CSME 206A – Natural Language & Speech Processing **Spring Semester**

Lecture 10: Finite automata and Morphological Analysis

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

10.1 Finite state automata and NLP

A formal language is a set of strings (i.e., sentences) made up by concatenating symbols (characters or words) drawn from a finite alphabet. If a language has only a finite number of sentences, then a complete description of the set of sentences can be given simply by presenting a finite list of all the sentences. But, if the language infinite (contains infinite number of sentences), like all the natural languages, then some sort of recursive or iterative description is to be provided to describe the sentences. This description is called grammar – a set of pattern-matching rules that can be applied either to produce the sentences in the language one after another or else to recognize whether a given string belongs to the set. The description may also be provided by specifying an automaton, a mechanistic device that also operates either to produce or recognize the sentences of the language.

A regular language can be described by grammars in various notations that are known to be equivalent in their expressive power of the language, and they each can be used to describe all and only the regular languages. The most common way of specifying a regular language is by means of a *regular expression* – a formula that indicates the order in which symbols can be concatenated, specifies the alternative possibilities at each position, and whether substrings can be arbitrarily repeated. The regular expression $(c^*b(a|b|c)^*|\varepsilon)c$ denotes the regular language described above. In this notation, concatenation is represented by order sequence in the formula, alternatives are separated by vertical bars, asterisks (often called the Kleene closure operator $*$) indicate that strings satisfying subexpression just before $*$, can be arbitrarily repeated, and ε denotes the *empty string* or null-string (string containing no elements).

The regular languages are also exactly those languages that are accepted by finite-state automata (FSA), which are presented in the next section. The FSA, shown in Fig. fig:fsmdig accepts the language $(c^*b(a|b|c)^*|\varepsilon)c$.

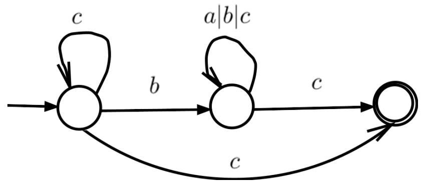

Figure 10.1: Finite state machine to accept strings $c^*b(a|b|c)^*|\Sigma)c$

Because of their mathematical and computational simplicity, regular languages and finite-state automata have been applied in many information processing tasks. Regular expressions are often used to specify *global search patterns* in word-processors and in operating-system utilities such as Unix' *grep*. The *lexical analysis* component of most modern programming language compilers is defined as a *finite-state atomata* that recognizes identifier classes, punctuation, numbers, braces, keywords, etc. But, complete and accurate natural language *syntax* and *semantics* lie beyond the power of finite-state automata. However, the recent work has identified a number of important problems for which very efficient and effective finite-state solutions can be found.

Another set of problems require that strings be systematically transformed into other strings. For example, the negative prefix in in the abstract phonological representation such as *in+practical* must be realized with an assimilated *nasal* as in *impractical*, or the inflected form stopping must be mapped either to its stem stop (for use in information retrieval indexing) or to its morphological analysis *stop+PresentParticiple* as an initial step in further processing.

10.2 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) that 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 of S can be represented by a regular expression: $baa^{+}!$. However, we will prefer it to be represented by a string $/baa{+}!/$ – a format of its pronunciation. The sounds in set S can be recognized by a FA shown in the Fig. 10.2.

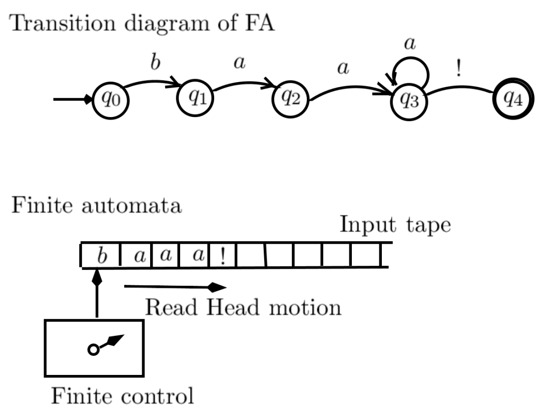

Figure 10.2: Finite Automata to recognize Sheep-talk

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.

Table 10.1: Transition table for FA in Fig. 10.2

State	State for Input a	State for Input b	State for Input !
q_0	-	q_1	-
q_1	q_2	-	-
q_2	q_3	-	-
q_3	q_3	-	q_4
q_4	-	-	-

The tape is considered as array indexed by *index* variable. For accepting by a FA it is necessary that read head has completely read the input, and the current state of the FA is the final state. Until this dual condition is not satisfied, the read head is advanced to next position by increasing the index and the state also changes as per the FA. When input is exhausted but *accepting* state (q_4) is not reached, the input is rejected. Note that the transition table 10.1 completely describes the FA shown in Fig. 10.2.

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

```
index ← initial-position-on-tape
while True do
  if end of input then
    if state== accept then
      return accept
    else
      return reject
    end if
  else
    if transition-table[state, tape[index]] == empty then
      return reject
    else
      state ← transition-table[state, tape[index]]
      index++
    end if
  end if
end while
```

The recognition of a string by FA is like searching a tree. Considering the recognition of some string, say of 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 time complexity remains $O(2^n)$ for both the Breadth First Search (BFS) as well as Depth First Search (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.

In case of DFS, for above string, the space complexity is $O(2 \times n)$, for branching factor ($m = 2$) and length of string n. As a general case, it is $O(mn)$. However, due to combinatorial explosion of number of states generated (see Fig. 10.3), 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*), which have better space complexity, are used.

The search-tree in Fig. 10.3 shows that for input string of length $n(= 5)$, and size of alphabet $|\Sigma| = 2$, there are $2^5 = 32$ different search paths in this tree to search the strings. And hence, for input $|w| = n$, and $|\Sigma| = m$, there is combinatorial explosion of number of search paths.

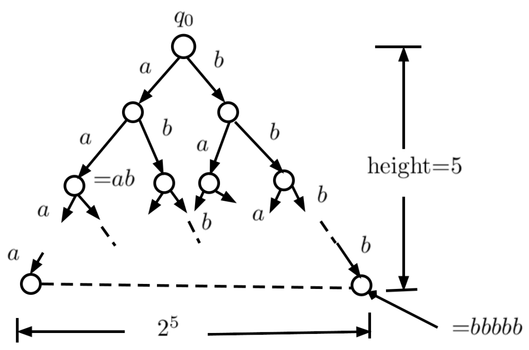

Figure 10.3: Binary search-tree of height 5

10.3 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. 10.4 shows the diagram where these (input and output) strings are shown on the transitions, separated by ":" sign. This FST acts as a translator, as it translates the input sentence "" Hello World" to "Hey there krc".

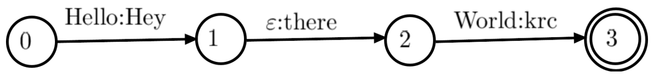

Figure 10.4: An FST as a translator

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

- 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 many possible applications: the translation of one string into another, generation of two strings, and relating one string to another,
- There is a defined state, called "start" state, and other as "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,

$$M = (Q, \Sigma, q_0, \delta, F) \tag{10.1}$$

where,

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

The input (I) and output (O) symbols, both include the empty string symbol ε. For $\Sigma = \{a, b, !\}$, corresponding to the sheep-talk 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:*[1] 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.

10.4 Sequential and P-subsequential Transducers

Sequential string-to-string transducers are used in 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[2]. In this section, the theoretical basis of the use of sequential transducers is described. A sequential transducer is defined as follows:

Definition 10.1 *A sequential transducer has deterministic input. At any state of such transducer, at most one outgoing arc is labeled with a given element of the alphabet.*□

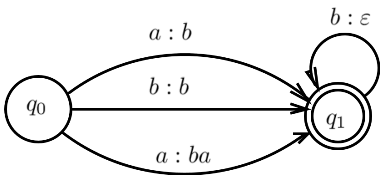

Figure 10.5: Sequential Transducer

Fig. 10.5 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

[1]A change in the form of a word (typically the ending part) to express a grammatical function or attribute such as tense, mod, person, number, case, and gender. Example: adding endings: -s to make a noun plural (book → books), -ed to put a verb in the past tense, -er to form the comparative form of an adjective.

[2]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 $p = 2$.

not necessarily to be deterministic. For example in Fig. 10.5, 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 10.2 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, $\delta : Q \times \Sigma \rightarrow Q$, and

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

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:

$$\forall s \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \quad \text{there is, } \delta(s, \varepsilon) = s, \delta(s, wa) = \delta(\delta(s, w), a); \text{ and}$$
$$\sigma(s, \varepsilon) = \varepsilon, \sigma(s, wa) = \sigma(s, w)\sigma(\delta(s, w), a). \tag{10.2}$$

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 $\sigma(q_0, w)$.

10.4.1 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. 10.6 gives an example of a 2-subsequential transducer. Here, the input string $w = ba$, gives two distinct outputs aba and abb with two strings at final state. 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.

A string $w \in \Sigma^*$ is accepted by the transducer τ_1 iff $\delta^*(q_0, w) \in F$ and output of transducer is $\sigma(q_0, a).\sigma(\delta(q_0, a), u)$, where $w = au$.

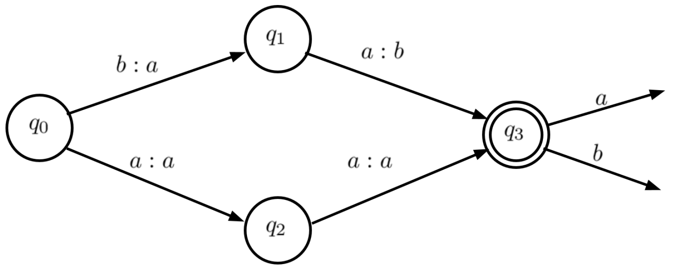

Figure 10.6: 2-Subsequential Transducer τ_1

10.4.2 Composing 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 (see Fig. 10.7). 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. This shows that subsequential and p-subsequential transducers are closed on concatenation. If $f : \Sigma^* \rightarrow \Delta^*$ be a p-subsequential transducer and $g : \Delta^* \rightarrow \Omega^*$ be a q-subsequential transducer, then $g \circ f$ is pq-subsequential transducer. The transducers τ_1 and τ_2 corresponds to functions f and g, respectively, as shown in Fig. 10.7. Note that, pq is product term, e.g., if $|p| = 2$ and $|q| = 3$, the $p.q$ will have six output sequences.

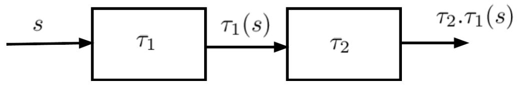

Figure 10.7: Concatenation of transducers τ_1 and τ_2

10.5 P-Subsequential Transducers for Language processing

Following are some of their applications in computational linguistics.

10.5.1 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.

Dictionaries and sets use *hash tables* in order to achieve their $O(1)$ look-ups and insertions. This efficiency is the result of a very clever usage of a hash function to turn an arbitrary key (i.e., a string or object) into an index for a list. The hash function and list can later be used to determine where any particular piece of data is right away, without a search. By turning the data's key into something that can be used like a list *index*, we can get the same performance as with a list. In addition, instead of having to refer to data by a numerical index, which itself implies some ordering to the data, we can refer to it by this arbitrary key.

10.5.2 Compilation of Morphological and Phonological Rules

Like dictionaries are represented, the 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.

10.5.3 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.

10.6 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. 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[3].

[3]For example, one path of a sequential string-to-weight transducer may have output "please/3, give/4, me/5, a/2, call/5", which makes a sentence with weight as, "please give me a call/19". The other path of this transducer may have, "please/3, call/6, me/7, a/2, give/0", which makes a weighted sentence as, "please call me a give/18". Naturally the first sentence should

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.

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. 10.8 outputs: $4 + 2 + 3 + 7 = 16$, the 4 being the initial and 7 the final weight. For string b, the output is $4 + 6 + 7 = 17$.

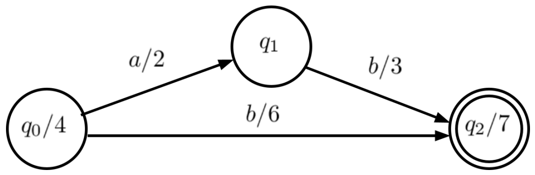

Figure 10.8: Sample Sequential string-to-weight Transducer

Definition 10.3 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,

Σ: the input alphabet,

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

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

R_+ : set of output weights,

$E \subseteq Q \times \Sigma \times R_+ \times Q$: a finite set of transitions (E: set of edges),

$\lambda : I \to R_+$, initial weight function mapping input I to set of output weights R_+, and

$\rho : F \to R_+$, 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.

be accepted.

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:

$$\sigma(\pi) = x_0 x_1 ... x_{m-1}.$$

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:

$$\forall R \subseteq Q, \forall w \in \Sigma^*, \delta(R, w) = \bigcup_{q \in R} \delta(q, w). \tag{10.3}$$

10.7 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 forms words using the same group word stem, e.g., write+*s*, word+*ed*, etc. The Table 10.2 shows the words constructed using inflective morphology.

   It is modification of a word to express different grammatical categories. Inflectional morphology is the study of processes, including affixation and vowel change, that distinguish word forms in certain grammatical categories. Inflectional morphology consists of at least five categories, from *language typology* and *syntactic description*, i.e., *grammatical categories* and the *lexicon*. However, the derivational morphology cannot be so easily categorized because derivation is not as predictable as inflection.

2. *Derivation:* Derivations morphology produce a words of different stem, for example computerization (noun) from computerize (verb) – the words belong to different groups.

   This morphology creates new lexemes, either by changing the syntactic category (part-of-speech) of a base or by adding substantial, nongrammatical meaning or both. On the one hand, derivation may be distinguished from inflectional morphology, which typically does not change category but rather modifies lexemes to fit into various syntactic contexts; inflection typically expresses distinctions like number,

Table 10.2: Inflectional Morphology

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

case, tense, aspect, person, among others. On the other hand, derivation may be distinguished from compounding, which also creates new lexemes, but by combining two or more bases rather than by affixation, reduplication, subtraction, or internal modification of various sorts. Although the distinctions are generally useful, in practice applying them is not always easy.

Table 10.3: Derivational Morphology

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

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 10.3 shows the examples of derivational morphology.

Before we can do the morphological analysis, the text sentence need to be tokenized, which through NLTK (Natural Language Toolkit) and Python language, as shown in the following example.

Example: Sentence tokenization using Python.

```
$ python
python 3.7.6 (default, Jan 8 2020)
>>> from nltk import tokenize
>>> text="This is a simple sentence, that any one can
     write."
>>> tokens=nltk.word_tokenize(text)
>>> print(tokens)
['This', 'is', 'simple', 'sentence', ',', 'that', 'any',
  'one', 'can', write', '.']
>>>
```

□

There are various approaches to morphology, as follows:

Morpheme Based Morphology Word-based morphology is often a word-and-paradigm approach. The theory takes paradigms as a central notion. Instead of stating rules to combine morphemes into word forms or to generate word forms from stems, word-based morphology states generalizations that hold between the forms of inflectional paradigms.

Lexeme Based Morphology Lexeme-based morphology usually takes what it is called an "item-and process" approach. Instead of analyzing a word form as a set of morphemes arranged in sequence, a word form is said to be the result of applying rules that alter a word-form or stream in order to produce a new one.

Word based Morphology Word-based morphology is usually a word-and -paradigm approach instead of stating rules to combine morphemes into word forms.

10.8 Application of Morphological Analysis

One application is in Text to Speech Synthesis. The morphological analysis can be used to reduce the size of lexicon and also plays an important role in determining the pronunciation of a homograph.

Other application is in Machine translation. Machine translation mainly helps the people who are belonging to the different communities and want to interact with the data present in the different languages. In lack of Morphological analysis, we need to store all the word forms, this will increase the size of database and will take more time to search. One more benefit of this analyzer is it provides the information of the word such as number, gender. This information can be used in target language to generate the correct form of the word.

Other application is in Spell Checker. A Spell checker is an application that is used to identify whether a word has been spelled correctly or not. Spell checker functionality can be divided into two parts: Spell check error detection and Spell check error correction. Spell check error detection phase only detects the error while Spell check error correction will provide some suggestions also to correct the error detected by Spell check error detection phase. One more advantage of using morphology based spell checker is that it can handle the name entity problem. If any word is not included in the lexicon, can be added easily.

Other application is in Search Engine. Morphological Analysis and Generation improves the result of the search engine. Suppose if a word is provided as a input but this word is not present in the lexicon, in that case Morphological analysis of that word is done.

10.9 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*,

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 10.4, these mapping of input and output corresponds to the input and output of finite state transducers.

Table 10.4: 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*

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.

10.9.1 Databases for Morphological Parsing

The following databases are needed for building a morphological parser:

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

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 rules). 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).

10.9.2 Morphological Parsing

The Table 10.5 shows some examples of regular and irregular nouns, and the Fig. 10.9 shows the basic idea of parsing of nouns using morphological parsing. The regular-noun, e.g., "cat" has corresponding three transitions c-a-t between q_0 and q_1 states.

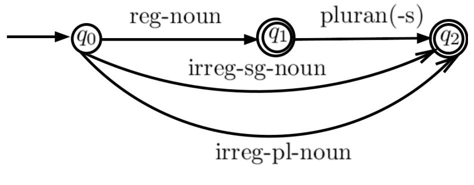

Figure 10.9: Morphological Parsing of nouns

Table 10.5: Regular and Irregular nouns.

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

Similarly, what is shown as a single transition for irregular-noun between q_0 and q_2 states, actually consists sequence of transitions, e.g., for the word "goose" there are five transitions, and for plural noun "geese" there are also five transitions. Such detailed transitions for parsing of noun words are shown in Fig. 10.10.

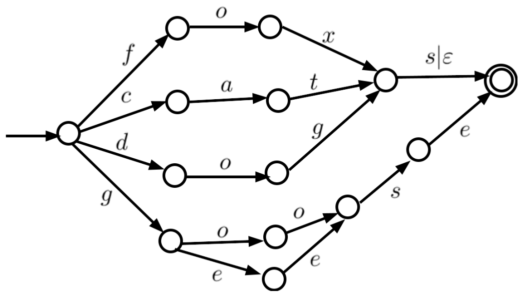

Figure 10.10: Morphological Parsing for noun words in details

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. 10.9 can be expanded to letters "*f o x*" connected by three states in a transition diagram. The three transitions are $f : f, o : o, x : x$. Similarly, the regular verb stem in Fig. 10.11 can be expanded by letters "*w a l k*", and so on, as shown in Fig. 10.10. 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.

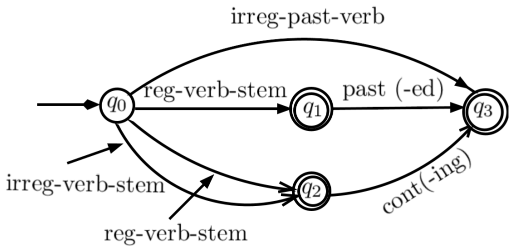

Figure 10.11: Morphological Parsing of verbs

A similar arrangement is possible for *verbs*' morphological parsing (see Fig. 10.11, and Table 10.6). The lexicon for verbal inflection have three stem classes (*reg-verb stem, irreg-verb stem,* and *irreg-past-verb*), with affix classes as: *-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. 10.12 is showing the morphological parsing for adjective words.

Table 10.6: 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	sang	-ing	-s

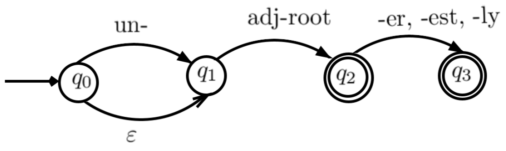

Figure 10.12: Morphological Parsing for adjectives

10.10 Morphological Analysis using Finite State Transducers

The objective of the morphological parsing is to produce output lexicons for a single input lexicon, e.g., like it is given in table 10.7. 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 10.7, and a surface level as shown in column 1. The two columns corresponds to two tapes of a finite state transducer 10.13.

Table 10.7: Lexical Transformation table.

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

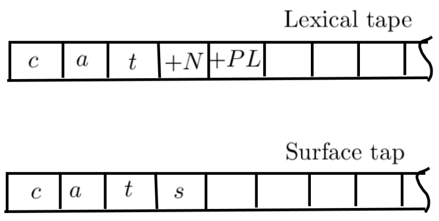

Figure 10.13: A FST

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 are as shown in Fig. 10.13, 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.

10.10.1 Closure properties of FSTs

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^{-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.,

$$T_2 \circ T_1(S) = T_2(T_1(S)) \tag{10.4}$$

Similarly, the composition is useful to convert a FST as *parser* to FST as a *generator*[4],[5].

10.10.2 Morphological parsing using FST

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 Fig. 10.14 shows transitions: $q_0 - q_1$.

The Fig. 10.14 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. The word, *stem* is basic word, for example, "cat" is stem for both its singular and plural forms.

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

[4]*Parser*: A parser converts a word into its constituent components, e.g., "cats" is parsed into "cat +N +PL."

[5]*Generator:* Given "cat" as a lexicon for noun, and that its plural form +PL, use "cat +N +PL", generate the word "cats."

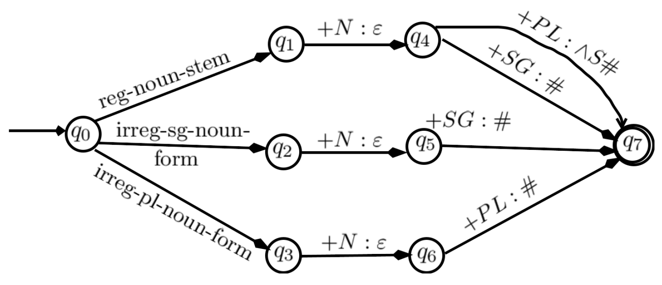

Figure 10.14: Morphological Parsing using FST

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. 10.15, producing a mapping *c:c a:a t:t +N:ε +PL:^S#*. The symbol sequence *+PL* maps to $^\wedge S$. The symbol ^ indicates the morpheme boundary, and # indicates the word boundary. In fact, the fist transducer T_1 will have *surface* tape as input and the *intermediate* tape as output. And, the second transducer (T_2) has intermediate tape as input and lexical tape as output tape. Hence, the composition $L = T_2 \circ T_1(S)$, where S source tape and L is content of lexical tape.

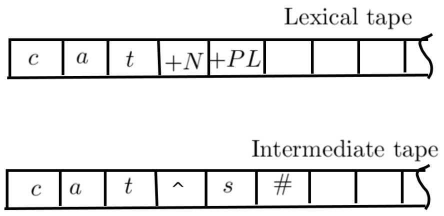

Figure 10.15: Morphological Parsing with Lexical and Intermediate tapes

10.10.3 Implementing 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*, *fox* to *foxes*, *rose* to *roses*, *doss* to *dosses*, etc.

Definition 10.4 *(Chomsky and Hall rule).The rule is stated as in equation (10.5),*

$$\varepsilon \rightarrow e/\left\{\begin{array}{c} x \\ s \\ z \end{array}\right\}\wedge - - - S\# \tag{10.5}$$

and 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.*$\square$

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. In the symbol $^\wedge$ indicates morpheme boundary. These boundaries are deleted by including the symbol pair $^\wedge : \varepsilon$ in Fig. 10.14, 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. (10.16), is called *morphological parsing.* There are n number of FSTs, indicating that there are n number of rules encoded.

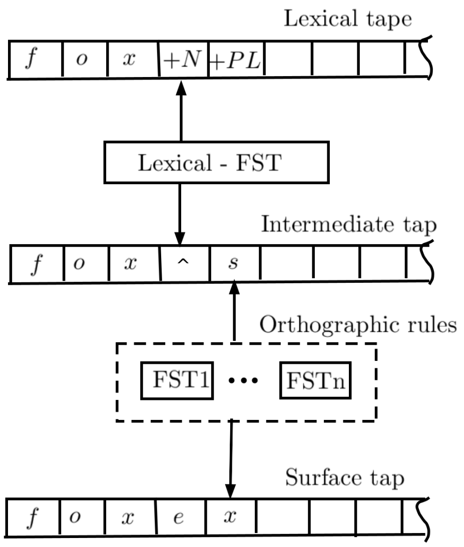

Figure 10.16: Morphological Parsing using 3-tape FSTs

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.