Chapter 1

Finite State Machines

1.1 Introduction

So far we have talked about the language features that languages may have. However, now we want to start talking about how we can take features from one language, and implement them in another. While a naive approach may be something like making a library or some wrapper functions. For a simple example, maybe I wanted to add booleans to C. I can just write a define macro for 1 and 0 which we name as true and false. For a more complicated example if I wanted to add pattern matching in C, then maybe I create a struct called data which can hold any value which can be pattern matched and a function: void* match(data* value, int (**patterns)( data* ), void* (**exprs)(data*)) which takes in a piece of data to match, a series of functions that return true if the value matches it, and a series of functions that return some value\(^1\). This way is terrible and so the typical way to add something is by changing the compiler (Or if we want to go one step further, let’s design our own language, which means we need to make a new compiler-HAH!).

1.1.1 Compilers

While this is not CMSC430: Compilers, we need to setup the basis of compilation. We will talk about this more in depth in a future chapter, but here’s a quick overview. A compiler is a language translator (typically some higher level programming language to assembly). To translate one language to another we need to do the following:

• break down the language to bits that hold information

\(^1\)See Appendix A for a rough implementation
CHAPTER 1.FINITE STATE MACHINES

• take those bits and figure out how to store that information in a meaningful way
• take the stored information and map it to the target language.
• generate the target language.

The best way to break down the language is to use regular expressions. However what if your language doesn’t have regular expressions? Simple: let’s implement regular expressions in a way that we don’t need to compile.

1.1.2 Background - Automata Theory

Imagine that we want to create a machine that can solve problems for us. Our machine should take in a starting value or values, a series of steps, and then give us output. Depending on when and who you took CMSC250 with, you already did this. A circuit or logic gate is the most basic form of this. If our input is values of true and false (1 and 0), let us put those inputs into a machine that ands, ors and negates to get an output value.

The issue here is we do not have any memory. We cannot refer to things we previously computed, but only refer to things we are inputting in each gate. Once we add a finite amount of memory, we can accomplish a whole lot more and we get what we call finite automata (FA). I use finite automata interchangeably with finite state machine (FSM) but typically FA is used in the context of abstract theory and FSM is used with the context of an actual machine, but they all refer to the same thing.

Once we start adding something like a stack (infinite memory), we get a new type of machine called push-down automata (PDA) where we theoretically have infinite memory, but we can only access the top of the stack. Lifting this top-only read restriction, we get what is known as a Turing machine. As formalized in the Church-Turing thesis, any solvable problem can be converted in a Turing Machine. A Turing machine that creates or simulates other Turing machines is called an Universal Turing Machine (UTM). Fun fact: Our machines we call computers are UTMs.

All of this is to say that a compiler wants to output a language that is turing complete, one which can be represented by a turing machine. Regular expressions on the other hand, describe what we call regular languages, and regular languages can be represented by finite automata. So we will start with FSMs but know that when we get to compilation, we need something more.

1.1.3 Finite State Machines

Let’s start by modeling a universe and breaking it down to a series of discrete states and actions. Let us suppose that my universe is very small. There is just me, an room and a compass. Suppose I am standing facing north in this room. Let’s call this state $N$. When facing north, I have two options: turn right $90^\circ$ and face East, or turn left $90^\circ$ and face West. Let’s give these states some names: states $E$ and $W$ respectively. From each of these new positions (facing west, or facing east), I could turn left or right again and either end up facing back north, or face South. Let’s give the state of facing south a name: $S$. If I create a

\footnote{Initially called an ‘a-machine’ or atomic machine by Alan Turing.}
1.2 REGEX

A graph that represents all possible states and actions of the universe, I could create a graph that looks like:

```
N  Left  Right
\  |     |
 \ |     |
  W  Left
```

This graph represents a finite state machine. A physical machine can be made to do these things but for the most part we will emulate this machine digitally. We typically define a FSM as a 5-tuple:

- A set of possible actions
- A set of possible states
- A starting state
- A set of accepting states
- A set of transitions

The set of transitions is the set of edges, typically defined as a 3-tuple (starting state, action, ending state). To be clear: this is a graph. A transition is an edge, and a state is a node. This is not a good example to show what a starting or accepting state is, but we will see that in the next section.

The important takeaway is Based on where I am (which state), and what action occurs (which edge I choose), I can tell you where I will end up. So given an input, and a series of instructions, I can give you an output (sound familiar?). For example, if I start at state \(N\), and my instructions are to go left, left, left, right, left, right, right, I can traverse my path \((N \rightarrow W \rightarrow S \rightarrow W \rightarrow S \rightarrow E \rightarrow S \rightarrow W)\) to know where I am and return it (My output is \(W\) here).

### 1.2 Regex

So we did this whole thing with graphs and talked about what a machine is and a single example a use case. Let’s talk about another use case: regular expressions. For regular expressions, we define a FSM as a 5-tuple very similarly as what we previously had, but instead of actions, we have a letter of the alphabet. That is we have
CHAPTER 1. FINITE STATE MACHINES

- the alphabet \( \Sigma \) which is a set of all symbols in the regex.
- a set of all possible states \( S \)
- a starting state \( s_0 \)
- a set of final (or accepting) states \( F \)
- a set of transitions \( \delta \)

To be clear on types: \( s_0 \in S \) and \( F \subseteq S \). This is because a FSM can only have 1 starting state (no more, no less), but any number of accepting states (including 0). In the previous example we had an understood starting state \( N \), but not really any accepting states. Let us see an example of a FSM for the regular expression \( / (0|1)*1/ \).

This machine represents the regular expression \( / (0|1)*1/ \). Recall that a regular expression describes a set of strings. This set of strings is called a language. Examples of strings in the language described by the regular expression \( / (0|1)*1/ \) would be "1", "10101", and "0001". When we say that a FSM accepts a string, it means that when starting at the starting state (denoted by an arrow with no origin), after traversing the graph after looking at each symbol in the string, we end up in an accepting state (states denoted by a double circle). Let's see an example.

Given the above FSM, suppose we want to check if the string "10010" is accepted by the regex. We start out in state \( S_0 \) since it has the arrow pointing to it as the starting state. We then look at the first character of the string: "1" and consume it. If we are in state \( S_0 \) and see a "1", we will move to state \( S_1 \). We then look at the next second of the string (since we consumed the first one): "0" and consume it. Since we are in state \( S_1 \), if we see a "0", then we move to state \( S_0 \). We then proceed to traverse the graph in this manner until we have consumed the entire string. The traversal should look something like

\[
S_0 \rightarrow 1 \rightarrow S_1 \rightarrow 0 \rightarrow S_0 \rightarrow 0 \rightarrow S_0 \rightarrow 1 \rightarrow S_1 \rightarrow 0 \rightarrow S_0
\]

Since we end up at state \( S_0 \), and \( S_0 \) is not an accepting state (it does not have a double circle), then we say this machine (this regular expression) does not accept the string "10010". Which is true, this regex would reject this string.

On the other hand if traversed the graph with "00101", our traversal would look like

\[
S_0 \rightarrow 0 \rightarrow S_0 \rightarrow 0 \rightarrow S_0 \rightarrow 1 \rightarrow S_1 \rightarrow 0 \rightarrow S_0 \rightarrow 1 \rightarrow S_1
\]

and we would end up in state \( S_1 \) which is an accepting state. So we could say that the machine (the regular expression) does accept the string "00101".
1.3. Deterministic Finite Automata

All FSMs can be described as either deterministic or non-deterministic. So far we have seen only deterministic finite automata (DFA). If something is deterministic (typically called a deterministic system), then that means there is no randomness or uncertainty about what is happening (the state of the system is always known). For example, given the following DFA:

Now this graph is missing a few states (one really). What happens when I am in state \( S_0 \) and I see a “b”\(^3 \)? There is an implicit state which we call a “garbage” or “trash” state. A trash state is a non-accepting state where once you enter, you do not leave. There is an implicit one if you are trying to find a transition symbol or action which does not have output here. That is, there is an edge from \( S_1 \) to the garbage state on the symbol “b”. There are also transitions to the garbage state from \( S_2 \) on “a” and \( S_3 \) on “c”. If we really wanted to draw the garbage state in, we could like so (but there really is no need to do so):

\(^3\)that is, if the string is in the language the regular expression describes. Any language a regular expression can describe is called a Regular Language

\(^4\)Determinism in philosophy is about if there is such a thing on free will and I would definitely recommend reading David Hume’s and David Lewis’s take on causality
CHAPTER 1. FINITE STATE MACHINES

Regardless if we indicate a trash state or not, no matter which state I am in, I know exactly which state I will be in at any given time.

1.4 Nondeterministic Finite Automata

The other type of FSM is a nondeterministic finite automata (NFA). Nondeterministic in math terms means that is something that is some randomness or uncertainty in the system. A NFA is still a FSM, the only difference is what are allowed as edges in the graph. There are 2 of them. Let's talk about one of them now. Consider the following FSM:

This machine represents the regular expression /(a|b)*abb/. There is still a starting state, transitions, ending states, all the things we see for a FSM. However, there is something interesting when we look at the transitions out of S0. If I am looking at the string "abb", then when I am traversing, do I go from S0 to S1 or do I loop back around and stay in S0?

In fact, there are two ways I could legally traverse this graph:

- S0 \(\xrightarrow{a} S1 \xrightarrow{b} S2 \xrightarrow{b} S3\)
- S0 \(\xrightarrow{a} S0 \xrightarrow{b} S0 \xrightarrow{b} S0\)

Since the traversal of the graph is uncertain, we call this non-deterministic. To check acceptance for an NFA, we have to try every single valid path and if at least one of them ends in an accepting state, then we accept the string. Since we have to check all possible paths, you can imagine that this is quite a costly operation on an NFA. Additionally, all NFAs have a DFA equivalent. So why use an NFA?

Let us first consider why we are using a FSM. We want to implement regex. To convert from a regular expression to a DFA can be difficult. Consider the above NFA for / (a | b) * abb / . Now consider the following DFA for the same regex:
It is much easier to go from a regular expression to an NFA than it is to go from a regular expression to a DFA. Additionally, NFAs, because they can be more condensed, are typically more spatially efficient than their DFA counterpart. However, there is of course a downside: NFA to regex is difficult, and checking acceptance is very costly. However, NFA to DFA is a one time cost and its less costly to check acceptance on a DFA. Additionally, going from a DFA to a regular expression is much easier. Now keep in mind, technically all DFAs are NFA, but not all NFAs are DFAs.

To visualize this, we typically draw the following triangle:

- Regex
- NFA
- DFA

We will talk about how to convert between all of these in a bit, but before we get too far ahead of ourselves, we need to consider the other difference an NFA has over a DFA: epsilon transitions.

An $\epsilon$-transition is a "empty" transition from one state to another. If we think of our graph as one where the edges transitions are the cost to traverse that edge, then an $\epsilon$-transition is an edge that does not cost anything to traverse (it does not consume anything). Consider the following NFA:

If I wished to check acceptance of the string "b", then my traversal may look like:

$$S0 \xrightarrow{\epsilon} S1 \xrightarrow{b} S2$$

Whereas my traversal of the string "B" may look like:

$$S0 \xrightarrow{a} S1 \xrightarrow{b} S2$$

Knowing this, you can see that this machine represents the regular expression: $/a?b/$.
1.5 Regex to NFA
1.6 NFA to DFA
1.7 DFA to Regex