Notes on Regular Expressions, Automata and Grammars for Computational Biology

Venkatesh Choppella

October 6, 2007

1 Introduction

These notes supplement the material I covered in class on Sep 28th 2007 andon Oct 5th 2007, where I discussed regular expressions, deterministic finite state automata and grammars. You should refer to the book I used in class [1]. These notes are meant to complement the material in the transformational grammars chapter of the book. There is some difference in notation in the book and what is used here. Also, some topics, like regular expressions and the CYK algorithm which are not covered very well in the book, are presented in these notes.

Most of the material presented here is in an informal style. If you are looking to supplement your understanding with a more rigorous rendition of the subject matter, you may wish to take a course on automata theory offered in computer science departments, or consider studying any automata theory book, like [2] for example.

1.1 Alphabet

When biologists come talk to computer scientists or mathematicians about their problems, the latter usually examine the problem by first abstracting it or generalizing it. So, for example, instead of focusing on DNA sequences, the computer scientist will generalize to strings over an *arbitrary alphabet*. An alphabet is a set of *symbols*. Here are some examples.

1. DNA $= \{a, c, g, t\}$

2. Binary = $\{0, 1\}$

3. Numerals = $\{0, 1, ..., 9\}$

4. English = $\{A, B, C, ..., Y, Z\}$

1.2 Strings and Languages

A *string over an alphabet* Σ is a finite (possibly empty) sequence of symbols drawn from Σ. A string will also be called by other names: *sequence*, and also *word*. The empty string is usually represented by the symbol ϵ. Note that ϵ is *not* part of the alphabet.

For an alphabet Σ, Σ^* denotes the set of *all* strings over Σ. A *language over* Σ is a (possibly empty, finite, or infinite) subset of Σ^*. Note that each of the words of a language are of finite length, although a language may contain an infinite number of words.

Example 1.1 (Strings).

1. Strings over the alphabet DNA. ϵ, ac, aac, gtc, etc.

2. Strings over the alphabet English. ϵ, a, aa, $hello$, $gdfclu$. $gdfclu$ is a string over the English alphabet. It isn't, however, a word in the English language.

Example 1.2 (Languages). Here are examples of different languages over the English alphabet.

1. The English language. This is defined, let us say, as the set of words in a 'complete' English dictionary.

2. The language $\{aerious, abstemious, arsenious\}$ What's special about these words?

3. A small language of gibberish words $\{aa, oiuoiu, adkla, lklkjl\}$.

4. The empty language $\{\}$.

5. The language $\{\epsilon\}$ containing the empty word.

6. The set of words $\{\epsilon, a, aa, aaa, aaaa, \ldots\}$, *ad infinitum*. Although the set is infinitely big, each of its words is of finite length.

Here are examples of different languages over the DNA alphabet.

1. The set of all possible DNA sequences, including the empty one. This is an excellent example of the kind of generalizations mathematicians like to make and that biologists find intriguing.

2. The set of DNA sequences $\epsilon, acgt, aagt, tgcc$.

3. The set of all DNA sequences that are 3×10^9 in size or smaller.

4. Let Σ denote the DNA alphabet. The set $\{x \in \Sigma^* | \ |x| \geq 3 \times 10^9\}$. This is just a mathematically abstruse way of identifying the set of all DNA sequences whose length is greater than or equal to 3×10^9.

2 Operations on Languages

We have seen examples of different languages. One way to represent a language is to list out its elements. This can be painful, and impossible to do if the language is infinite in size. Another way is to use English, like we did in the first DNA example above. But this often leads to ambiguity. Is there a rule or a *formula* we can use to represent languages? Before we do that, however, we define *operations* over languages. There are three operations we are interested: union, concatenation and repetition.

2.1 Union of Languages

The first is the notion of *union* of two languages. The union of two languages is just the union of all elements of either language. The second is the notion of *concatenation of languages*. Let's say L_1 and L_2 are two languages. Then L_1L_2 is defined as the set of all strings obtained by pairwise concatenating every string of L_1 with every string of L_2.

Example 2.1. Consider the alphabet $\Sigma = \{a, b\}$. Suppose $L_1 = \{\epsilon, aa, b\}$. $L_2 = \{bb, ab\}$. Then $L_1L_2 = \{bb, ab, aabb, aaab, bbb, bab\}$.

2.2 String Concatenation

Concatenation is a fundamental operation over strings. If s and s' are strings, then ss', the concatenation of s and s' is just the string obtained by first writing down s_1 and immediately following that s_2. This is best explained by an example. The concatenation of ab with aba is $ababa$. Note that any

string s concatenated with ϵ, the empty string (or ϵ concatenated with s) results in the same string s. For example ϵ concatenated with aba results in aba.

2.3 String repetition

Let L be a language. Then L^2 is the language LL. For example, if $L = \{a, ab\}$. Then $L^2 = \{aa, aab, aba, abab\}$. It's not difficult to guess what L^3 is and how it is calculated, even though the computation is somewhat cumbersome.

$$L^3 = \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\}$$

.

We define L^1 to be L and L^0 to be $\{\epsilon\}$. It should not be difficult to generalize the idea to L^n, for any $n \geq 0$. Now consider the language $L^0 \cup L^1 \cup L^2 \cup \ldots L^n \cup \ldots$ ad infinitum. The result is a big language, infact infinite, unless L is empty or contains only ϵ. We denote this language by L^*. Intuitively L^* consists of all words built by concatenating any (zero or more) finite number of words in L. L^* is called the *Kleene closure* of L, so named after the 20th century logician Stephen Kleene. We will call L^* the *repetition* of L. Repetition builds words of L^* by repeatedly concatenating words drawn from L.

Example 2.2. Let $L = \{a\}$ be a language over the alphabet $\{a, b\}$. Then $L^* = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$. Notice the language consists of repetitions of the symbol a.

Let $L = \{a, ab\}$.

$$L^* = \{\epsilon, a, ab, aa, aba, aaa, abaa, \ldots\}$$

3 Regular Expressions

We now have the notion of a language, and three operators on languages: union, concatenation and repetition. Using these, we can construct expressions to denote more languages, eg., $(\{a\} \cup \{b\})(\{ab, bba\}^*)$. This notation is slightly cumbersome (the curly braces and the $\cup$), so we use a simpler notation to represent language expressions. This notation of regular expressions, or *regexps* for short, to specify languages.

Let Σ be an alphabet. *regular expressions* are expressions used to represent languages over Σ according to the following rules.

1. For each $a \in \Sigma$, the regexp a denotes the language $\{a\}$.
2. If regexps r_1 and r_2 denote languages L_1 and L_2, then $r_1 + r_2$ denotes the language $L_1 \cup L_2$.
3. If regexps r_1 and r_2 denote languages L_1 and L_2, then $r_1 r_2$ denotes the language $L_1 L_2$.
4. If regexps r denotes the language L, then r^* denotes the language L^*.

The best way to understand regular expressions is to consider some examples.

Example 3.1 (Regular Expressions). Consider the alphabet consisting of symbols a and b. We consider regular expressions representing languages over this alphabet.

1. The regexp a denotes the language $\{a\}$.
2. b denotes the language $\{b\}$.
3. $a + b$ denotes the language $\{a, b\}$.
4. ab denotes the language $\{ab\}$.
5. $ab(a + b)$ denotes the language $\{aba, abb\}$.
6. a^* denotes the language $\{\epsilon, a, aa, aaa, \ldots\}$.
7. ab^* denotes the language $\{ab, ab, abb, abbb, \ldots\}$.
8. $(ab)^*$ denotes the language $\{\epsilon, ab, abab, ababab, \ldots, \}$.

4 Finite State Machines

Suppose you're given a language, say L, and a sequence s. How do you determine if s is in L? This is called the language *recognition* problem. Why is language recognition interesting? Because language sequence pattern matching problem can be reduced to language recognition. For example, suppose you're looking for a pattern specified by a regular expression in a target sequence, then the regular expression itself denotes a language L. The string matching problem then reduces to recognizing subsequences in the pattern that are elements of L.

A *deterministic finite state machine* is a mathematical device, that can be implemented as a program, or even as hardware (or perhaps even wetware) which determines if a string belongs to a particular language.

A finite state machine, also called a *deterministic finite automaton* or DFA, consists of *states*.

The automaton sequentially reads one symbol of the input string at a time and makes a *transition* to another state based on *transition rules*. A transition rule specifies to which state the machine goes on a particular input symbol. For some states and some inputs, a transition may *not* be defined, in which case the machine is said to be *stuck*, presumably because it doesn't know which state to go to next. Some of the states of the machine are designated as *accept states*.

Here is how the DFA runs on a particular string input. At any given time, the DFA is in a particular *configuration* $\langle q, s\rangle$, which consists of the DFA's state q and the remaining input string s. The DFA starts from the initial configuration $\langle q_0, t\rangle$. The DFA proceeds from configuration to configuration until it halts. From any given configuration $\langle q, s\rangle$, the machine examines the rest of the input s:

1. If s is empty,
   (a) if q is an accepting state, then the DFA is in an *accepting configuration*. It then halts and declares that it *accepts* the input t.
   (b) if q is not an accepting state, then the DFA is in a *non-accepting configuration*. In this case DFA halts *without* accepting the input t.

2. If s is non-empty, then the DFA examines its first symbol, say x. Let the rest of s be the string y.

   (a) if there is a transition defined from q to another state q' on x, the DFA proceeds to the new configuration $\langle q', y\rangle$.

   (b) if there is no transition defined from q on the symbol x, then the machine is said to be in *stuck configuration*. In this case also, then the DFA halts *without* accepting the input string t.

Thus the DFA *accepts* the input string t if and only if the machine the machine starts with the configuration $\langle q_0, t\rangle$ and ends up in an accepting configuration, i.e, a configuration $\langle q', \rangle$, where q' is an accepting state.

The language *recognized* by the machine is defined to be exactly the set of strings that the machine accepts.

Suppose you want to recognize the language $\{aa\}$ over the alphabet $\{a, b\}$. Let's build a DFA to recognize this language. The DFA has a state called a *start* state where it waits for input. We will usually indicate the start state with q_0. When it sees an a as the next (first in this case) symbol of the string, it transits to the "saw an a" state. Let's call that state q_1. From this state, on reading another a, it transits to q_2, which essentially is the "saw the 2nd a" state. This state is marked to be an accepting state. No other transitions are defined.

Here's a trace of the machine configurations when the above DFA runs on the input aa.

$\langle q_0, aa\rangle$
$\langle q_1, a\rangle$
$\langle q_2, \epsilon\rangle$ accepting configuration
ACCEPT

Here's another trace for the same machine with input aab which is not accepted by the DFA.

$\langle q_0, aab\rangle$
$\langle q_1, ab\rangle$
$\langle q_2, b\rangle$ stuck configuration
REJECT

Another trace for the same machine with input a. Note the machine ends up in a non-accepting configuration.

$\langle q_0, a\rangle$
$\langle q_1, \epsilon\rangle$ non-accepting configuration
REJECT

4.1 Representing DFA's with transition rules

A DFA is identified by its states and the transitions of its states on symbols from an alphabet. We write each transition as a *rule*. The rule $q \xrightarrow{a} q'$ represents the transition of the DFA from state q to q' on reading the symbol a.

The state on the left side of the first rule is assumed to be the start state of the DFA.

Thus, the above DFA recognizing the langauage $\{aa\}$ is represented by the rules:

$$q_0 \xrightarrow{a} q_1$$
$$q_1 \xrightarrow{a} q_2$$

Example 4.1 (DFA for Human FMR-1 gene triplet repeat region). The book shows how the Human FMR-1 gene has a region of repeating triplets ggc. This region is easily captured by the regular expression $ggc(ggc)^*$.

A DFA for this is shown using the transitions:

$$q_0 \xrightarrow{g} q_1$$
$$q_1 \xrightarrow{g} q_2$$
$$q_2 \xrightarrow{c} q_3$$
$$q_3 \xrightarrow{g} q_1$$

Example 4.2. Can you guess what language the regular expression $(b^*ab^*a)^*$ denotes? Its the language of all strings over the alphabet $\{a, b\}$ such that each string in the language contains an *even number of a's*. The DFA for this contains only two states q_E and q_D representing the states of having read an even (and odd, respectively) number of a's so far in the input. The DFA is specified as a set of rules:

$$
\begin{aligned}
q_E &\xrightarrow{a} q_D \\
q_E &\xrightarrow{b} q_E \\
q_D &\xrightarrow{a} q_E \\
q_D &\xrightarrow{b} q_D
\end{aligned}
$$

Exercise 4.1. *For the above DFA's, run the DFA's on different inputs and trace the configurations in each case.*

5 Regular Grammars

Consider the *transition rules* in the example DFA

$$
\begin{aligned}
q_E &\xrightarrow{a} q_D \\
q_E &\xrightarrow{b} q_E \\
q_D &\xrightarrow{a} q_E \\
q_D &\xrightarrow{b} q_D
\end{aligned}
$$

These rules can instead be expressed as the following *production rules*. Each state is represented by a *non-terminal*. The symbols from Σ, i.e., a and b are called *terminal symbols*.

$$
\begin{aligned}
W_E &\longrightarrow aW_D \\
W_E &\longrightarrow bW_E \\
W_D &\longrightarrow aW_E \\
W_D &\longrightarrow bW_D \\
W_E &\longrightarrow \epsilon
\end{aligned}
$$

This is an example of a *regular grammar*. A regular grammar has two types of rules: $W \longrightarrow aW'$ or, $W \xrightarrow{\epsilon}$. Note the last rule of the grammar above; it is used to simulate the accepting state W_E. A shift from automata to grammars also implies a shift from recognition to *generation*. Each production rule can be considered a *rewrite* ruleGrammars specify how a string is generated from the start symbol (in this case W_E). Thus, instead of a DFA recognizing the string *aab*, say, we have the grammar generating the string

aab according to the following *derivation*:

$$\begin{aligned} W_E &\longrightarrow aW_D \\ &\longrightarrow aaW_E \\ &\longrightarrow aabW_E \\ &\longrightarrow aab \end{aligned}$$

The set of all strings derivable from a grammar starting from its start non-terminal, and using one or more rules is defined to be the language *generated* by the grammar.

We will not be formally constructing an algorithm for converting a DFA into Regular Grammar, but it should be intuitively quite clear how to do it.

$$\begin{aligned} W_E &\longrightarrow aW_D \mid bW_E|\epsilon \\ W_D &\longrightarrow aW_E \mid bW_D \end{aligned}$$

We sometimes employ | to denote all the alternative productions for a non-terminal.

6 Context-Free Grammars

Not all languages are generatable by a regular grammar and correspondingly, recognizable by a DFA.

For example, consider the language of balanced paranthesis, that is, the language over the alphabet $\{(,)\}$ consisting of strings whose parenthesis "balance". Elements of this language are (), (()), (())(), etc. This is a language that is very common in the syntax of programming languages. In sequence analysis, such parenthesis languages are used to capture RNA folding. It turns out that this language is not generatable by any regular grammar.

Instead, we need to consider *context-free grammars* in which the production rules are slightly more general: rules are allowed to be of the form $W \longrightarrow \alpha$, where α is any string consisting of non-terminals and terminals. The balanced parenthesis language may be specified by a CFG consisting of the following three productions:

$$S \longrightarrow SS \mid (S) \mid \epsilon$$

The book contains more examples of the use of CFG in biological sequence analysis.

We can think of a grammar as *generating* the strings of a language. The language $\mathcal{L}(G)$ of a grammar G is exactly the set of terminal strings generated by G. A context free language is one that is the language of a CFG.

6.1 Parsing

The derivation of a string using a grammar may be pictorially displayed as a tree with the start non-terminal as the root, and the string of terminals as the fringe of the tree (the sequence of leaves from left to right). There are examples of parse trees in the book, so I won't repeat them here.

6.2 Chomsky Normal Form

It turns out that every ϵ-free context free language can be generated by a special kind of context free grammar containing exactly two kind of productions: one in which a non-terminal N goes to a sequence N_1N_2 of two non-terminals, or goes to a terminal t. A grammar whose rules are of this form is said to be in *Chomsky Normal Form* (CNF). In other words the rules in grammar in CNF look like this

$$\begin{aligned} N &\longrightarrow N_1N_2 \\ N &\longrightarrow t \end{aligned}$$

6.3 CYK algorithm for CFL recognition

Let L be a context free language. How do we determine if a given input string is an element of L? Of course, if L is a CFL, then it is captured by a CFG. We could methodically generate all the strings of L from G and check if each of these strings is equal to the input string. Unfortunately, this could be a very long process. Instead, can we design an *algorithm* that takes the grammar and the string as input and determines if the input is generatable from the grammar *without* mechanically generating all the strings from the grammar?

The CYK algorithm, so called due to its inventors Cocke, Younger and Kasami, takes a grammar G in CNF and an input string s, and computes s is in $\mathcal{L}(G)$. The CYK algorithm we discuss here is described in [2]. In this section, we proceed to give a high-level description of the algorithm and an

intuitive understanding of why it works by relating it to the more familiar operation of matrix multiplication.

Example 6.1 (Hopcroft and Ullman[2], Example 6.7). The example grammar consists of the following productions

$$S \longrightarrow AB \quad (1)$$
$$S \longrightarrow BC \quad (2)$$
$$A \longrightarrow BA \quad (3)$$
$$A \longrightarrow a \quad (4)$$
$$B \longrightarrow CC \quad (5)$$
$$B \longrightarrow b \quad (6)$$
$$C \longrightarrow AB \quad (7)$$
$$C \longrightarrow a \quad (8)$$

The input string is *baaba*.

The algorithm computes, for each substring of s starting at position i and of length j, the set V_{ij} of non-terminals from which the string $s[i..(i+j-1)]$ may be derived. V_{1j}, therefore, is the set of non-terminals which derive the string $s[1..j]$. The CYK builds a matrix V_{ij}. A moment's thought should convince you that values of V_{ij} for which $i + j <= |s| + 1$ are relevant. This corresponds to the upper left half of the matrix. (See Figure 6.9 in the Hopcrof-Ullman book). Thus the CYK algorithm reduces to filling the entries of this triangular matrix.

Since $s[i..|s|]$ is just the string s, $V_{1\ |s|}$ is set of non-terminals that generates s. If S is in $V_{1|s|}$, then s is generatable by the grammar.

The first step of the algorithm is really simple. For any string s of size 1, say a, to find out all the non-terminals that derive that string, we simply inspect the rules of the grammar, identifying productions of the form $N \longrightarrow a$. This means that, given an input string s, $s[i..i]$ is a substring of size one at position i. The set V_{i1} of non-terminals deriving $s[i..i]$ is thus simple to compute. Thus, for the above example, $V_{i1} = B$, because $s[1..1]$, which is b is generated by B (see production 6).

To compute V_{ij} for $j > 1$ requires some more work. Before we give a formula for V_{ij}, let us define two kinds of operations:

Let u and v be sets of non-terminals of G. We define

$$uv = \{N | N \longrightarrow N_1 N_2 \text{ where } N_1 \in u, N_2 \in v\}$$

We consider each sequence N_1N_2 and see if it occurs as the right hand side of a production $N \longrightarrow N_1N_2$ of a grammar. If it does, then we include the left hand side N in uv. For example, in the example above, let $u = \{A, B, C\}$ and $v = \{B, C\}$. Then, we consider each of AB, AC, BB, BC, CB, and CC. The first, AB is the rhs of productions 1 and 7, whose lhs's are, respectively A, and C. Thus we add S and C to uv. AC, BB, CB do not occur in the lhs of any production, so we ignore them. But BC occurs as the rhs of production 2 whose rhs is S. S is already in uv. Also, CC occurs in the rhs of production 5, so we add B to uv. Thus $uv = \{S, B, C\}$.

Now, let U and V be *vectors* of sets of non-terminals. That is if $U = [u_1 \ldots ...u_n]$ is and $V = [v_1 \ldots v_n]$ for some n. Then the *product* is

$$UV^T = \bigcup_{k=1}^{n} u_k v_k$$

The transpose on V is takes so that the product resembles matrix multiplication. U can be thought of as row vector of dimensions $1 \times n$, and V^T as column vector of dimensions $n \times 1$. The big union operators builds the union of all the individual $u_k v_k$ sets.

Note the analogy with matrix dot product: If U and V were vectors of *numbers*, their *numerical dot product* would be

$$UV^T = \sum_{k=1}^{n} u_k v_k$$

We can now understand the computation of entry V_{ij} as the product of two vectors:

$$V_{ij} = [V_{i1} \ldots V_{i\ (j-1)}] \begin{bmatrix} V_{(i+1)(j-1)} \\ \vdots \\ V_{(i+j-1)\ 1} \end{bmatrix}$$

The row vector here is the column on top of the entry V_{ij}. The second (column vector) is the vector formed by traversing diagonally upward to the right.

The key thing to note is that if we populate the V matrix starting from the first row ($j = 1$) to the last row ($j = |s|$), then we can ensure that V_{ij} depends only on entries already populated in the table.

Therefore, the implementation should lend itself to an easy recursive algorithm. The algorithm in the Hopcroft-Ullman book is, however presented not as a recursive algorithm, but an iterative one. As an exercise, you should implement the CYK as a recursive algorithm and relate that to the one given in the Hopcroft Ullman book.

References

[1] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. *Biological Sequence Analsysi: Probabilistic Models of Protiens and Nucleic Acids.* Cambridge University Press, 1999.

[2] Hopcroft, J., and Ullman, J. *Automata, Languages and Computation.* Addison-Wesley, 1979.