CSME 206A Natural language & Speech Processing **Spring Semester**

Lecture 5: Finite Automata and Morphological Parsing

Lecturer: K.R. Chowdhary *: Professor of CS*

5.1 Introduction

Finite-state machines have been extensively used in areas of speech processing. Their use can be justified by both *linguistic* and *computational purpose*. In linguistics, finite automata are convenient as they allow one to describe easily most of the relevant local phenomena encountered in the empirical study of language. They often lead to a compact representation of *lexical rules*, or *idioms* and *cliches*, that appears natural to linguists. Graphic tools also allow one to visualize and modify automata, which helps in correcting and completing a grammar. Other more general phenomena, such as parsing context-free grammars, can also be dealt with using finite-state machines. In fact, the underlying mechanisms in most of the methods used in parsing are related to automata.

From the computational point of view, the use of finite-state machines is mainly motivated by considerations of *time* and *space* efficiency. Time efficiency is usually achieved using deterministic automata. The output of deterministic machines depends, in general linearly, only on the input size and can therefore be considered of optimal time complexity. Space efficiency is achieved with classical minimization algorithms. Applications such as compiler construction have shown deterministic finite automata to be very efficient in practice.

The recent applications of finite automata in natural language processing are, ranging from the construction of lexical analyzers and the compilation of morphological and phonological rules, to speech processing.

In this chapter we will discuss theoretical and algorithmic bases for the use and application of the devices based on FA, that support very efficient programs: the . The idea of deterministic automata is extended to *transducers* with deterministic input, so that these machines produce output strings or weights in addition to (deterministically) accepting input. Hence, these methods are consistent with the initial reasons for using finite-state machines – the time efficiency of deterministic machines, and the space efficiency achievable through minimization algorithms, for sequential transducers.

Both time and space efficiency are important when dealing with languages. Indeed, one of the recent trends in language studies is a large increase in the size of data sets. Lexical approaches have been shown to be the most appropriate in many areas of computational linguistics ranging from large-scale dictionaries in morphology to large lexical grammars in syntax. The effect of the size increase on time and space efficiency is probably the main computational problem of language processing.

The sequential finite-state transducers are used in all areas of computational linguistics. In the following sections, we will discuss in details these devices. We will first consider string-to-string transducers, which have been successfully used in the representation of large-scale dictionaries, computational morphology, and local grammars and syntax, and describe the theoretical bases for their use. In particular, we recall classical theorems and provide some new ones characterizing these transducers.

We then consider the case of sequential string-to-weight transducers. Language models, phone lattices, and word lattices are among the objects that can be represented by these transducers, making them interesting from the point of view of speech processing.

In the last section, we describe some applications of *determinization* and *minimization* of string-to-weight transducers in speech recognition, illustrating them with several results that show them to be very efficient. The implementation of the determinization is such that it can be used on the fly: only the necessary part of the transducer needs to be expanded. This plays an important role in the space and time efficiency of speech recognition. The reduction in the size of word lattices that these algorithms provide sheds new light on the complexity of the networks involved in speech processing.

5.2 Finite Automata

The finite automata are the machines which recognize regular languages, i.e, languages represented by regular expressions (RE). Hence, there is relation of implication like this, $FA \rightarrow REGLANG \rightarrow RExp \rightarrow FA$, in other words, it is a bijection between all three entities. Let us consider that, talk or sound of a sheep (sheep-talk language S) can be represented by strings $S = \{baa!, baaa!, baaa!, ...\}$,. Where each sentence in language is dependent on the length of 'a' sound. The set of strings S can be represented by a regular expression: $baa^{+}!$. However, we will prefer it to be represented by a string $/baa+!/$ – a format of pronunciation. The sounds in set S can be recognized by a FA shown in the Fig. 5.1.

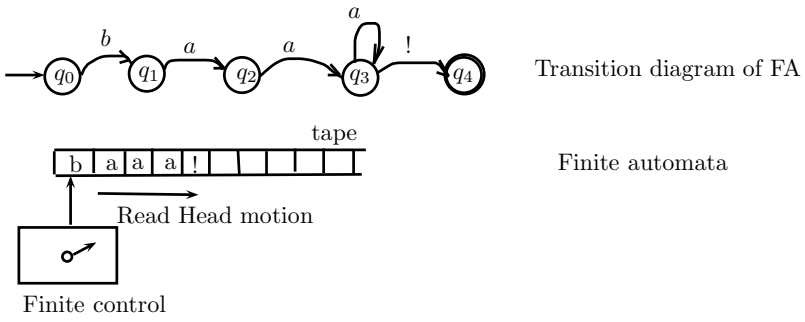

Figure 5.1: Finite Automata.

Formally, a FA is represented by $M = (Q, \Sigma, q_0, \delta, F)$, where

Q is finite set of states; here $Q = \{q_0, q_1, q_2, q_3, q_4\}$

Σ is finite set of alphabets; here $\Sigma = \{a, b, !\}$

δ is transition function; i.e., $\delta : Q \times \Sigma \rightarrow Q$,

F is set of finite states, called accepting or final states, and here, $F = \{q_4\}$,

$S = L(M) = \{baa!, baaa!, baaaa!, \dots\}$.

The recognition process for the language string through FA, is represented by the algorithm 1. For this, the FA's *tape* is divided into squares, called *index* positions, where each position is holding one symbol from alphabet Σ. We consider that w is input string, and $|w| = n$ is length of input [Jurafskyd].

The recognition of a string by FA is searching a tree. Considering the recognition of some string, say length n, for alphabet set $\Sigma = \{a, b\}$, one needs to perform a worst case search of $O(2^n)$ in *space* and *time*. The

Algorithm 1 : function dfa-recognize(tape, machine) return *accept / reject*;

```
1: index ← initial-position-on-tape
2: while True do
3:    if end of input then
4:       if current state== accept then
5:          return accept
6:       else
7:          return reject
8:       end if
9:    else
10:      if transition-table[current-state, tape[index]] == empty then
11:         return reject
12:      else
13:         current-state ← transition-table[current-state, tape[index]]
14:         index++
15:      end if
16:   end if
17: end while
```

time complexity remains $O(2^n)$ for both the BFS as well as DFS. In general, if size of alphabet $|\Sigma| = m$, and length of string (sentence) is n, then space and time both have complexities equal $O(m^n)$, for BFS.

However, in case of DFS, space complexity is $O(2 \times n)$, for branching factor ($b = 2$) and length of string n. As a general case, it is $O(mn)$. However, due to combinatorial explosion of number of states generated, even shorter length strings, makes it difficult to efficiently process the input for recognition. Since, the ordinary brute force algorithms, like DFS and BFS are highly inefficient, better algorithms like *best-first search*, A^*, and SA (*simulated annealing*) are considered superior.

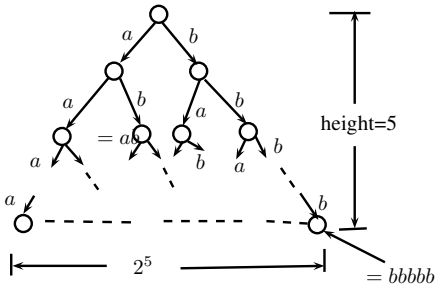

Figure 5.2: Search-tree.

5.3 Finite state transducers

A finite state transducer (FST) is a finite state machine with two tapes: an input tape and an output tape, with finite number of states. The Fig. 5.3 shows the diagram where these (input and output) strings are shown on the transitions, separated by ":". This FST has been used as a translator, as it translates the input sentence "" Hello World" to "Hey there krc".

Thus, an FST is a directed graph, like finite automata, with,

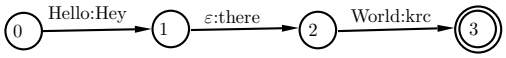

Figure 5.3: An FST as a translator

- Edges / transitions that have input / output labels,
- some times there are empty labels indicated by ε,
- Traversing through to the end of an FST implies the translation of one string into another, or generation of two strings, or relating one string to another,
- There is a defined state, called "start" state, and other ass "final" state.

We define here, the FST as *Mealy machine*, which is an extension of normal finite state (FS) machine. The formal representation of Mealy machine is given by,

$$M = (Q, \Sigma, q_0, \delta, F) \tag{5.1}$$

where,

$Q = \{q_0, q_1, \ldots, q_{N-1}\}$, is finite set of states,
Σ is finite alphabet of complex symbols, and
$\Sigma \subseteq I \times O$,
q_0 is start state,
$\delta : Q \times \Sigma \to Q$, for example, $\delta(q', i : o) = q_j$.

The I and O are input and output symbols, respectively, and both include the symbol ε. For $\Sigma = \{a, b, !\}$, corresponding to the language discussed earlier, the FST has $i : o$ set as, {*a:a, b:b, !:!, a:!, a:ε, ε:!*}.

The FSTs are useful for variety of applications:

- *Word inflections:* For example, finding the plural of the words, cat → cats, dog → dogs, goose → geese, etc.
- *Morphological parsing:* Extracting the properties of a word, e.g., cats → cat + [nouns] + [plural].
- Simple word *translations:* For example, US English to UK English.
- Simple *commands* to computer.

5.4 Sequential and P-subsequential Transducers

Sequential string-to-string transducers are used in various areas of natural language processing. Both determinization and minimization algorithms are used for the class of p-subsequential transducers ($p \geq 1$), which

includes sequential string-to-string transducers[1]. In this section, the theoretical basis of the use of sequential transducers is described. A lot of theoretical base exists for using these devices for applications to natural language processing.

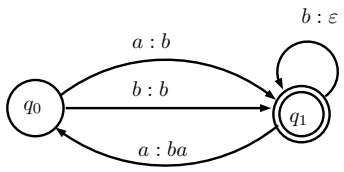

Figure 5.4: Sequential Transducer.

A sequential transducers is a transducers with deterministic input. At any state of such transducers, at most one outgoing arc is labeled with a given element of the alphabet. Fig. 5.4 shows an example of a sequential-transducer. Note that output labels might also include the empty string (ε). However, an empty string is not allowed on input. The output of a sequential transducer is not necessarily to be deterministic. For example in Fig. 5.4, where two distinct arcs with output labels b leave the state q_0. The Sequential transducers are computationally efficient because their use with a given input does not depend on the size of the transducer but only on the size of the input. Using a sequential transducer with a given input consists of following the only path corresponding to the input string and in writing consecutive output labels along this path. Hence, the total computational time is linear in the size of the input, provided that the cost of copying out each output label does not depend on its length.

Definition 5.1 String-to-string Transducer. *A sequential string-to-string transducer T is a 7-tuple $(Q, q_0, F, \Sigma, \Delta, \delta, \sigma)$, where,*

Q is set of states,

q_0 is initial state,

$F \subseteq Q$ is set of final states,

Σ is set of input alphabet,

Δ is set of output alphabet,

δ is state transition function, $\delta : Q \times \Sigma \to Q$, and

σ is output function, $\sigma : Q \times \Sigma \to \Delta^*$.

The functions δ and σ are generally *partial functions*, i.e., a state $q \in Q$ does not necessarily admit outgoing transitions labeled on the input side with all elements of the alphabet. These functions can be extended to mappings from $Q \times \Sigma^*$ by the following classical recurrence relations:

$$\begin{aligned} \forall s \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \quad \text{there is, } & \delta(s, \varepsilon) = s, \delta(s, wa) = \delta(\delta(s, w), a); \text{ and} \\ & \sigma(s, \varepsilon) = \varepsilon, \sigma(s, wa) = \sigma(s, w)\sigma(\delta(s, w), a). \end{aligned} \tag{5.2}$$

Thus, a string $w \in \Sigma^*$ is accepted by Transducer T iff $\delta(q_0, w) \in F$, and in that case the output of the transducer is $\sigma(q_0, w)$.

[1]Here P stands for number of strings which are handled in parallel. At the minimum, a transducer handles one input string, and one output string, thus, the most common case is $p = 2$.

Resolving ambiguity The Sequential transducers discussed above can be generalized by introducing the possibility of generating an additional output string at final states. The application of the transducer to a string can then possibly finish with the concatenation of such an output string to the usual output. Such transducers are called *sub-sequential transducers*.

The language processing applications often require more general extensions due to ambiguities. For example, the ambiguities encountered in language, like, ambiguities of grammars, morphological analyzers, and of pronunciation dictionaries, cannot be taken into account when using sequential or subsequential transducers. These devices associate at most a single output to a given input. In order to deal with ambiguities, one can introduce *p-subsequential* transducers, namely transducers provided with at most p final output strings at each final state (here $p \geq 2$). Fig. 5.5 gives an example of a 2-subsequential transducer. Here, the input string $w = ba$, gives two distinct outputs *aba* and *abb*. Since one cannot find any reasonable case in language in which the number of ambiguities would be infinite, p-subsequential transducers appears to be sufficient for describing linguistic ambiguities. However, the number of ambiguities could be very large in some cases. Note that 1-subsequential transducers are exactly the subsequential transducers [Mohrimeh].

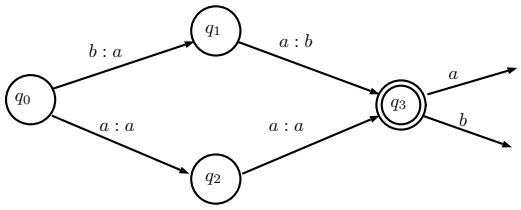

Figure 5.5: P-Subsequential Transducer ($P = 2$).

Composing of P-subsequential Transducers We know that transducers are devices that represent mappings from strings to strings. As such, they admit the composition operation defined for mappings, using which one can construct more complex transducers from simpler ones.

Consider that there are two transducers τ_1 and τ_2, such that these are in sequential order, first τ_1 and then τ_2, i.e., an input string s is applied to τ_1, and the output of that goes as input to τ_2. As a result of the application of $\tau_2 \circ \tau_1$, the string s can be computed by first considering all output strings associated with the input s in the transducer τ_1, then applying τ_2 to all of these strings. The output strings obtained after this application represent the result $(\tau_2 \circ \tau_1)(s)$. In fact, instead of waiting for the result of the application of τ_1 to be completely given, one can gradually apply τ_2 to the output strings of τ_1 yet to be completed. This is the basic idea of the composition algorithm, which allows the construction of a single transducer $\tau_2 \circ \tau_1$ given transducers τ_1 and τ_2, and input s is applied to that transducer. That way, τ_1 and τ_2 will both be processing their input in parallel at different stages.

5.5 p-Subsequential transducers for Language processing

In the above, we have briefly mentioned some of the theoretical and computational properties of sequential and p-subsequential transducers. These devices are used in many areas of computational linguistics, where determinization algorithm can be used to obtain a p-subsequential transducer, and a minimization algorithm to reduce the size of the p-subsequential transducer used. The algorithms for subsequential transducers that perform composition, union, and equivalence are also useful in many applications. Following are some of the applications in computational linguistics.

Finite-state transducers are automata in which transitions are labeled with both an input and an output

symbol. Transducers have been used successfully to create complex systems in many applications such as text and language processing, speech recognition and image processing. The time efficiency of such systems is substantially increased when subsequential transducers, i.e. finite-state transducers with deterministic input, are used. Subsequential machines can be generalized to p-subsequential transducers which are transducers with deterministic input with p, ($p \geq 1$), final output strings. This generalization is necessary in many applications such as language processing to account for finite ambiguities.

Not all transducers admit equivalent p-subsequential transducers however. We will discuss the characterization of p-subsequentiable transducers, i.e. transducers that admit equivalent p-subsequential transducers. This characterization is based on the twins property and leads to an efficient algorithm for testing p-subsequentiability. More generally, our results show the equivalence of the following three fundamental properties for finite-state transducers: determinizability in the sense of a generalized algorithm, p-subsequentiability, and the twins property.

This can also be viewed as a generalization of the results known in the case of functional transducers: determinizable functional transducers are exactly those that admit equivalent subsequential transducers. We generalize these results by relaxing the condition on functionality: determinizable transducers are exactly those that admit equivalent p-subsequential transducers and exactly those that admit the twins property [cyrmehry].

5.5.1 Representation of Dictionaries

Very large-scale dictionaries can be represented using p-subsequential transducers because the number of entries and that of the ambiguities they contain are finite. The corresponding representation offers fast look-up since the recognition does not depend on the size of the dictionary but only on that of the input string considered. The minimization algorithm for sequential and p-subsequential transducers allows the size of these devices to be reduced to the minimum. Experiments have shown that compact and fast look-up representations for large natural language dictionaries can be efficiently obtained.

5.5.2 Compilation of Morphological and Phonological Rules

Similar to dictionaries, context-dependent *phonological* and *morphological* rules can be represented by finite-state transducers. Most phonological and morphological rules correspond to p-subsequential functions. Often, the result of the computation is not necessarily a p-subsequential transducer, but it can often be determined using the determinization algorithm for p-subsequentiable transducers. This considerably increases the time efficiency of the transducer. It can be further minimized to reduce its size. These observations can be extended to the case of weighted rewrite rules.

5.5.3 Syntax

Finite-state machines are also commonly used to represent local syntactic constraints. Linguists can conveniently introduce local grammar transducers that can be used to disambiguate sentences. The number of local grammars for a given language and even for a specific domain can be large. The local grammar transducers are mostly p-subsequential. The determinization and minimization can then be used to make the use of local grammar transducers more time efficient and to reduce their size. Since p-subsequential transducers are closed under composition, the result of the composition of all local grammar transducers is a p-subsequential transducer. The equivalence of local grammars can also be tested using the equivalence algorithm for sequential transducers.

Because the sequential transducers are so time and space efficient, they are being used increasingly in natural language processing as well as in other connected fields. In the following, we consider the case of string-to-weight transducers.

5.6 Formal representation of Subsequential String-to-Weight Transducers

We will consider string-to-weight transducers, i.e., transducers with input strings and output as weights. These transducers are used in domains, such as *language modeling*, *representation of word*, or *phonetic lattices*, etc [Mohrimeh02]. The usages can be done in the following way: one reads and follows a path in a directed graph corresponding to a given input string and outputs a number obtained by combining the weights along this path. In most applications to natural language processing, the weights are simply added along the path, since they are interpreted as (negative) logarithms of probabilities. In case the transducer is not sequential, that is, when it does not have a deterministic input, one proceeds in the same way for all the paths corresponding to the input string. In natural language processing, specifically in speech processing, one keeps the minimum of the weights associated to these paths. This corresponds to the *Viterbi* approximation in speech recognition or in other related areas for which hidden Markov models (HMM's) are used. In all such applications, one looks for the best path, i.e., the path with the minimum weight [Mohrimeh02].

In addition to the output weights of the transitions, string-to-weight transducers are provided with initial and final weights. For instance, when used with the input string ab, the transducer in Fig. 5.6 outputs: $4+2+3+7=16$, the 4 being the initial and 7 the final weight. For string b, the output is $4+6+7=17$.

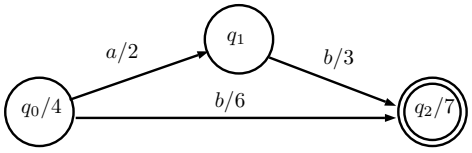

Figure 5.6: Sample Sequential string-to-weight Transducer.

Definition 5.2 String-to-weight transducer. *A string-to-weight transducer T is defined by $T=(Q,\Sigma,I,F,E,\lambda,\rho)$ with:*

Q a finite set of states,

G the input alphabet,

$I \subseteq Q$ the set of initial states,

$F \subseteq Q$ the set of final states,

$E \subseteq Q \times \Sigma \times R_+ \times Q$ a finite set of transitions,

λ the initial weight function mapping input I to set of output weights R_+, and

ρ the final weight function mapping set of final states F to R_+.

For the transducer T, a transition (partial) function $\delta : Q \times \Sigma \to 2^Q$ can be defined as:

$$\forall(q,a) \in Q \times \Sigma, \delta(q,a) = \{q' \mid \exists x \in R_+ : (q,a,x,q') \in E\}$$

and an output function $\sigma : E \to R_+$ can be defined as:

$$\forall t = (p,a,x,q) \in E, \sigma(t) = x$$

where $\forall t$ stands for "for all transitions t.

A path π in T from $q \in Q$ to $q' \in Q$ is a set of successive transitions from q to q', given as:

$$\pi = ((q_0,a_0,x_0,q_1),...,(q_{m-1},a_{m-1},x_{m-1},q_m)),$$

with,

$$\forall i \in [0,m-1], q_{i+1} \in \delta(q_i,a_i).$$

The extended definition of σ to a path in graph is:

$$\sigma(\pi) = x_0x_1...x_{m-1}.$$

Let us denote by $\pi \in q \rightsquigarrow q'$ the set of paths from q to q' labeled with the input string w. The definition of δ can be extended to $Q \times \Sigma^*$ as:

$$\forall(q,w) \in Q \times \Sigma^*, \delta(q,w) = \{q' : \exists \text{ path } \pi \text{ in } T, \pi \in q \rightsquigarrow q'\}.$$

Similarly, the δ can be extended to $2^Q \times \Sigma^*$, by:

$$\forall R \subseteq Q, \forall w \in \Sigma^*, \delta(R,w) = \bigcup_{q \in R} \delta(q,w). \tag{5.3}$$

5.7 English Language Morphology

Morphology is study of, how the words are constructed. Construction of English language words through attachment of prefixes and suffixes (both together called *affix*) are called *concatenative morphology*, because a word is composed of number of morphemes concatenated together. A word may have more than one affix, for example rewrites (*re+write+s*), unlikely (*un+like+ly*), etc. There are broadly two ways to form words using morphemes:

1. *Inflection:* Inflectional morphology form the words using the same group word stem, e.g., write+*s*, word+*ed*, etc. The Table 5.1 shows the words constructed using inflective morphology.
2. *Derivation:* Derivations morphology produces a word of different stem, for example computerization (noun) from computerize (verb) – the words belong to different groups.

Table 5.1: Inflectional Morphology.

Type	Regular nouns	Irregular nouns
Singular	cat, thrush	mouse, ox
Plural	cats, thrushes	mice, oxen

The examples of *regular verbs* are walk, walks, walking, walked. Similarly, *irregularly inflected* verbs are: "eat, eats, eating, ate, eaten, catch, catches, cut, cuts, cutting, caught," etc.

The derivation is a combination of word stem with *grammatical morpheme*, usually resulting in a word of different class. For example, formation of nouns from verbs and adjectives. The Table 5.2 shows the examples of derivational morphology.

Table 5.2: Derivational Morphology.

Suffix	Base verb/adjective	Derived Noun
-action	computerize (V)	Computerization
-ee	appoint (V)	appointee
-er	kill (V)	killer
-ness	fuzzy (A)	fuzziness

5.8 Morphology and Finite-state Transducers

To know the structure about a word when we perform the *morphological parsing* for that word. Given a *surface form* (input form), e.g., "going" we might produce the parsed form: *verb-go + gerund-ing*. Morphological parsing can be done with the help of *finite-state transducer*. A *morpheme* is a meaning bearing unit of any language. For example,

fox: has single morpheme, *fox* and,

cats: has two morphemes, *cat, -s*.

Similarly, eat, eats, eating, ate, eaten have different morphemes.

Some examples of mapping of certain words and corresponding morphemes are given in the Table 5.3. The mapping of input and output correspond to the input and output of finite state machines.

In speech recognition, when a word has been identified, like cats, dogs, it becomes necessary to produce its morphological parsing, to find out its true meaning, in the form of its structure, as well to know how it is organized. These include the features, like N (noun), V (verb), specify additional information about the word stem, e.g., $+N$ means that word is noun, $+SG$ means singular, +PL for plural, etc.

We require following databases for building morphological parser:

1. **Lexicon:** List of stems, and affixes, plus additional information about them, like *+N, +V*.

Table 5.3: Mapping of input word to Morphemes.

Input Words	Morphological parsed output
cats	cat *+N +PL*
cat	cat *+N +SG*
cities	city *+N +PL*
geese	goose *+N +PL*
goose	goose *+V +3SG*
caught	catch *+V +PAST-Part*

2. **Morphotactics rules:** Rules about ordering of morphemes in a word, e.g. *-ed* is followed after a verb (e.g., worked, studied), *un* (undo) precede a verb, for example, unlock, untie, etc.
3. **Orthographic rules** (spelling): For combining morphemes, e.g., city+ *-s* gives cities and not citys.

We can use the *lexicons* together with *morphotactics* (rules) to recognize the words with the help of finite automata in the form of stem+affix+part-of-speech (N, V, etc). The Fig. 5.7 shows the basic idea of parsing of nouns using morphological parsing. Recognition of nouns by FA is subject to reaching to final state (marked by double circle in figure) of FA. Table 5.4 shows some examples of regular and irregular nouns.

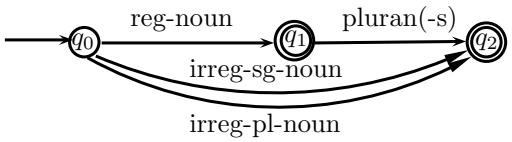

Figure 5.7: Morphological Parsing of nouns.

Table 5.4: Regular and Irregular nouns.

Reg-noun	Plural	Irreg-noun	Irreg-sg-noun
fox	-es	goose	geese
cat	-s	sheep	sheep
dog	-s	mouse	mice

A similar arrangement is possible for *verb* morphological parsing (see Fig. 5.8, and Table 5.5). The lexicon for verbal inflection have three stem classes (*reg-verb stem, irreg-verb stem,* and *irreg-past-verb*), with affix classes are: *-ed* for past and participle, *-ing* for continuous, and 3rd person singular has *-s*.

Adjectives can be parsed in the similar manner like, the nouns and verbs. Some of the adjectives of English language are: big, bigger, biggest, clean, cleaner, cleanest, happy, unhappy, happier, happiest, real, really, unreal, etc. The finite automata in Fig. 5.9 is showing the morphological parsing for adjective words.

At the next stage, the lexicon can be expanded to sub-lexicons, i.e, individual letters, to be recognized by the finite automata. For example, regular-noun in Fig. 5.7 can be expanded to letters "*f o x*" connected by three states in a transition diagram. Similarly, the regular verb stem in Fig. 5.8 can be expanded by letters "*w a l k*", and so on, as shown in Fig. 5.10. Note that in the parsing of N, V, ADJ, and ADV discussed above, for the sake of simplicity we have not shown the transitions separated by colon (':"), however, the FST has two tapes as usual, for input and output.

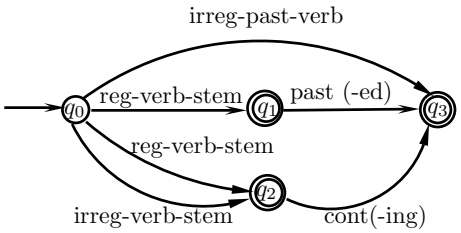

Figure 5.8: Morphological Parsing of verbs

Table 5.5: Regular and Irregular verbs.

Reg-verb	Past	Irreg-verb	Irreg-past-v	Cont.	3sg
walk	-ed	catch	caught	-ing	-s
fry	-ed	eat	ate	-ing	-s
talk	-ed	sing	eaten	-ing	-s

5.9 Finite State Transducers and Morphological Analysis

The objective of the morphological parsing is to produce output lexicons for a single input lexicon, e.g., like it is given in table 5.6. The second column in the table contains the stem of the corresponding word (lexicon) in first column, along with its morphological features, like, +N means word is noun, +SG means it is singular, +PL means it is plural, +V for verb, and pres-part for present participle. We achieve it through two level morphology, which represents a word as a correspondence between lexical level - a simple concatenation of lexicons, as shown in column 2 of Table 5.6, and a surface level as shown in column 1. These corresponds to two tapes of a finite state transducer.

The FST is a multi-function device, and can be viewed in the following ways:

- *Translator*: It reads one string on one tape and outputs another string on other tape, it may receive input "cats" on surface tape, and produce parsed output "cat +N +PL" on lexical tape. Alternatively, the role of input and output tape can be interchanged.
- *Recognizer*: It takes a pair of strings as two tapes and accepts/rejects based on their matching. Foe example, when both the contents is as shown in Fig. 5.11, then it accepts translation, if one of the tape is having different contents, then the FST rejects (no match).
- *Generator*: It outputs a pair of strings of that language, on two tapes along with yes/no result based on whether they are matching or not. Hence, acts as generator.
- *Relater*: It compares the relation between two sets of strings available on two tapes.

Table 5.6: Lexical Transformation table.

Input	**Parsed output**
cat	cat +N +SG
cats	cat +N +PL
geese	goose +N +PL
reading	read +V +Pres-part

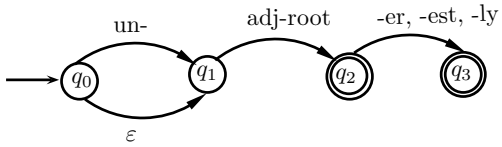

Figure 5.9: Morphological Parsing for adjectives.

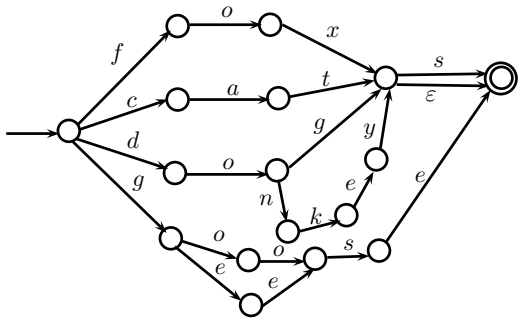

Figure 5.10: Morphological Parsing for noun words in details.

Like FSA (Finite State Automata) are isomorphic to regular expressions, the FSTs are isomorphic to *regular relations*. The FSTs are closed on the following relations:

1. *Union*: If R_1 and R_2 are relations on FST, then $R_1 \cup R_2$ is also a relation on FST.
2. *Composition:* If T_1 is FST from I_1 to O_1, and T_2 is FST from I_2 to O_2, then $T_2 \circ T_1$ is FST from I_1 to O_2.
3. *Inversion:* The FSTs are closed on *inversion*. A transducer T' (or T^{-1}) simply switches the input and output labels on each transition.

The composition operation is useful because it replaces two FST running in series by a single FST. The composition works as in algebra. Applying $T_2 \circ T_1$ to input sequence S is equal to applying T_1 to S, and then T_2 to result $T_1(S)$, i.e.,

$$T_2 \circ T_1(S) = T_2(T_1(S)) \tag{5.4}$$

Similarly, the composition is useful to convert a FST as *parser* to FST as a *generator*[2],[3].

In two level morphology, the lexical tape is composed of symbols from a in $a : b$ pairs, and the surface tape comprises the symbols from b in this pair. Hence, each symbol pair $a : b$ gives mapping from one tape to other tape. The symbols $a : a$ are called *default pairs* and written simply as a as shown in Fig. 5.12 shows as transitions: $q_0 - q_1$.

The Fig. 5.12 shows the transition diagram for FST with additional symbols $+SG$ (singular), $+PL$ (plural), corresponding to each morpheme. These symbols map to empty string (ε), as there are no corresponding symbols on output (surface) tape.

[2] *Parser*: A parser parse (convert) a word into its constituent components, e.g., "cats" is parsed into "cat +N +PL."
[3] *Generator:* Given "cat" as a lexicon for noun, and that its plural form +PL, use "cat +N +PL", generate the word "cats."

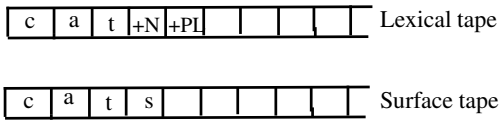

Figure 5.11: A FST.

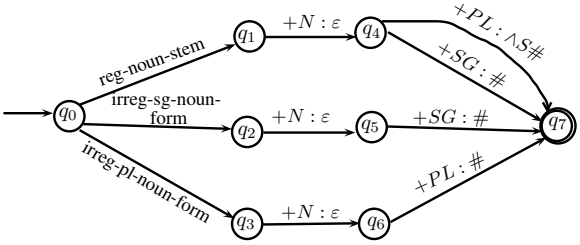

Figure 5.12: Morphological Parsing using FST.

The symbol # stands for boundary symbol. Typical example of mapping, e.g., in case of word "geese" (irregular noun), on surface tape will be parsed into *goose +N +PL* on lexical tape, and symbols on the arc joining states q_0 - q_2 are "g:g o:e o:e s:s e:e", which is written as "g o:e o:e s e". Since, there are five letters in the word, there will be five state transitions between q_0 - q_2. For regular noun, like fox, it will be "f:f o:o x:x". The surface form "geese" is mapped to lexical form "goose +N +SG" through *cascading* the FSTs, where two automata are run in series, i.e. output of first becomes input to next. This is what we discussed earlier as P-subsequential transducer.

Instead of cascading two transducers, we perform this job using *composition* operator. Composing the transducers in this way helps in taking many different levels of input and outputs, and converting them into a single two level transducer with one input and one output tape. A typical FST, which results for morphological parsing of "cat" is shown in Fig. 5.13, producing a mapping *c:c a:a t:t +N:ε +PL:^S#*. The *+PL* maps to ^*S*. The symbol ^ indicates the morpheme boundary, and # indicates the word boundary.

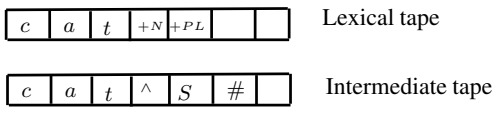

Figure 5.13: Morphological Parsing (Lexical and Intermediate tapes).

5.9.1 Orthographic Rules

We note that concatenating the morphemes can work to parse the words like "dog", "cat", "fox", but this simple method does not work when there is spelling change, like "foxes" is to be parsed into lexicons "fox +N +PL" or "cats" is to be parsed into "cat +N +3SG", etc. This requires introduction of spelling rules (also called orthographic rules).

To account for the spelling rules, we introduce another tape, called *intermediate tape*, which produces the output slightly modified, thus going from 2-level to 3-level morphology. Such a rule maps from intermediate tape to surface tape. For plural nouns, the rule states, "insert *e* on the surface tape just when intermediate tape has a morpheme ending in *x* or *z* or *s* and next morpheme is *-s*". The examples are *ox* to *oxes*, and *fox*

to *foxes*. The rule is stated as equation (5.5),

$$\varepsilon \rightarrow e/ \left\{ \begin{array}{c} x \\ s \\ z \end{array} \right\} \wedge - - - S\# \tag{5.5}$$

The equation 5.5 is called *Chomsky and Hall* notation. A rule of the form $a \rightarrow b/c - d$ means rewrite a as b, when it occurs between c and d. Since symbol ε is null, and it occurs between $\wedge$ and S on intermediate tape, therefore replacing ε (null) by e means inserting e between $\wedge$ and S. The symbol $\wedge$ indicates morpheme boundary. These boundaries are deleted by including the symbol pair $\wedge : \varepsilon$ in Fig. 5.12, the default pairs for the transducer $(I : O)$, i.e., in the graph, the symbol ':' indicates that 1st symbol is on intermediate tape and ε is on surface tape. The mapping of symbols shown in Fig. (5.14), is called *morphological parsing*. There are n number of FSTs, indicating that there are n number of rules encoded.

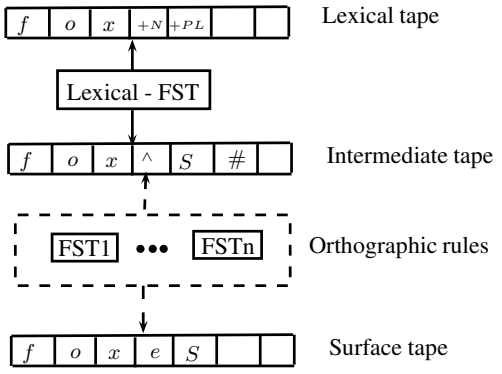

Figure 5.14: Morphological Parsing using 3-tape FSTs.

Using these multi-level FSTs in sequence between different tapes, as well as through parallel transducers for spelling checks, we are able to parse those words whose morphological analysis is simple. However, considering the sentence "The police books the right culprit", here it is not clear as per above rules that whether the lexical parser's output is "book +N +PL" or it is"book +V +3SG" ! However, to human it is not difficult to infer that it is the second option. This is due to the ambiguity in the word, which may be a noun or a verb, depending on its position in a sentence. This type of ambiguity is called *lexical ambiguity*, and is the subject of later discussions.

Review Questions

1. Explain the difference between sequential and subsequential transducers.

2. What are the roles of lexical tape and surface tape in a finite state transducer? Explain.

3. Can we use Chomsky-Hall rule for verb, adverb, adjective?

Exercises

1. Draw the minimal deterministic automaton that accepts:

$$\{CV, CVC, VC, V\}(.\{CV, CVC, VC, V\})^*$$

2. What is space and time complexity of FA in following?
   (a) Lexical Analysers
   (b) Morphological Parsing
3. Define the field and scope of computational linguistics.
4. Explain with example, how a P-subsequential transducer can be used for resolving ambiguity.
5. Explain, how a FA can be useful in Computational Morphology? Give examples.
6. Demonstrate the morphological parsing for any five English language words.
7. Give your idea, as how you will translate the English language words to Hindi. Give five examples.
8. What are the functions of following databases used by Morphological parser?
   (a) Lexicon
   (b) Morphotactic rules
   (c) Orthographic rules
9. Justify the statement: "FSTs are isomorphic to regular relations."
10. What are the general applications of FSTs? Explain each application with at least one example, in details.
11. Give some idea to resolve the morphemes to similarly pronounced words, like "bread", "dread", "red", "read", "raid."
12. Write a program /algorithm to produce your own version of *stemer*, which produces stem words from some nouns only.
13. Explain the *Morphological parsing* of following words using two level tapes FSTs: horse, donkey, house, cat, men.
14. What are the Sequential Transducers? How they can be used for language modelling? Explain.
15. Explain the application of sub-sequential and p-subsequential transducers for ambiguity resolution in languages and grammars.
16. How the dictionaries can be looked / searched at speed using P-subsequential transducers? Give an example, as well as an algorithm for the same.
17. Suggest a block diagram / flow-diagram, as how the large size dictionaries can be implemented and searched using P-subsequential transducers?
18. What is difference between two-level and three level FSTs, used for morphological parsing? For the following words, which type of FST is required?

    book, booking, eat, ate, eaten

19. (a) What is Chomsky-and-Hall(CH) equation? How is is useful for defining orthographic rules? Explain.

    (b) Which of the following words' morphological parsing is generated by CH equation?

    cats, dogs, goose, foxes, oxes.

20. Can all the languages be modelled with Finite state automata?

21. How can you prove that a language is regular?

22. How many levels ("tapes") are used for morphological parsing? What are their names? Give an example of how these tapes would like when parsing following words:

    (a) Mice

    (b) Mouse

    (c) Churches

    (d) Temples

    (e) Mosques

23. Write the algorithms for following:

    (a) Parsing of regular nouns.

    (b) Parsing of irregular nouns.

    (c) Parsing of regular verbs.

    (d) Parsing of irregular verbs.

    (e) Parsing of adjectives.

24. What are the databases required for morphological parsing? Describe each, with its possible structure.

25. Perform the study of information available in any English dictionary, and list the information content in that, particularly that which is helpful for morphological parsing.

    Note: In the next four exercises, it is aimed to parse nicknames such as *Robbie* (for Robert) or *Patty* (for Patricia).

26. Build a lexicon automaton FSA1 that recognizes the first names followed by "+NN"(for nick name).

27. (a) Build a FSA that represents the *morphotactics* behind the nicknames, i.e., abbreviate and append by *ie* or *y*. The FSA should recognize the strings such as "Robˆy#".

    (b) Convert this automaton to a finite state transducer FST-A which maps lexical level, e.g., "Robert +NN", to an intermediate level, e.g., "Robˆy#".

28. (a) Give an *orthographic* rule which doubles the consonant before *y* or *ie*.

    (b) Implement this rule as a finite state transducer FST-B.

29. How will you compose FSA, FST-A and FST-B to build a morphological parser for these nicknames?

References

[1] Jurafsky D and Martin J, *Speech and Language Processing, 3rd Ed.*, Pearson India, isbn: 3257227892, Nov. 2005.

[2] Mohri M, *Finite-state Transducers in Language and Speech Processing*, *Comput. Linguist.*, Vol. 23, No. 2, June 1997, issn: 0891-2017, pp.269–311, MIT Press, Cambridge, MA, USA

[3] Mohri M, et al. *The Design Principles of a Weighted Finite-State Transducer Library*, Preprint submitted to Elsevier Preprint, 5 May 2002, pp. 1-19

[4] Cyril Allauzen and Mehryar Mohri, *p-Subsequentiable Transducers AT&T Labs*, J. M. Champarnaud and D. Maurel (Eds.): CIAA 2002, LNCS 2608, pp. 24-34, 2003. Springer-Verlag Berlin Heidelberg 2003.