CSC444

October 26, 2014

Contents

1 Introduction

The title of this course is "Automata Theory and Formal Grammars".

This is an apt description since the purpose of the course is to discuss:

- Finite automata, regular expressions, the relationship between the two models, and the limitations of the models.
- Context-free grammars, pushdown automata, the relationship between the two models, and the limitations of the models.

All of these models are formalizations of computing devices.

1.1 Computing devices

Informal defn: A *computing device* is a machine that takes input, processes that input, and produces a resulting output and/or action.

Some common computing devices include:

- Computer
- Calculator

- Combination lock
- Elevator
- Vending machine
- Ventilation system thermostat

The *mechanical details* of each example varies significantly:

- Levers: Vending machine
- Circuits: Computer, calculator, elevator, thermostat, vending machine
- Tumblers: Combination lock
- Sensors: Vending machine, thermostat, elevator, computer

But we do not need to know how each device is built in order to understand it.

Example 1.1. Very few of us know how elevators are constructed, yet we use them every day to get to class and/or work.

It is often more instructive to find an *abstract way* to describe the behavior of the device.

Using this abstraction we can understand the device's behavior as well as compare its behavior to other similar devices, all without knowing the details of its construction.

1.2 Sample abstractions

Let us find some more abstract descriptions for the computing devices we have mentioned so far.

Calculator

Input: A numerical expression, that is, a combination of numbers and function symbols

Processing: Evaluate the numerical expression by applying the functions to the values

Output: The final value of the numerical expression

Combination lock

Input: A sequence of numbers and directional information

Processing: Match each number and direction with the lock's combination

Output: Unlock or no action

Elevator

Input: A sequence of floor requests, directional information

Processing: Make passes up and down the building to fulfill the requests; when possible, fulfill the requests in a FIFO manner

Output: A fulfilled request; a pass through the building

Vending machine

Input: Money or smart card, information about buttons/levers

Processing: Count money inserted and/or debit the smart card; compare against the amount needed for the item desired

Output: Item desired, change

Thermostat

Input: Building temperature(s), desired temperature(s)

Processing: Compare current temperature(s) with desired temperature(s)

Output: Increase or decrease the work of the ventilating system; make no changes

While instructive, these descriptions are somewhat imprecise.

Examples 1.2.

- How would a vending machine handle giving change when a customer inserts more than is necessary for an item?

- Which method would be used by the calculator to determine the precedence of operations for the numerical expression?

We need to use mathematics to ensure that our descriptions are both abstract (do not depend on the implementation of the device) and precise (have few, if any, ambiguities).

1.3 Finite state machines

To introduce finite-state machines, consider the example of a simple newspaper vending machine.

Rules for the newspaper vending machine:

- The cost of the newspaper is \$0.30
- The machine accepts nickels, dimes, and quarters
- Once \$0.30 has been inserted, the lid can be lifted and the newspaper removed
- If more than \$0.30 is inserted, the extra money will be lost

All finite-state machines, including our vending machine, record information about the world using a set of states.

In particular, the newspaper vending machine will use a set of states to record how much money remains to be inserted before the lever will be released and the newspaper can be removed.

Thus, the *newspaper vending machine states* are:

- Needs \$0.30: Before any coins are input
- Needs \$0.25: After a nickel has been input
- Needs \$0.20: After two nickels or a dime have been input
- Needs \$0.15: After 3 nickels, or a dime and a nickel have been input

- Needs \$0.10: After 4 nickels, 2 dimes or a dime and 2 nickels have been input
- Needs \$0.05: After 5 nickels, or 2 dimes and a nickel, or 1 dime and 3 nickels or a quarter have been input
- Needs \$0.00: After at least \$0.30 has been input

Two of these states are special:

- The machine starts in the *initial state*, that is, the "Needs \$0.30" state
- The machine will release a newspaper only when it is in the *accepting state*, that is, the "Needs \$0.00" state

1.3.1 Graphical representation

A finite state machine can be represented in a graphical form.

The representation:

- Each state corresponds to a circle. A label indicates what state is represented by the circle.
- The initial state is represented by an arrow.
- The accepting state is indicated by double lines.
- Lines with arrows are used to indicate how to move from state to state.

Example 1.3. Draw the state diagram for the vending machine.

To understand the representation, consider a few *sample computations*:

- Input: Nickel, dime, dime, nickel
- Output: Newspaper is available

- Input: Quarter, nickel, quarter

- Output: Newspaper is available

- Input: Nickel, nickel, dime
- Output: Newspaper is not available

1.3.2 Issues

There are two issues that we did not address in our example. First, we do not allow our vending machines to *make change*. This is a rather user-unfriendly thing to do. However, allowing the machine to make change complicates the design of the automaton. We will not consider it now.

Second, we *cannot produce a finite-state machine* for all of the sample devices we discussed earlier. In particular, a calculator cannot be implemented this way. The model is not powerful enough. The issue here is that finite-state machines cannot represent devices that must remember an arbitrary amount of information. Since there is no fixed bound on the size of arithmetic expressions, a calculator cannot be constructed this way.

1.4 Pushdown automata

To motivate the next more powerful machine beyond finite-state machines, consider the problem of determining whether an arithmetic expression has a balanced set of parentheses.

1.4.1 Balanced parentheses

Suppose that we wish to check whether the parentheses in an arithmetic expression are balanced, that is, there are as many left parentheses as right parentheses.

Examples 1.4.

- ((()(()))) : balanced
- ((()()) : not balanced

- ()) : not balanced

Consider the following *algorithm*:

1. Initialize a counter to 0
2. Increment the counter when a left parenthesis is read.
3. Decrement the counter when a right parenthesis is read.
4. Repeat until one of the following conditions occurs:
   (a) If the counter becomes negative, output "not balanced".
   (b) If the expression is completely read, do one of the following:
      i. If the counter is zero, output "balanced"
      ii. If the counter is non-zero, output "not balanced"

Why can we *not design a finite-state machine* for this algorithm?

Evaluating the expression may require us to store a value as large as half the number of total parentheses before making a decision. (Actually, as large as log n where n is the number of parentheses, but the same argument holds . . .)

Example 1.5. (((((((((())))))))))

Since the number of parentheses is arbitrary, a machine with a fixed number of states cannot do this.

1.4.2 An extension

However, if we make one minor modification to the finite-state model, we can solve the problem. Suppose that our finite-state machine is allowed access to a stack of finite, but arbitrary, depth. As the machine changes state it can push items onto the stack or pop items off the stack. The machine, however, only has access to the top of the stack. It cannot "peek" into the middle.

A *pushdown automaton* is a finite automaton (i.e. finite-state machine) that has a stack as auxiliary storage.

To *solve the balanced parentheses* problem:

1. Start with an empty stack.
2. Whenever a left parenthesis is read, push a "+" symbol onto the stack.
3. Whenever a right parenthesis is read, pop a "+" symbol off the stack.
4. Continue until one of the following conditions is met:
   (a) An attempt is made to pop an empty stack. Output "not balanced".
   (b) The input is empty, but the stack is not. Output "not balanced".
   (c) The input is empty, and the stack is empty. Output "balanced".

1.5 The course

What will we do in this course?

- Learn about each of the models previously mentioned.
- Characterize the relationship between each model in a formal way.

A lesser priority:

- Discuss applications of these models.
- Implement Finite State Automata.

1.5.1 A hierarchy of models

This course will introduce several ways of describing and representing languages. As we introduce a new method we will show that either:

- It is *strictly more powerful* that the representations we have seen before in the sense that it can describe everything the old method could plus at least one language the old method could not; or
- It has the *exact same expressibility* as a representation we've already seen.

The problem is that anything we can manage to describe finitely will "miss" the vast majority of languages.

What do we mean by *a finite description*?

- We must be able to write it down in a finite amount of time, thus it must be a string over some alphabet
- Different languages must have different representations or we have not captured the essence of the language

We can certainly achieve the above for any finite set, since all we have to do is list out all the strings in the language. The tricky part is trying to find specifications for infinite languages.

We won't use the fact that we have "missed" languages very much, but in 544 it will be used to find a set missed by all representations introduced in both courses.

2 Mathematical foundations

The purpose of this course is to investigate the power of and relationship between several fundamental models of computation.

We will do that by formally defining those models and then using mathematics to reason about the expressiveness and limitations of the result.

2.1 Languages

In this course we will measure the power of a model by the set of languages it accepts or recognizes.

We will define what we mean by accepting or recognizing a language as we encounter each new model, but for now let's consider what we mean by languages.

Definition 2.1. A *language* is a set of strings over a given alphabet.

Let's briefly review what we know (or should know) about sets.

2.2 Sets

Definition 2.2. A *set* is a collection of objects. The objects comprising a set are called its elements or members.

Examples 2.3.

- months = { January, February, . . . , December }
- $\mathbb{N} = \{0, 1, 2, ...\}$
- even = $\{\, x \in \mathbb{N} \mid$ and x is divisible by 2 $\}$

You should know the following symbols and their meanings: $\in$, $\notin$, $\subset$, $\subseteq$, $\not\subset$

You should also be familiar with the basic set operations of union, intersection, and difference and know what it means for two sets to be disjoint.

2.3 Alphabets and strings

Definition 2.4. An *alphabet* is a finite set of symbols.

Examples 2.5.

- { a, b, . . . , z }
- { a, b, . . . , z, ä, ö, ü, ß }
- { 0,1 }

Alphabets are usually denoted by Σ. Often the alphabet will be understood from context.

Definition 2.6. A *string* over an alphabet is a finite sequence of symbols from the alphabet.

Examples 2.7.

- sour

- süß
- 010101110

As we can note above, strings are written without the use of brackets or commas, with the letters simply placed one after another

Notation. The *length* of a string is the length of its sequence of characters. If w is a string, the length of w is denoted $|w|$.

The *empty string*, usually denoted by ε, (sometimes denoted λ or Λ) is the empty sequence, that is, the unique string of length 0.

The set of all strings, including the empty string, over a given alphabet Σ is denoted by Σ^*. The set of all strings of length $\leq n$ for some integer n is denoted by $\Sigma^{\leq n}$, and the set of all strings of length exactly n is denoted by $\Sigma^{=n}$.

Note. The characters may be duplicated

Examples 2.8.

- $|\text{sour}| = 4$
- $|\text{süß}| = 3$
- $|010101110| = 9$
- $|\varepsilon| = 0$

In a slight abuse of notation, we will sometimes treat the name of a string as a function from the integers to alphabet symbols.

Example 2.9. If $w = \text{sour}$ then

- $w(1) = s$,
- $w(2) = o$,
- $w(3) = u$ and
- $w(4) = r$.

Question 2.10.
Is this function necessarily one-to-one?

Review: A function $f : A \to B$ is one-to-one if for any two distinct elements $a, a' \in A$, $f(a) \neq f(a')$.

Answer 2.11.
No. If $w = 010101110$ then $w(1) = w(3) = w(5) = w(9) = 0$.

If a string has duplicated symbols, each duplication will be referred to as an occurrence of the symbol.

Two strings can be concatenated to form a third. We denote the *concatenation* operator by $\cdot$

Examples 2.12.

- $001 \cdot 11100 = 00111100$
- fuzzy $\cdot$ kitty = fuzzykitty
- $\varepsilon \cdot w = w$, for any string w

Notation. We will write xy to represent $x \cdot y$

Other terminology:

- v is a *substring* of a string w if there are strings x and y s.t. $w = xvy$
- if $w = uv$ for some string u then v is a *suffix* of w
- if $w = uv$ for some string v then u is a *prefix* of w

Degenerate cases:

- ε is a substring of any string
- Any string is a substring of itself
- ε is a prefix and suffix of any string
- Any string is a prefix and suffix of itself

2.4 Induction

As a review, a proof by induction, has three parts:

- A basis step
- An inductive hypothesis
- The inductive step

If this doesn't bring back lots of memories, please talk to me for references.

We will do proof by induction in this course, but more commonly we will use a form of definition that uses induction.

2.4.1 Definition by induction

Definition 2.13. For each string w and each natural number i, the string w^i is defined as:

- $w^0 = \varepsilon$
- $w^{i+1} = w^i \cdot w$ for each $i \geq 0$

Example 2.14. Let $w =$ he

- $w^1 =$ he
- $w^2 =$ hehe
- $w^4 =$ hehehehe

In a definition by induction we have two parts: one (or more) basis cases and then one (or more) inductive parts.

You've probably already seen a definition by induction and not called it that.

Example 2.15. The i^{th} Fibonacci number $f(i)$

- $f(0) = 0$

- $f(1) = 1$
- $f(i) = f(i-1) + f(i-2)$, for $i \geq 2$

Let's do another definition by induction.

Definition 2.16. The *reversal* of a string w, denoted $w^{\mathcal{R}}$, is:

- if $|w| = 0$ then $w^{\mathcal{R}} = w = \varepsilon$
- if $|w| = n + 1$, for $n \geq 0$, then $w = ua$ for some $a \in \Sigma$ and $w^{\mathcal{R}} = au^{\mathcal{R}}$

Less formally, the reversal of a string is the string "spelled backward".

Definitions by induction lend themselves very well to proofs by induction.

2.5 Languages

Definition 2.17. A *language* is a set of strings over a given alphabet Σ, that is, a subset of Σ^*.

Since a language is simply a set, we can specify a finite language by listing all of its strings.

However, most languages are infinite, so we must find a shorthand way to represent the language.

We will typically specify languages using the following form:
$L = \{\, w \in \Sigma^* \mid w \text{ has property } P \,\}$

Examples 2.18.

- $\{\, w \in \{0,1\}^* \mid w \text{ has at least one } 0 \,\}$
- $\{\, w \in \{a, e, i, o, u\}^* \mid w \text{ is an English word} \,\}$

2.5.1 Set operations on languages

Since languages are sets, they can be combined using set operations.

Notation. When the alphabet Σ is understood from context, we will write the complement of A as $\overline{A}$ instead of $\Sigma^* - A$.

Certain operations apply only to languages:

Concatenation

If L_1 and L_2 are languages over Σ then $L = L_1L_2$ where $L = \{\, w \in \Sigma^* \mid w = xy \text{ for some } x \in L_1 \text{ and } y \in L_2 \,\}$

Examples 2.19. Let $\Sigma = \{0, 1\}$ and let $L_1 = \{\, w \in \Sigma^* \mid |w| \text{ is even} \,\}$ and $L_2 = \{\, w \in \Sigma^* \mid |w| \text{ is odd} \,\}$.

- $L = L_1L_2 = L_2$
- $L = L_1L_1 = L_1$
- $L = L_2L_2 = L_1 - \varepsilon$

Kleene star

Given a language L, L^* is the language that results from concatenating zero or more strings from L.

More formally, $L^* = \{\, w \in \Sigma^* \mid w = w_1 ... w_k \text{ for some } k \geq 0 \text{ and some } w_1, ..., w_k \in L \,\}$.

Examples 2.20.

- $L = \{0,1\}^*01\{0,1\}^* = \{\, w \in \{0,1\}^* \mid w \text{ contains the substring } 01 \,\}$
- $\emptyset^* = \{\varepsilon\}$

Notation. $L^+ = LL^*$

What is the difference between L^* and L^+?

2.6 Relations

Definition 2.21. An *n-ary relation* on sets $A_1, \ldots, A_n$ is a subset of $A_1 \times A_2 \times ... \times A_n$. A 2-ary relation is called a binary relation.

Examples 2.22. Let $A = \{1, 2, 3, 4\}$ and consider binary relations on $A \times A$

- $R_1 = \{(1, 2), (1, 1), (3, 4), (4, 1)\}$
- $R_2 = \{(1, 4), (3, 4), (2, 2)\}$
- $R_3 = \{(1, 1), (1, 2), (2, 1), (3, 4)\}$

You can associate a *directed graph* with a binary relation R on a set and itself. You do this by drawing a node for each element in the set and then drawing an arc from one node a_1 to another node a_2 if $(a_1, a_2) \in R$.

Let A be a set. The following are several important properties a binary relation R on $A \times A$ can have:

- R is *reflexive* if $(a, a) \in R$ for each $a \in$ A
- R is *symmetric* if $(a, b) \in R$ implies that $(b, a) \in R$
- R is *transitive* if whenever $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$

Definition 2.23. A relation that is reflexive, symmetric and transitive is called an *equivalence relation*.

Each of the three properties above has a *graphical interpretation*.

Reflexive Each node has a self-loop

Symmetric Either two nodes are unrelated or there is an arc in both directions between them

Transitive Whenever there is a way to get from node a_1 to node a_2 then there is a direct arc from a_1 to a_2

Definition 2.24. Let A be a set. The *reflexive, transitive closure* of a relation R on $A \times A$ is the relation $R^* = \{\, (a, b) \in A \times A \mid$ and there is a path from a to b in $R\,\}$.

Intuitively, R^* is the smallest possible relation that contains all the pairs of R and is reflexive and transitive.

Thus, to get R^* from R we need to add in just enough arcs to get both properties.

Example 2.25. Let $A = \{1, 2, 3, 4\}$ and let $R = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$.
Then $R^* = \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4)\}$.

3 Deterministic finite automata

We now introduce the first model of computation we will consider, that of a deterministic finite automaton.

3.1 Basic operation

A finite automaton (FA), like all other models we will consider in the course, takes strings as input.

The input string is written on an *input tape*.

The tape is divided into squares, and each square contains a single character from a specified alphabet.

Computation is done according to the *finite control*.

- Inside the finite control, information is recorded using a set of states.
- The finite control can access the input via a movable reading head.

Initially the reading head is at the first tape square, and the finite control is in a specified initial state.

At regular intervals the automaton:

- Reads the character at the end of the reading head;
- Enters a new state based on its current state and the character it just read;
- Moves the reading head one square to the right.

This process is repeated until the tape-head reaches the end of the input string.

Once the input has been read, the FA *accepts the string* it has just read if it is in one of the accepting states. Otherwise it *rejects (or does not accept) the string.*

The *language accepted by the FA* is the set of strings that cause it to enter the accepting state.

3.2 Definition

Definition 3.1. A *deterministic finite automaton* (DFA) is a quintuple $M = (K, \Sigma, \delta, s, F)$ where

- K is a finite set of states,
- Σ is a finite alphabet,
- $s \in K$ is the initial state,
- $F \subseteq K$ is the set of final states, and
- $\delta : K \times \Sigma \to K$ is the transition function.

The rules by which M chooses the next state are encoded in the transition function.

If M is in state $q \in K$ and the symbol read from the input tape is $a \in \Sigma$, then $\delta(q, a) \in K$ is the uniquely determined state that M enters next.

Note. This definition is for a *deterministic* finite automaton because δ is a function, and not a relation. We'll contrast that with a non-deterministic finite automaton later.

3.3 Computation of a FA

The computation of a FA consists of a sequence of configurations that represent the status of the machine at successive moments.

Things to keep track of:

- Finite control
- Reading head
- Input tape

Note. Since the FA cannot move its reading head left, the part of the input tape that has already been viewed cannot affect the future computation.

Thus the configuration of a FA is determined by the current state of the finite control and the unread part of the input tape.

Definition 3.2. A *configuration of a DFA* $M = (K, \Sigma, \delta, s, F)$ is any element of $K \times \Sigma^*$.

Note. The configuration (q, ε) signifies that the FA has viewed all the input and is in state q.

Consider a binary relation $\mapsto_M$ that holds between two configurations of M iff the machine can pass from one configuration to the other as a result of a single move.

More formally, if (q, w) and (q', w') are two configurations of M, then $(q, w) \mapsto_M (q', w')$ iff $w = aw'$ for some symbol $a \in \Sigma$ and $\delta(q, a) = q'$.

In this case, we say that (q, w) *yields* (q', w') *in one step.*

Note that $\mapsto_M$ is in fact a function on $K \times \Sigma^+$ to $K \times \Sigma^*$ since every configuration except the one with input string ε has a uniquely determined next configuration.

Let $\mapsto_M^*$ be the reflexive, transitive closure of $\mapsto_M$.

Then $(q, w) \mapsto_M^* (q', w')$ is read as (q, w) *yields* (q', w') (after some number, possibly zero, of steps).

q	σ	δ(q,σ)
q_0	0	q_1
q_0	1	q_1
q_1	0	q_0
q_1	1	q_0

$\mathcal{L}(M) = \{\, w \mid |w| \text{ is even} \,\}$

What would M' look like so that $\mathcal{L}(M') = \{\, w \mid |w| \text{ is odd} \,\}$?

Example 3.7. $K = \{q_0, q_1, q_2, q_3\}$, $s = q_0$, $F = \{q_2\}$, and δ is given by the following table:

q	σ	δ(q,σ)
q_0	0	q_1
q_0	1	q_2
q_1	0	q_0
q_1	1	q_3
q_2	0	q_3
q_2	1	q_0
q_3	0	q_2
q_3	1	q_1

q_0 = "even 1's, even 0's", q_1 = "odd 0's, even 1's", q_2 = "even 0's, odd 1's", q_3 = "odd 1's, odd 0's"

$\mathcal{L}(M) = \{\, w \mid w \text{has an even number of 0's and an odd number of 1's} \,\}$

What other languages could we compute if we change M slightly?

Definition 3.3. A string $w \in \Sigma^*$ is said to be *accepted* by a FA M iff there is a state $q \in F$ s.t. $(s, w) \mapsto_M^* (q, \varepsilon)$.

Definition 3.4. The *language accepted* by a FA M is the set of all strings accepted by M.

3.4 State diagrams

Before we do some examples to clarify these definitions, it is helpful to introduce a more graphical representation for FA.

A *state diagram* for a FA is a digraph with certain additional information.

The states of the FA are represented by nodes, and there is an arc labeled with a from state q to state q' whenever $\delta(q, a) = q'$.

Final states are indicated by double circles, and the initial state is indicated using a bracket.

3.5 Sample finite automata

Let $\Sigma = \{0, 1\}$ in the following examples.

Example 3.5. $K = \{q_0, q_1\}$, $s = q_0$, $F = \{q_1\}$, and δ is given by the following table:

q	σ	δ(q,σ)
q_0	0	q_1
q_0	1	q_0
q_1	0	q_1
q_1	1	q_0

$\mathcal{L}(M) = \{\, w \mid w \text{ ends in } 0 \,\}$

Example 3.6. $K = \{q_0, q_1\}$, $s = q_0$, $F = \{q_0\}$, and δ is given by the following table:

q_0 = "even", q_1 = "odd"