Notes on the Theory of Computation

Part 1: Regular Languages

Coding Club

May 27, 2024

Mohammed Alshamsi
2021004826
mo.alshamsi@aurak.ac.ae

Department of Computer Science and Engineering
American University of Ras Al Khaimah
2023–24

Contents

List of Figures

1 Introduction

Welcome to the second set of "lecture notes" for the AURAK Coding Club. The idea is to introduce a useful topic that lends itself easily to coding applications. These topics are mathematical in nature, but most (if not all) of it will be intuitive material that doesn't require much background knowledge.

1.1 How to Read this Document

This is essentially just a short monologue about regular languages and related concepts. We'll go over the most basic definitions and results, and give some examples. This is by no means meant to give you a comprehensive introduction; it's just a quick tour to get you started. Check the bibliography if you want to learn more about the topic.

I'll be sure to bring up coding applications whenever it makes sense. Any code I write at those points will be in the C language. If you've taken (or are taking) CSCI 112, you'll be able to follow along without much trouble. Furthermore, there will be coding exercises. You're free to solve them in any language.

If you don't know any coding, check this link for a quick intro to C.

1.2 Basics

It takes a bit of time to explain what a *regular* language is, but the definition of a *language* is fairly simple.

Definition 1.1 (Alphabets).

An alphabet Σ is any nonempty finite set. The members of the alphabet are said to be the *symbols* of the alphabet.

Definition 1.2 (Strings).

A string w over Σ is a finite sequence of symbols from Σ.

$|w|$ denotes the *length* of w, which is the number of symbols it contains. w^R is the *reverse* of w. ϵ is the empty string, containing zero symbols, and is a string over all alphabets.

For two strings w and v, wv is the *concatenation* of the strings, and is the string obtained by "appending" v to the end of w.

w^n for some natural number n denotes the concatenation of n copies of w. Further, $w^0 = \epsilon$.

Definition 1.3 (Languages).

A language over Σ is *a* set of strings over Σ. The empty language, containing no strings, is denoted $\emptyset$. The *Kleene closure* of Σ, denoted Σ^*, is the set of *all* strings over Σ.

Example (Languages).

Suppose $\Sigma = \{a, b\}$. Examples of strings over this are ab, $baab$, $ababa$. The lengths of these strings are respectively 2, 4, and 5; and their reverses are respectively ba, $baab$, and $ababa$. A language is any set of strings composed of symbols from Σ:

1. The empty set $\{\}$ is a language over Σ. (And, in fact, over all alphabets.).
2. $\{aab, b\}$ is a language over Σ.
3. So is $\{a, aa, aaa, \ldots\}$.

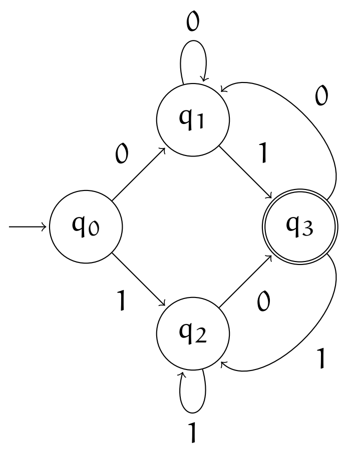

Figure 1: A rudimentary DFA.

2 Finite Automata

Moving forward, we'll discuss basic models of computation which *recognize* languages. Then, we'll discuss a few applications.

2.1 Deterministic Finite Automata

Take a look at Figure 1. A DFA is composed of several states, here q_0 through q_3. Start at q_0; this is the state you are currently on. (You can place your finger on it to signify this.) You will have some input string, say 1110. Now notice that each state has two arrows from it: one labeled 0, and one labeled 1. For each character in the input string, move your finger to the state indicated by the arrow corresponding to that character. For 1110, you would first move to q_2 through the first 1. Then, you would remain in q_2 for the next two 1 inputs, because the arrow corresponding to input 1 is a loop back onto q_2. Finally, the 0 input will take you to q_3.

Notice that q_3 is circled twice—this indicates that it is a final state. If the input ends while your finger is here, then the string is said to be *accepted*. Otherwise, it is rejected. The set of all strings accepted by an automaton is said to be the language recognized by that automaton.

2.1.1 Formal Description

The formal definition will be necessary to speak precisely about automata.

Definition 2.1 (Deterministic Finite Automaton).
A deterministic finite automaton is a 5-tuple $(Q, q_0, F, \Sigma, \delta)$, where:

1. Q is a *set of states*.
2. q_0 is an element of Q known as the *starting state* or the *initial state*.
3. F is a subset of Q and is the *set of final states*.
4. Σ is the input alphabet.
5. δ is the *transition* or the *next-state* function, mapping $Q \times \Sigma$ to Q.[1]

[1]This maps a state-and-symbol pair to a state. It tells you the next state based on the current state and the input character.

Example.

For the DFA in Figure 1:

1. The set of states is $Q = \{q_0, q_1, q_2, q_3\}$
2. The initial state is q_0.
3. The set of final states is $\{q_3\}$.
4. The input alphabet Σ is $\{0, 1\}$.
5. δ can be represented by this table:

δ	0	1
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_3	q_2
q_3	q_1	q_2

DFAs are usually the easiest to implement. Here's a C program for the DFA of Figure 1:

```
#include<stdio.h>

int main() {
    int delta[4][2] = {
        { 1, 2 },
        { 1, 3 },
        { 3, 2 },
        { 1, 2 }};
    int currentState = 0;
    char input[512];

    printf("Input: ");
    scanf("%511[^\n]",input); // Read a full line, spaces included

    int index = 0;

    while (input[index] != '\0') { // while not end of string
        printf("Reading %c\n",input[index]);
        // set next state according to table
        currentState = delta[currentState][input[index] - '0'];
        index += 1;
    }

    if (currentState == 3) { // 3 is final state
        printf("String %s accepted\n",input);
    else {
        printf("String %s rejected\n",input);

    return 0;
}
```

This design can be generalized to any DFA; mainly, the changes would be to the δ table and to the method of detecting whether the state is a final state. For the latter, a more general approach would be to use an array for F, and to check whether the current state is in that array.

Exercise 1.

What language is recognized by the DFA in Figure 1? Remember that this is the set of all strings that are accepted by the automaton.

Exercise 2.

This one's a little tough, but not impossible. Construct a DFA which recognizes the language

$$L = \{\epsilon, 0^2, 0^3, 0^4, 0^6, 0^8, \ldots\},$$

which contains all strings of n zeroes, where n is divisible by 2 or by 3 (or both).

Give the 5-tuple $(Q, q_0, F, \Sigma, \delta)$ for this automaton, and implement it.

2.2 Nondeterministic Finite Automata

Now for the NFAs. There are two types: NFA, and NFA$_\epsilon$. For a standard NFA, its δ function is from $Q \times \Sigma$ to $\mathcal{P}(Q)$, which is the power set of Q.[2] This means that the same state-symbol combination can take you to multiple states, or even to no states at all.

When tracing the behavior of an NFA with your hand, you won't be restricted to one finger like with DFAs. Whenever your finger is on a state q and you read an input character σ, you'll need to place a finger on each of the states in the set $\delta(q, \sigma)$; that is, every state that follows from q on input σ. When there are multiple states in $\delta(q, \sigma)$, this results in a form of "branching". When there is no transition from q on input σ—meaning $\delta(q, \sigma) = \emptyset$—you'll simply remove the finger from q, since there is nowhere to move it to.

An NFA recognizes a string if at least one branch is on a final state when the input ends.

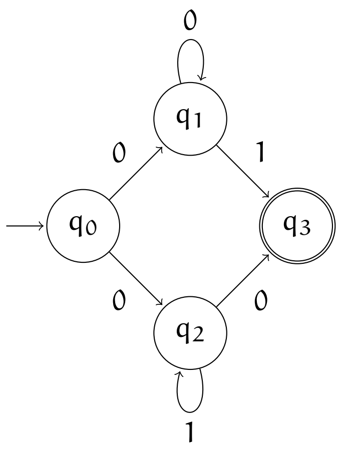

Figure 2: A rudimentary NFA.

[2]The power set of a set A is the set of all possible subsets of A. For example, $\mathcal{P}(\{a, b, c\})$ is

$$\{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Example.

Figure 2 shows an example of an NFA. Notice that there are two 0-transitions from q_0, and no 1-transitions. This means $(q_0, 0)$ maps to the set $\{q_1, q_2\}$, and $(q_0, 1)$ maps to the empty set.

We can make similar observations about the other states, and construct a formal representation of this NFA.

1. The set of states is $Q = \{q_0, q_1, q_2, q_3\}$
2. The initial state is q_0.
3. The set of final states is $\{q_3\}$.
4. The input alphabet Σ is $\{0, 1\}$.
5. δ can be represented by this table:

δ	0	1
q_0	$\{q_1, q_2\}$	$\emptyset$
q_1	$\{q_1\}$	$\{q_3\}$
q_2	$\{q_3\}$	$\{q_2\}$
q_3	$\emptyset$	$\emptyset$

For the NFA in figure 2, given an input of 0001, you would start at q_0, then:

1. Read the first 0, and place fingers on both q_1 and q_2, since q_0 has 0-transitions to both of those states.
2. Read the second 0. The finger on q_1 stays in place, while the finger on q_2 goes to q_3.
3. Read the third 0. The finger on q_1 stays in place. As for the finger on q_3, since q_3 has no 0-transitions, you will remove this finger.
4. Read the 1. The finger on q_1 moves to q_3.
5. The input has been exhausted, and there is a branch on the final state q_3, so the string 0001 is accepted.

Exercise 3.

Implement the NFA of Figure 2.

Now, let's cover the NFA_ϵ. These are NFAs with ϵ-transitions. Knowing that ϵ is the empty string, these transitions are those which do not consume an input. Meaning, if you are on a state q and it has ϵ-transitions to several other states, you need to have a finger on all of those other states, as well as q itself.

Importantly, the NFA_ϵ is equivalent in "strength" to the NFA, and to the DFA. No one of them can recognize a language that another can't. This also means that given an automaton of one type that recognizes some language L, you can construct an equivalent automaton of a different type (of those three) that recognizes the same language.

2.3 Applications to String-Searching

Automata have several applications. One of them is string-searching, which is the problem of finding occurrences of some pattern w inside of a string s.

There are several famous algorithms, most notably the Knuth-Morris-Pratt (KMP) algorithm and the Boyer-Moore algorithm. We'll only go over the former here, since it's simpler to describe, but the latter is typically faster in practice. (Both have $\mathcal{O}(|s|)$ time complexity.)

2.3.1 The Naïve Algorithm

Before that, however, let's try a few simpler approaches to string searching, to get a feel for the problem.

Example.

Suppose you want to search for all occurrences of the pattern "ana" in the string "hananoana". The most obvious approach is to "slide" w across s in the below way. You compare w with every length-$|w|$ substring of s, and whenever a match is found, indicate its position. In this example, the underlined instances of w match the corresponding position in s.

```
h   a   n   a   n   o   a   n   a
a   n   a
    a   n   a
  -------------
        a   n   a
            a   n   a
                a   n   a
                    a   n   a
                        a   n   a
                      -------------
```

The code for this is below. We'll represent w and s as strings `w` and `s`. For each position `i` in s, we take the string of length $|w|$ starting at `i` and compare it with w itself, character by character. If any mismatch is found, we skip to the next `i` without wasting time.

```c
int ls = strlen(s), lw = strlen(w);
for (int i = 0; i < ls - lw; i++) {
    if (!strncmp(s+i, w, lw))
        printf("Match found at %d", i);
}
```

`strncmp` is a string-comparison function which returns 0 when the two strings are equal; if this occurs, the `!` operator turns it into a 1 so that the `if` condition is true. As for `strlen`, it just returns the length of the string.

This approach is the most simple. To give a hand-wavy worst case "analysis" of the algorithm, the outer loop runs $|s| - |w|$ times, and each string comparison takes $\mathcal{O}(|w|)$ time, so we have $\mathcal{O}\big((|s| - |w|)|w|\big)$ operations, which simplifies to $\mathcal{O}(|s||w|)$ time, since $|s| \geqslant |w|$.

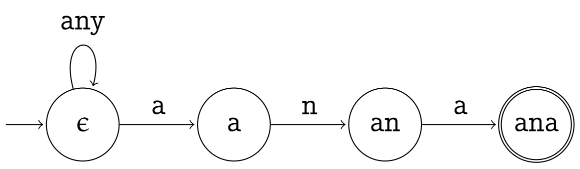

Figure 3: An NFA that recognizes strings containing “ana”.[3]

2.3.2 The NFA Algorithm

Let’s start incorporating automata into the mix. Figure 3 shows an NFA that recognizes all strings containing “ana” as a substring. Note that each state is “labeled” with the prefix of “ana” that has been recognized up to that point. Also note that the any-transitions occur on any character, including ‘a’ and ‘n’.

To implement this, we can take a standard NFA implementation as in Exercise 3, but move the “final state detection” part of the code to the inside of the loop that parses the input. Whenever one of the branches of the NFA reaches the final state, the position of the substring corresponding to that branch will be printed.

Exercise 4.

Write a program that follows the above description, implementing the NFA of Figure 3 in such a way that the location of the substring “ana” is printed whenever it is recognized.

Exercise 5.

Try determining the big-$\mathcal{O}$ time complexity of the NFA implementation in Exercise 4.

2.3.3 The Shift-And Algorithm

This algorithm is a more efficient implementation of the NFA method in the previous section. It represents the set of current states of the NFA as a bit vector. The main observation is that since transitions follow a linear pattern (q_0 to q_1; q_1 to q_2; and so on), they can be encoded by a simple bit-shift within that vector; we shift by one bit every time we read a character. This discussion is based on [4].

Shifting by 1 is simple enough to do in C: we have the >> operator to do this. However, we of course need to also consider whether the character that we just read actually *corresponds* to the symbols associated with the transitions. For this, we make use of something called a *mask*. Each unique symbol σ in our pattern w has its own mask, which itself is a bit vector that represents the states that are destinations of a σ-transition. Whenever we perform a shift of the bit vector of current states, we perform a bitwise AND operation (& in C) between that and the mask of the character that was read.

Example. We’ll continue the “ana” example to show these masks in action.

Since the NFA has four states, the current set of states can be represented by a bit vector of length 4. The any-transition means that the first bit of this vector is always 1. A bit vector of

[3]This NFA recognizes only strings that *end* in “ana”, rather than strings that simply contain it. However, the implementation in Exercise 4 is smoother if you follow this automaton.

1000 means we are only on the first state (corresponding to ϵ), and 1010 means we are on both the first state and the third state (which corresponds to “an”).

The symbol ‘a’ has mask 1101 because the first state, second state, and fourth state are all destinations of an a-transition. (In other words, their last character is ‘a’.) The symbol ‘n’ has mask 1010 for similar reasons. All other symbols in the alphabet have mask 1000.

Now let’s try reading the string “hananoana”. We start at 1000. We read ‘h’, and simulate a transition by shifting right by 1 then setting the leftmost bit to 1. (We set the leftmost bit to 1 because of the any-transition.) So now we have 1100. We perform a bitwise AND between this and the mask for ‘h’, which is 1000. The result is 1100 & 1000 = 1000.

We repeat this with the next character, ‘a’. Shift right and set the leftmost bit to get 1100, then perform the bitwise-AND: 1100 & 1101 = 1100. With ‘n’, we have 1110 & 1010 = 1010. The next ‘a’ gives us 1101 & 1101 = 1101. Since the rightmost bit is 1, we can print the current position and move on.

The rest of the string “hananoana” proceeds similarly. Try working it out yourself.

The advantage of the shift-and algorithm is that as long as $|w|$ doesn’t exceed your machine’s word size in bits (most likely 64 bits), it can be very efficient, running in $\mathcal{O}(|s|)$ time. However, if $|w|$ does exceed the word size, the performance drops to $\mathcal{O}(|s||w|/r)$ (where r is the machine’s word size in bits), which simplifies to $\mathcal{O}(|s||w|)$ again since r is unlikely to vary. This does have a smaller constant factor than the naïve algorithm we went over, which is neat.

Exercise 6.

Implement the shift-and algorithm described above. **Hint:** To set the leftmost bit after shifting, you may use a bitwise OR (which is the | operator in C).

2.3.4 The DFA Algorithm

The next algorithm we’ll take a look at uses a DFA to perform the search instead. The description is based on [2]. It might seem odd that we went from an automaton to a bit vector to increase efficiency, but are now going back to automata.

The advantage that DFAs have over NFAs is that they only need to track a single current state, rather than a set of states. This means we can represent the current state by a single integer as in Section 2.1 rather than a bit vector, such that we are not heavily limited by the machine’s word size (practically speaking).

Like before, we have $|w| + 1$ states, but since DFAs need a transition on every symbol from every state, the transition table is quite a bit more complicated. For the “ana” example, the DFA looks like what’s in Figure 4. Again, states are labeled with their corresponding prefix of “ana”.

We’ll need a bit of extra notation to continue the discussion. w may be represented in terms of its constituent symbols $\sigma_1\sigma_2\cdots\sigma_n$, where each σ_i is the i’th symbol of w. For each state q_i, its label can then be expressed as $\sigma_1\sigma_2\cdots\sigma_i$. ($q_0$’s label is the empty string ϵ, as we saw.) This label will be denoted by $l(q_i)$.

Suppose we have matched the first i symbols of w, giving us a prefix $\sigma_1\sigma_2\cdots\sigma_i$ that has been recognized so far. This puts us on state q_i. Now, we need to figure out where a transition from this state would take us if we read a character σ. That is, we need to find $\delta(q_i, \sigma)$.

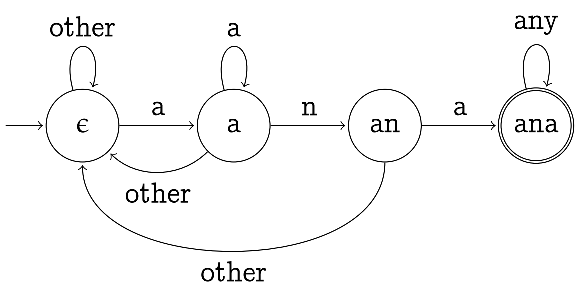

Figure 4: A DFA that recognizes strings containing "ana".

The computation of $\delta(q_i, \sigma)$ is fairly simple if σ is just σ_{i+1}, the next symbol of w. This would put us on q_{i+1}, of course. But in other cases, we might go back to q_0, or stay on q_i, or end up anywhere in between, depending on what σ is. It turns out that the state $\delta(q_i, \sigma)$ is that whose label is the longest string x that is both a prefix of w, and a suffix of $l(q_i)\sigma$ (that is, the concatentation of $l(q_i)$ with σ). Take a moment to think about why.

This property is something we can translate into code. The goal is to find the longest possible x. In the below code, we'll represent w and $l(q_i)\sigma$ as strings `w` and `v` respectively. As for x, its length $|x|$ will be given by `lx`. Of course, given $|x|$ and either w or $l(q_i)\sigma$, we can determine what x should be.

```
int lw = strlen(w), lv = strlen(v), lx = 0;

for (int i = 1; i <= lv; i++)
    if (!strncmp(w, v + (lv - i), i))
        lx = i;

return lx;
```

This is a fairly straightforward algorithm. It iterates over every possible length of x, counting from 1 up to $|l(q_i)\sigma|$. (Clearly, you can't have a substring bigger than the original string.) For each length—represented by `i`—it compares the prefix of w and suffix of $l(q_i)\sigma$ that are of that length. If they're equal (indicated by `strncmp` returning 0), then that's a candidate for x, and its length its recorded in `lx`. This can occur multiple times. By the end of the loop, the maximum such length will be in `lx`, and that's our return value. `lx` can then be used to identify the state $\delta(q_i, \sigma)$.

Exercise 7.

Based on the above discussion, write a function that constructs the full transition table of an automaton, given some input pattern and alphabet.

To analyze the above approach informally: a single string comparison takes linear time in the worst case, and $|w|$ string comparisons are done, giving $\mathcal{O}(|w|^2)$ time for a single entry of the δ table. The δ table has $|w + 1||\Sigma|$ entries ($|\Sigma|$ is the size of the alphabet), but the $|\Sigma|$ factor is a constant that can be ignored. So all in all, this gives $\mathcal{O}(|w|^3)$ time complexity to calculate the transition table of the DFA.

The above was just to build the DFA from scratch. It's also possible to construct the NFA—which takes $\mathcal{O}(|w|)$ time—then convert that into a DFA using an $\mathcal{O}(|w|^2)$ algorithm.[4]

As for running the DFA, as should be clear from the implementation in Section 2.1, it only takes $\mathcal{O}(|s|)$ time. This is great, but the time to construct the DFA is too long. The algorithm in the next section will rectify that, moving us from $\mathcal{O}\left(|w|^3 + |s|\right)$ to $\mathcal{O}\left(|s|\right)$ time.

2.3.5 The Knuth-Morris-Pratt Algorithm

So finally, we've built up to the KMP algorithm. It's based on the same observations made in the previous section about the longest prefix-suffix, but streamlines the process by a great deal. The main difference revolves around replacing δ with a more compact table, called the "failure function". Its computation remarkably only takes $\mathcal{O}(|w|)$ time. Then, running the algorithm takes $\mathcal{O}(|s|)$ time. Since we assume that $|s| \geqslant |w|$, the overall running time is $\mathcal{O}(|s|)$. The description and code in this section are adapted from [1].

In the previous section, we wanted to find the longest string x that is both a prefix of w and a suffix of $l(q_i)\sigma$. To do this, we used a brute-force method that—given these two strings—found the length of x, which could then be used to obtain x itself. This string x is the label of the state $\delta(q_i, \sigma)$.

The computation of the failure function is based on the same prefix-suffix observation as the brute-force approach, but skips having to compute the entire transition table. Instead, you get a table that specifies which state to move to when there is a mismatch. That is, if you're on q_i and read σ such that $\sigma \neq \sigma_{i+1}$, you move to the state given by the failure function, which is $q_{f(i)}$. Then, you try σ again from $q_{f(i)}$, comparing it with $\sigma_{f(i)+1}$. If there's *another* mismatch, you move to $q_{f(f(i))}$ and try again (and so on and so on). As much of this as possible is compressed into the algorithm's pre-processing step.

The above description is in terms of automata. If you remember the picture from Section 2.3.1, the naïve algorithm would "slide" w across s and try to find matches. The KMP algorithm still does this, more or less, but skips steps whenever possible, by sliding $i - f(i)$ units upon encountering a mismatch, to the earliest next possible location of w in s. Below is a visual representation of that, following the same format. Note that here, $f(1) = 0$, $f(2) = 0$, and $f(3) = 1$—although, the $f(2)$ case isn't relevant for the choice of $s = $ hananoana, since every single 'a' is followed by an 'n', so we never fail when trying to match an 'n'. Notice for instance that whenever we check the last symbol of ana ($\sigma_3 = $ a), we slide right by $3 - f(3) = 2$ units.

h	a	n	a	n	o	a	n	a
a	n	a						
	a	n	a					
			a	n	a			
					a	n	a	
						a	n	a

Because we just want to know what state to move to if we fail, it doesn't matter what character σ we read that caused the mismatch, so the prefix-suffix we'll be computing is instead

[4]Given an NFA with n states, it can be converted to a DFA that has up to 2^n states, meaning the conversion takes up to $\mathcal{O}\left(2^n\right)$ time. For string-search, this bound is $\mathcal{O}\left(n^2\right)$. Conversions are out of the scope of this document.

that of $l(q)$ rather than w and $l(q)\sigma$. Below is the code that does that in $\mathcal{O}(|w|)$ time. We'll represent the failure function as an array indexed by i, meaning `f[i]` is just $f(i)$. The pattern `w`, and the string `s`, will also be indexed by `i`, meaning `w[1]` is σ_1, `w[2]` is σ_2, and so on, with `w[0]` left undefined.[5]

```
int t = 0;
int lw = strlen(w+1); // +1 because one-indexing
f[1] = 0;
for (int i = 1; i < lw; i++) {
    while (t > 0 && w[i+1] != w[t+1])
        t = f[t];
    if (w[i+1] == w[t+1])
        f[i+1] = ++t;
    else
        f[i+1] = 0;
}
```

Let's go through this step by step. `t`, much like `i`, represents a state. The outer loop, at the beginning of each iteration, can be said to state "suppose we are on state q_i." The variable `t` will, by the end of the iteration, contain the value of $f(i+1)$, which is the state we would move back to if we fail to match σ_{i+1}.

In the inner loop's condition, `w[i+1]` and `w[t+1]` can respectively be said to represent the suffix and the prefix of $l(q_{i+1})$. While this is indeed a comparison of only one character for a prefix-suffix of greater length, it's enough because the value of `t` encodes enough information that this comparison is enough to determine whether there is a mismatch between the prefix and suffix. If there is a mismatch, the value of `t` is set to `f[t]` to signify that, and the prefix-suffix is tested again.

Eventually, the inner loop will break, on the condition that `t == 0` or `w[i+1] == w[t+1]`. If it is the latter case, there is one (more) character present in the prefix-suffix, and so `t` is incremented, and `f[i+1]` is set to that new value of `t` to signify the next-state upon failing to match σ_{i+1}. The `else` statement is reached only in the case that `t == 0` and `w[i+1] != w[t+1]`, and signifies that there is no prefix-suffix whatsoever—concretely, this means we mismatched enough times that we reached q_0, and still mismatched again after that, so we stay in place, having recognized ϵ.

Although we do have nested loops in the above code, rest assured that it really does run in $\mathcal{O}(|w|)$ time. We will be stating this without proof here, but feel free to look it up yourself if you're interested.

Now that we've completed the pre-processing of w by computing the failure function, we can actually start searching s.

[5]This is in contrast to the code in the previous sections, where `w` and `s` were zero-indexed, with the i'th symbol being at index $i-1$ of the array. The reason I deviate from that here is to preserve the relation between the automaton's states and the symbols of w; it's neater when you have that q_i is the state reached after reading σ_i. In practice, you're free to zero-index `w` and `s`, and change some of the numbers and indices to reflect that. One such change is `int t = -1` on the first line.

Below is the code; the actual KMP algorithm. Remember that `i` is the current state in the automaton (although we haven't *explicitly* constructed one). `p` will be the current position within s.

```
int i = 0; // current state
int ls = strlen(s+1); // again, one-indexing
int lw = strlen(w+1);
for (int p = 1; p <= ls; i++) {
    while (i > 0 && s[p] != w[i+1])
        i = f[i]; // slide to the next possible location of w
    if (s[p] == w[i+1])
        i++;
    if (i == lw) // occurrence found?
        // position of w is at (p - i + 1). output that however you want
        i = f[i]; // slide to the next possible location of w
}
```

Like we mentioned previously, all this does is proceed through the states of the automaton, performing `i = f[i]` to slide forward by $i - f(i)$ positions whenever there is a mismatched character. One remark is in the case that an occurrence is found: the location of `w[1]` will be at index `p - i + 1` of `s`. Once this is output (printed, put into a list, etc.), we slide forward to the earliest next possible location of the pattern.

For example, if searching for "ana" in the string "anana", the second 'a' in "anana" appears in two different occurrences of the pattern "ana". One starts at index 1, and the other at index 3. The failure function, by its nature, accounts for overlapping cases such as these, and allows the algorithm to slide the pattern to the appropriate position after matching.

3 Regular Languages and Regular Expressions

The finite automata we described in Section 2 can recognize many different languages, but not all of them. The class of all languages that can be recognized by these automata is known as the class of *regular languages*.

3.1 Regular Languages

Let's first define some operations on languages in general. We'll use these to define regular languages. Recall first that a language is just a set of strings.

Definition 3.1 (Operations on Languages).

- Given two languages A and B, the language which contains all strings in A and all strings B and no other strings is called the *union* of A and B, and is denoted $A \cup B$.
- Given two languages A and B, the *concatenation* of the two languages is the set of all strings of the form ab, where a is in A and b is in B. This language is denoted AB. Further, A^n is the concatenation of A with itself n times.
- Given a language A, the *Kleene closure* of A is denoted A^*. It is the union of all languages $A^0, A^1, A^2, \ldots$

It would be good to present one example, at least.

Example.

Suppose $\Sigma = \{a, b\}$, A is the language $\{\epsilon, aa, aaaa, aaaaaa\}$, and B is the language $\{\epsilon, bbb, bbbbb\}$. Then,

- AB is the language

$$\{\epsilon, bbb, bbbbb, aa, aabbb, aabbbbb, aaaa, aaaabbb,$$
$$aaaabbbbb, aaaaaa, aaaaaabbb, aaaaaabbbbb\}.$$

- $A \cup B$ is the language containing ϵ, aa, $aaaa$, $aaaaaa$, bbb, and $bbbbb$.
- A^* contains all strings with an even number of a's. As for B^*, it contains all strings with 0, 3, 5, 6, or 8 or more b's.[6]

Now to define regular languages according to these operations. The class of regular languages over some alphabet Σ uses a recursive definition; you start from a few base cases, and build up the rest by repeatedly applying different operations.

Definition 3.2 (Regular Language)**.** The class of regular languages over an alphabet Σ is defined as follows:

- The empty language $\emptyset$ is a regular language.
- For each symbol a in Σ, the language $\{a\}$ is a regular language.
- If A is a regular language, then A^* is also a regular language.
- If A and B are regular languages, then $A \cup B$ and AB are regular languages.
- No other languages over Σ are regular.

Now, regular expressions are simply a notation to express a regular language. Union, concatenation, and closure are all that is needed (along with parentheses for grouping). Here's some examples of regular expressions over $\{a, b\}$ and what they mean.

Example (Regular Expressions).

- $\mathbf{ab} + \mathbf{a}^*$ is a regular expression for the language $\{ab, \epsilon, a, aa, aaa, \ldots\}$.
- $(\mathbf{a} + \mathbf{b})^*$ is a regular expression for the language Σ^* of all strings composed of a's and b's.
- $\mathbf{b}^*\mathbf{ab}^*$ is a regular expression for the language of all strings containing exactly one a.

Regular expressions are extremely powerful and see heavy use in string-processing methods more general than the string-search problem in Section 2.3. For example, your text editor or IDE uses them for syntax highlighting.[7]

[6] You can take this for granted, or read more here.

[7] Usually regex does the job, but sometimes something more sophisticated is needed. For example, Visual Studio colors local variables and global variables differently, but regex by itself can't differentiate between the two. My hope is to cover some of the relevant techniques in the future.

3.2 Applications to Compilers

The regular expressions we saw above are a mathematical notation to specify a regular language. In practice, a different syntax is available to us which extends the three operations of concatenation, union, and closure with "syntactic sugar", allowing us to specify languages with more ease and conciseness. Some extensions even take us out of the limited scope of regular languages and into a larger class of languages.

One area of string-processing is that of compiler design and implementation. Source code is little more than a long string, with some kind of structure according to the programming language that it was written in. Compilers process this string across several phases, each time making a certain transformation to the program to poke at and understand its structure and meaning, then generating code according to that.

3.2.1 Lexing

If we were to split the compilation process into phases, we have the following:

1. **Lexing** or **tokenizing** the input, by grouping individual characters into meaningful units.
2. **Parsing** the stream of tokens to produce some intermediate representation. Simultaneously, performing **semantic analysis** to verify that the input "makes sense".
3. **Code Generation**, which produces the equivalent target code to the original program, based on the intermediate representation.

Our topic in this section will be on the first step, as it uses regular expressions to tokenize the input. We want to scan the program and logically transform it from a stream of characters—each of which individually have no meaning—into a stream of tokens, which are the smallest meaningful "units" in a programming languages. Here are some examples:

- Identifiers
- Integers
- Floating-point numbers
- Keywords
- Left/right parentheses
- Arithmetic operators
- Logical operators
- Left and right braces
- Semicolons
- Commas
- Strings
- What else?

Each of these will have an associated regular expression. For a simple example, in the C language, the keyword `void` is the concatenation of the four symbols v, o, i, and d. Identifiers (names of variables and functions) and numbers are more complicated, and use different combinations of concatenations, unions, and closures.

For each regular expression we have, we can convert it into an NFA or DFA that recognizes the associated language. That way, we can run several automata in parallel and return a value whenever a token[8] is recognized.

[8]What we are actually recognizing are *lexemes*, rather than *tokens*. Lexemes are merely instances of tokens. For instance, "`foo`" and "`bar`" are both lexemes of the "identifier" token.

3.2.2 `Flex`

`Flex` is a program that generates lexers. It takes in a specification as input. This is a `.l` file that contains the regular expressions to be recognized, as well as C code for the actions to be taken upon recognizing them. The output is a lexer, which is a program that takes in some input file (typically a program), and outputs the consistuent lexemes in some format. The specific format of that output depends on the actions defined in the C code.

The `.l` input file has three sections. These sections are separated by two percent signs %%.

1. The first is the **Definitions** section, in which you define things that will be used in the later sections. This includes shorthands for regular expressions, and C code containing things such as `#include` directives.

2. The second is the **Rules** section, which contains a list of regular expressions. Each one is followed by a single C statement (or block of statements, enclosed in braces) which determines what is to be done whenever a lexeme is recognized. The format looks like this:

   ```
   your_regex_here { statement; statement; }
   ```

   Within the C code, you may use `yytext` to refer to the lexeme that was just recognized. For example,

   ```
   another_regex { printf("Lexeme recognized: %s\n",yytext); }
   ```

   will print "`Lexeme recognized: some_lexeme`", whatever the lexeme is.

   When it comes to conflicts between rules (due to their regular expressions both matching the same string), two disambiguating principles are followed. The first is that the lexer always recognizes the longest possible lexeme. The second is that when two or more rules recognize the same such lexeme, the rule that is higher in the list is prioritized.

3. The third section is the **User Code** section. This is just additional C code you may want to define, possibly including a `main` function.

3.2.3 `Flex` Regular Expressions

Now let's go over the regular expression syntax used in `Flex`. We won't cover everything here; you can check [3] for that.

- Concatenation has no specific notation; just put things one after the other. The regular expression `abc` means "`a` followed by `b` followed by `c`".

- There is the `|` operator for union; for instance, the regular expression `a|b|c` means "`a` or `b` or `c`".

- Kleene closure uses the `*` operator. Note that this has higher precedence than union and concatenation, so that a regular expression such as `ab*` means "`a` followed by zero or more instances of `b`".

- The *positive closure* is also possible, with the `+` operator, and means "one or more" rather than "zero or more". The regular expression `ab+` means the same thing as `abb*`.
- The "character class" notation which abbreviates unions of large sets of characters, and allows you to specify ranges of characters. The regular expression `[abc]` is equivalent to `a|b|c`, and so is `[a-c]`. You may also specify multiple ranges, such as in the regular expression `[a-zA-Z0-9]`, which means "anything in the range `a-z`, or anything in the range `A-Z`, or anything in the range `0-9`". Note that the character class needs some special considerations with regards to escaping with `\`.
- The "negated character class" is the same as the character class, except it matches all characters that are not specified. This is done by putting a caret `^` after the opening square bracket. For instance, `[^0-9]` matches all characters that are not digits.
- Parentheses may be used to group things together. For example, the regular expression `(ab)+` means "one or more instances of `ab`".
- A '?' operator exists for "optional" parts of the regular expression. The regular expression `ab?` means "`a`, possibly followed by a `b`", and is equivalent to `a|ab`. Another example is `a(bc)?`, equivalent to `a|abc`.
- The `.` matches any character apart from a newline. The regular expression `a.c` means "`a`, followed by any one character, followed by `c`".

3.2.4 Example: Calculator Lexer

Now that we have the basic syntax down, let's do an example for a simple calculator language. The vision for this language is simple: users may declare a variable, or do an assignment to change the value of a variable. They may also write an expression consisting of numbers and/or variables, in which case the expression is evaluated and its value printed.

The means to track declared variables and their values, as well as to interpret and evaluate expressions, are beyond the scope of this document.[9] We want the following tokens:

- A `let` keyword to declare variables.
- Identifiers for variables. These will contain letters only.
- Numbers, which we'll restrict to be decimals only.
- Four arithmetic operators: addition, subtraction, multiplication, and division.
- Parentheses for grouping parts of the input expression together.
- An assignment operator '='.

[9]As mentioned on page 13, we may cover the relevant material in the future. It corresponds to step 2 of the compilation process on page 14, and involves *context-free languages* (CFLs), a larger class of languages than the regular languages. Feel free to look into those, the corresponding *context-free grammars* which describe them, and the *pushdown automata* that can recognize them. Also check out `Bison`, a parser generator used alongside `Flex` in generating compilers.

The corresponding Flex program is shown below. Some things to keep in mind as you read:

- A line_number variable is used to output the line on which an invalid character was encountered, by the very last rule. This is incremented when we see a newline character (line 16).
- yytext (described earlier) is used to print the lexeme corresponding to the regular expression that was recognized.
- The wildcard . operator is escaped with a backslash to recognize the actual character, in line 11. Similarly, the - character is escaped in the following rule to avoid specifying a range of characters.
- You can ignore the yywrap() function. It needs to be there for *reasons*, but isn't something worth explaining.

```
1   /* DEFINITIONS SECTION */
2  %{
3      #include<stdio.h>
4      int line_number = 1;
5  %}
6
7  %%
8   /* RULES SECTION */
9  let            { printf("Keyword: let\n"); }
10 [a-zA-Z]+      { printf("Identifier: %s\n",yytext); }
11 [0-9]+\.[0-9]+ { printf("Number: %s\n",yytext); }
12 [+\-*/]        { printf("Arithmetic operator: %s\n",yytext); }
13 [()]           { printf("Parenthesis: %s\n",yytext); }
14 [=]            { printf("Assignment: =\n"); }
15 [ \t]          ; /* empty statement. whitespace is ignored */
16 [\n]           { line_number++; }
17 .              { printf("Invalid character %s on line %d\n",yytext,line_number); }
18
19 %%
20  /* USER CODE SECTION */
21 int yywrap() { return 1; }
22
23 int main() {
24     yylex();
25 }
```

After running Flex on this and compiling the output, we can feed this any input we like, and it'll scan through it, giving outputs in the process. For instance, suppose we input this:

```
let letlet x abcdef abcd1 abcd1.1
hello = 1 + world * 1.1
```

We'll get the following output.

```
Keyword: let
Identifier: letlet
Identifier: abcdef
Identifier: abcd
Invalid character 1 on line 1
Identifier: abcd
Number: 1.1
Identifier: hello
Assignment: =
Invalid character 1 on line 2
Arithmetic operator: +
Identifier: world
Arithmetic operator: *
Number: 1.1
```

This concludes the section on regular expressions and `Flex`, and in fact the whole document. Again, keep in mind that this was only the first step of the compilation process—we can recognize the tokens, but they don't really mean anything just yet.

References

[1] Alfred Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[2] David Eppstein. *Automata and String Matching*. URL: https://ics.uci.edu/~eppstein/161/960222.html.

[3] Vern Paxson, Will Estes, and John Millaway. *Lexical Analysis with Flex*. 2023. URL: https://www.cse.iitk.ac.in/users/swarnendu/courses/spring2024-cs335/flex.pdf.

[4] Sven Rahmann. *Exact Pattern Matching with Automata*. 2021. URL: https://www.rahmannlab.de/lehre/alsa21/01-2-automata.pdf.