CS332: M02.1 Finite State Machines (FSM)

Outcomes: By the end of these lectures you should be able to perform the following.

1. Given a grammar, justify why it is or is not regular.
2. State the definition of the Finite State Machine (FSM) and its component parts.
3. Given a regular language create the Finite State Machine for it.
4. State the acceptance criteria for an FSM.
5. Given an FSM and a string, demonstrate the state transitions for processing the string.
6. Given an FSM and a string, state whether the FSM accepts or rejects the string.
7. Provide real world examples of an FSM with justification.

References:

Notes:

1. Motivation:
   (a) From a theoretical computer science standpoint, FSMs assist is the study of what computers can compute (the core question of computer science). In order to study what computers can compute, a formal definition of computation is required. We've already seen how formal definitions are required before proofs can be performed, so computation must be well defined before any proofs can be constructed for any aspect of that computation. These mathematical definitions of computation are referred to as *models of computation*. The FSM is one of several model of computation. It is the weakest model, meaning that FSMs cannot model all of the computations that a real computer can do (as opposed to Turing Machines, which do model all computations that real world computers are capable of). So, from a theoretical computer science standpoint, FSMs are important.

   (b) From a practical standpoint, FSMs are important because real world systems have state, and changes in those systems are changes in state (we call these state transitions). These systems can be modeled as FSMs, or variations on a pure FSM. For example, a basic traffic light has a few basic states: each side can be red, green, or yellow. If we consider only one direction, the traffic light becomes a FSM that transitions between its red, green, and yellow states in turn. This basic model can be used to analyze traffic light software for correctness. For example, it is useful to prove that the software will never make all of the lights green at the same time – that's a bad state! State-based systems are everywhere in your everyday life. Your car transitions from off to on, from park to drive, and so on. Your television has state (on/off, channel, volume) and transtions to new states whenever you change the channel or turn it on and off. Modeling systems as state-based entities is ubiquitous in analysis of hardware/software systems. The FSMs studied in this course are the foundation for that.

2. Introduction:

   (a) The FSM is a simplified mathematical model of computation. It is limited because, unlike real world computers, it does not have memory. It only has states, and transitions from one state to another. We often think of computation as having an input and an output. For the FSM, the input is a finite length *sequences of symbols*, and the output is either *accept* or *reject.* The FSM's purpose then is to *recognize* certain languages, and it does so by *accepting* any and all sequences that are in the language, and *rejecting* any and all sequences not in the language. In this part of the course we will invent simple languages (sequences of symbols that follow a given pattern), and then create the FSM that can recognize the language. We will refer to the sequences of symbols as *strings* and create simple languages such as "all strings that have at least three a's," or "strings where all a's occur before any b's." Believe it or not, these simple rules form a language. All of these basic ideas, of course, deserve formal definition.

   (b) It is helpful to dispel our comon sense notions of computation and language (espeicially language!) at this point. We have all used computation to calculate some *thing*, such as calculating the area of a shape, or sizing a structural element in an aircraft. But performing a calculation for a specific purpose is not computation, it is a *use* of computation. Similarly, we have all used language to do some *thing*, such as communicating with other people, or programming a computer. But those are *uses* of language. Both computation and the concept of language exists outside of those uses, and that is what this course deals with. We will focus on the definition of computation and the definition of language separately from their usage. For our purposes, computation is simply a mindless manipulation of well-defined symbols using well-defined rules, and language is simply a set of strings. The languages (sets of strings) that we are concerned with are those that can be recognized by a FSM. Other categories of languages exist, but this course does not deal with them. This concept is foreign to most students, because you've always done computation and used language to do some thing. Once you learn to view computation and language separately from their usage, the course material becomes much easier.

3. Definitions and Notation:

   (a) Symbol: A symbol is a single, distinct, written representation. Typical symbols used in the formal language community are $a, b, 1$ and 0.

   (b) Alphabet: An alphabet, Σ, is a finite collection of symbols.

   (c) String: a string, u, is a sequence of symbols from a single alphabet.

   (d) Language: A language, L, is a set of strings.

   (e) Finite State Machine (FSM) : A FSM, M = $\{$S, Σ, q_0, F, δ $\}$, where

      i. S : A finite set of states.

      ii. Σ : The alphabet.

      iii. q_0 : The initial, or starting state.

      iv. F : A set of final, or accepting states.

      v. δ : The transition function.

   (f) Regular language: A regular language is one that can be recognized by a FSM.

(g) Regular Expression: Regular expressions are expressions that correspond to the regular languages.

4. Example: Let $\Sigma = \{\ a,\ b,\ c\}$ for all examples today.

   (a) Let L_4 = The set of strings ending in *bbb*. (Draw machine, then define S, Σ, q_0, F, and δ.)

5. Regular Languages: **Problem:** This is an awful way to specify languages. We need notation. We'll develop notation for regular languages. This notation is also called *regular expression.* A regular language is composed of strings defined by regular expressions. Regular expressions are defined recursively as follows:

   (a) λ (the empty string) is a regular expression.

   (b) $\forall a \in \Sigma$, a is a regular expression.

   (c) If u and v are regular expressions, then $u + v$ is a regular expression. The + operator is the OR operator for regular expressions, and this represents u OR v.

   (d) If u and v are regular expressions, then uv is aregular expression (concatenation).

   (e) If u is a regular expression, then u^* is a regular expression. The * operator is called the *Kleene star* and represents zero or more repetitions of u.

   (f) If u is a regular expression, then u^+ is a regular expression. The + operator represents one or more repetitions of u, and is equivalent to uu^*.

6. More formally, the set of regular expressions, RE, is recursively defined as follows:

$$RE = \begin{cases} \lambda \\ a, \forall a \in \Sigma \\ uv, \forall u, v \in RE \\ u*, \forall u \in RE \\ u^+, \forall u \in RE \\ u + v, \forall u, v \in RE \end{cases} \tag{1}$$

7. Regular expression examples: (Draw the machines for at least one of these.)

   (a) Let L_5 be the set of strings having zero or more as followed by zero or more bs. Then $L_5 = a^*b^*$.

   (b) Let L_6 be the set of strings beginning with a single a, followed by at least one b, and ending with a single a. Then $L_6 = ab^+a$.

   (c) Remember L_4 was the set of strings ending in *bbb*. Then $L_7 = (a + b + c)^*bbb$ Note: The $(a + b + c)^*$ notation represents any combinations of as, bs, and cs, and can have any length including zero.

   (d) Σ^* is shorthand for *all possible strings composed from some alphabet.* Note that a useful language has some restrictions on the strings, so $L \subset \Sigma^*$.

8. Relationship between FSM and regular languages. Reminder: The job of a FSM is to *accept* strings in the language and *reject* strings not in the language. In other words, the FSM and the language are equivalent. They are two representations of the same thing.

9. Proving a language is regular. Not all languages are regular. If you can create a FSM for a given language, you have proven that the language is regular.

10. Example non-regular language: Let $L_7 = a^n b^n$, or strings having some number of as followed by the same number of bs. Discuss why this cannot be a *finite* state machine, as an infinite number of state are needed. This is the canonical non-regular language and points out a characteristic of FSMs: They have no memory. You know where you are, but not how you got there.

11. The Chomsky Hierarchy (may not get this far). Chomsky defined four classes of languages, and Comp Sci people took that classification as part of computationl theory.

    (a) Language: Regular, Machine: FSM, Charateristic: No memory.

    (b) Language: Context Free Machine: Stack Machine Characteristic: Has memory, Basis for all programming languages.

    (c) Language: Context Sensitive Machine: Turing Machine (finite tape) Characteristic: Defines NP-Complete problems

    (d) Language: Unrestricted Machine: Turing Machine (infinite tape) Characteristic: Defines all computable problems.