$	Empty surface character; empty set.
$RL(\Sigma)$	Regular languages over the alphabet Σ .
$RR(\Delta, \Sigma)$	Regular relations between the two alphabets.
$SL(\Delta, \Sigma)$	Same-length regular relations between the two alphabets.
$SL_{all}(\Delta, \Sigma)$	A same-length relation accepting all same-length pairs.
FT_{all}	A transducer accepting the 2-language $SL_{all}(\Delta, \Sigma)$.

Appendix 2. Sample of the Lexicon Format

The Finnish inflection expressed by the lexicon format is based on Koskenniemi [1983]. The sample is part of verb inflection.

```
LEXICON verb0 = {
        "("     1.3.n/v  "present", "active";
        "$("   verbClitic#    "imperative", "active", "sg2";
        "(mi"   nen/s  "toSubstantive";
        // Agent construction: Kallen osta_ma_ auto.
        "(mA"   sNoCompound/s      "agent", "active";
        "(vA"   aNoCompound      "participle", "present", "active";
        "(mAtTo" n.mA/a "participle", "past", "negative";
}
LEXICON verb1 = {
        "+i("  1.3/v "past", "active";
}
LEXICON verb2 = {
        "(isi("         1.3/v "conditional", "active";
}
LEXICON verb3 = {
        "(ne"       1.3.n/v       "potential", "active";
        "(kOOn"     clitic#      "imperative", "active", "sg3";
        "(kAAmme"   clitic#      "imperative", "active", "pl1";
        "(kAA"      verbClitic# "imperative", "active", "pl2";
        "(kAAtte"   end          "imperative", "active", "pl2";
        "(kOOt"     end          "imperative", "active", "pl3";
        "(kO"       end          "imperative", "active", "negative";
        "*$DA"      4.n/v          "present", "passive";
        "*$Xtiin"   clitic#          "past", "passive";
        "*$XtAisi"  4.n/v          "conditional", "passive";
        "*$XtAne"   4.n/v          "potential", "passive";
        // hänet tuotakoon / häntä ei tuotako
        "*$XtAkO"   4.n/v                "imperative", "passive";
        "(DA"       clitic#  "infinitive", "nominative";
        "(DA+kse"   possessive   "infinitive", "translative";
        "+De+ssA"   possessive#  "infinitive2", "active", "inessive";
        "+Den"      clitic#      "infinitive2", "active", "toAdverb";
...
```

Appendix 3. Sample of the Rule Format

The Finnish rules are based on Koskenniemi [1983]. These rules are part of consonant gradation.

```
RULE consgrad = {

// s.79-81 Consonant gradation
// paikka - paikan
[ "K", null] =>  Vowels [ "lrh", "lrh" ]? [ "k", "k" ]? _ ;
[ "K", "" ] <=>  Vowels [ "a", "a" ] _ [ "a", "a" ] Grad ||
        Vowels [ "e", "e" ] _ [ "e", "e" ] Grad ||
        Vowels [ "i", "i" ] _ [ "i", "i" ] Grad ||
        Vowels [ "o", "o" ] _ [ "o", "o" ] Grad ||
        Vowels [ "u", "u" ] _ [ "u", "u" ] Grad ||
        Vowels [ "y", "y" ] _ [ "y", "y" ] Grad ||
        Vowels [ "ä", "ä" ] _ [ "ä", "ä" ] Grad ||
        Vowels [ "ö", "ö" ] _ [ "ö", "ö" ] Grad;

[ "K", "v" ] =>  Cons [ "uy", "uy" ] _ [ "uy", "uy" ];
[ "K", null ] /=  Cons [ "uy", "uy" ] _ [ "uy", "uy" ];
[ "K", "g" ] => Vowels [ "n", "n"  ] _ ;

// This rule should not allow laKi$+n -> lajin.
// Probably the i should be restricted to plural I.
[ "K", "j" ] => Vowels [ "lrh", "lrh" ]? _ ( [ "eE", "e" ] | [ "E", null ] | [ "i", "i" ] ) ;
// Fix laKi and other non-plural-i forms.
[ "K", "j" ] /= Vowels _ [ "i", "i" ] [ "$", "Z" ]? ["+", "Z"];
// Fix väKE$+ttä -> väjettä.
[ "K", "j" ] /= Vowels _ [ "E", surfAll ];

[ "K", null] /= Vowels [ "lrh", "lrh" ] _ ( [ "eE", "e" ] | [ "E", null ] | [ "i", "i" ] ) ;

[ "P", "v" ] =>  Vowels  [ "lr", "lr" ]? _ ;
[ "P", "m" ] =>  Vowels  [ "m",  "m"  ]  _ ;
[ "P", null] => [ deepAll, surfVowels ][ "lr", "lr" ]? [ "p", "p" ] _ ;

[ "T", "d" ] => [ deepAll, surfVowels ][ "h", "h" ]? _ ;
[ "T", "l" ] => [ deepAll, surfVowels ][ "l", "l" ]  _ ;
[ "T", "r" ] => [ deepAll, surfVowels ][ "r", "r" ]  _ ;
```

```
[ "T", "n" ] => [ deepAll, surfVowels ][ "n", "n" ]  _ ;
[ "T", null] => [ deepAll, surfVowels ][ "lrnh", "lrnh" ]? [ "t", "t" ] [ deepAll, null ]* _ ;

[ "K", "k" ] <= [ "s", "s" ] _ ;
[ "P", "p" ] <= [ "s", "s" ] _ ;
[ "T", "t" ] <= [ "s", "s" ] _ ;

[ "K", "'vZgj" ] <= _ [ deepAllNoLimit, surfAll ]* Grad;
[ "P", "vmZ"   ] <= _ [ deepAllNoLimit, surfAll ]* Grad;
[ "T", "dlrnZ" ] <= _ [ deepAllNoLimit, surfAll ]* Grad;

[ "K", "k" ] <= _ [ deepAllNoGrad, surfAll ]* EndOrLimit;
[ "P", "p" ] <= _ [ deepAllNoGrad, surfAll ]* EndOrLimit;
[ "T", "t" ] <= _ [ deepAllNoGrad, surfAll ]* EndOrLimit;
}
```