

CSI 3104

Formal Languages

Study Guide

winter 2024

Languages

- Languages considered as symbols with formal rules, and not as expressions of ideas in the minds of humans
- We're interested in the form of the strings of symbols
- Start with only one finite set of fundamental symbols out of which we build structures. This set is called the alphabet.
- The language is a certain specified set of strings of symbols from the alphabet
- Words are the strings that are permissible in the language
- Null string have no letters, denote by Λ .
- Null word is the word that has no letters, also denote by Λ .
- Two words are considered the same if all their letters are the same and in the same order.
- We usually do not allow Λ to be part of alphabet of any language
- Language with no words, denote by \emptyset
- $\Lambda \notin \emptyset$
- $\Lambda + \emptyset = \Lambda$ if $\Lambda \notin L$
- $\Lambda + \emptyset = \emptyset$

Defining Languages

Rules can be of 2 kinds:

- Can tell us how to test if a string of alphabet letter is a valid word
- Can tell us how to construct all the words in the language by some clear procedures

To define a language, first define an alphabet, then define the set of words permitted.

Length

- Length of a string is the number of letter in the string.
- For any word w in any language, if $\text{length}(w)=0$, then $w=\Lambda$

Reverse

- If a is a word in some language L , then $\text{reverse}(a)$ is the same string of letters spelled backward, even if this backward string is not a word in L

Palindrome

Define new language called Palindrome over alphabet $\Sigma = \{a, b\}$

$$\text{PALINDROME} = \{\Lambda, \text{and all strings } x \text{ s.t. } \text{reverse}(x)=x\}$$

Kleene Closure

Given the alphabet Σ , we define a language in which any string from Σ^* is a word, even the null string Λ . We call this the closure of the alphabet Σ , denote this language by Σ^*

If S is a set of words, S^* is the set of all finite strings formed by concatenating words from S , where any word may be used as often as we like, and where Λ is also included

Positive Closure

- If we wish to modify the concept of closure to refer only the concatenation of some strings from a set S , we use the notation $+$ instead of $*$.
- Plus operation is called positive closure

Theorem 1

For any set S of strings, we have $S^+ = S^* - \{\Lambda\}$

Recursive Definition

3-step process

- Specify some basic objects in set
- Give rule for constructing more objects in the set from basic ones we already know
- Declare that no objects except those constructed in this way are allowed in the set

Theorem 2

An arithmetic expression cannot contain the character \$

Theorem 3

No arithmetic expression can begin or end with the symbol /

Theorem 4

No arithmetic expression can contain the substring //

Kleene star

Notation: x^*

$$x^* = \Lambda \text{ or } x \text{ or } x^2 \text{ or } x^3 = x^n \text{ for some } n=0,1,2,3$$

Plus sign

Expression: $x+y$

This means either x or y

Regular Expressions

Set of regular expressions defined by the following rules:

- Every letter of the alphabet Σ can be made into a regular expression by writing it in boldface; Λ itself is a regular expression.
- If r_1 and r_2 are regular expressions, then so are
 - (i) (r_1)
 - (ii) $r_1 r_2$
 - (iii) $r_1 + r_2$
 - (iv) r_1^*
- Nothing else is a regular expression

Product Set

If S and T are sets of strings of letters, we define the product set of strings of letters to be

$$ST = \{\text{all combinations of a string from } S \text{ concatenated with a string from } T \text{ in that order}\}$$

Language associated with regular expressions

- The language associated with the regular expression that is just a single letter is that one-letter word alone, and the language associated with Λ is just $\{\Lambda\}$, a one-word language
- If r_1 is a regular expression associated with the language L_1 and r_2 is a regular expression associated with the language L_2 , then:

(i) The regular expression $(r_1)(r_2)$ is associated with the product $L_1 L_2$, that is the language L_1 times the language L_2 :

$$\text{language}(r_1 r_2) = L_1 L_2$$

(ii) The regular expression $r_1 r_2$ is associated with the language formed by the union of L_1 and L_2

$$\text{language}(r_1 + r_2) = L_1 + L_2$$

(iii) The language associated with the regular expression $(r_1)^*$ is L_1^* , the kleene closure of the set L_1 as a set of words:

$$\text{language}(r_1^*) = L_1^*$$

Theorem 5

If L is a finite language, then L can be defined by a regular expression. In other words, all finite languages are regular

Finite Automaton

A finite automaton is a collection of three things:

- A finite set of states, one of which is designated as the initial state, called the start state, and some (maybe none) of which are designated as final states
- An alphabet Σ of possible input letters
- A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go next

Transition Tables

The transition list can be summarized in a table format in which each row is the name of one of the states, and each column is a letter of the input alphabet

Transition Diagram

- Represent each state by a small circle
- Draw arrows showing to which other states the different input letters will lead us. Label these arrows with the corresponding input letters
- If a certain letter makes a state go back to itself, indicate this by a loop
- Indicate start state with minus sign or label with the word start
- Indicate final states by plus signs, or by labeling with word final

Transition Graphs

Collection of three things:

- A finite set of states, at least one of which is designated as the start state (-), and some (maybe none) of which are designated as final states (+).
- An alphabet Σ of possible input letters from which input strings are formed
- A finite set of transitions (edge labels) that show how to go from some states to some others, based on reading specified substrings of input letters (possibly even Λ)

FA vs. TG

1) Finite Automation

- One initial state
- Any number of final states
- One letter as input from every state
- All states must have the same number of outgoing edges as number of letters in the alphabet
- It is deterministic as it takes single letter inputs only

2) Transition Graph

- One or more initial states
 - Any number of final states
 - Input can be any combination of letters from alphabet. Not restricted to single letters.
 - No restrictions on number of outgoing edges from each state
 - Nondeterministic since it can take substrings of any combination and any number of letters
- Every FA is a TG, but not every TG is a FA

Generalized Transition Graphs

Collection of three things:

- A finite set of states, of which at least one is a start state and some (maybe none) are final states.
- An alphabet Σ of input letters.
- Directed edges connecting some pairs of states, each labeled with a regular expression.

Kleene's Theorem (Theorem 6)

Any language can be defined by regular expression, or finite automation, or transition graph can be defined by all three methods

Transition Graph to Regular Expression

- Create a unique, unenterable minus state and a unique, unleaveable plus state.
- One by one, in any order, bypass and eliminate all the non-minus or non-plus states in the TG. A state is bypassed by connecting each incoming edge with each outgoing edge. The label of each resultant edge is the concatenation of the label on the incoming edge with the label on the outgoing edge.
- When 2 states joined by more than 1 edge, add their labels

Regular Expressions to FA

Rule 1: There is an FA that accepts any particular letter of the alphabet. There is an FA that accepts only the word Λ .

Rule 2: If there is an FA called FA₁ that accepts the language defined by the regular expression r₁, and there is an FA called FA₂ that accepts the language defined by the regular expression r₂, then there is an FA that we shall call FA₃ that accepts the language defined by the regular expression (r₁+r₂).

Algorithm to construct FA₃:

- 1) Start with 2 machines FA₁ with states x₁, x₂, x₃, ..., and FA₂ with states y₁, y₂, y₃, ..., we construct a new machine FA₃ with states z₁, z₂, z₃, ..., where each z_i is of the form x something or y something
- 2) The combination state x_{start} or y_{start} is the start state of the new machine FA₃
- 3) If either the x part or the y part is a final state, then the corresponding z is a final state.
- 4) To go from one state z to another by reading a letter from the input string, we observe what happens to the x part and what happens to the y part and go to the new state z accordingly. In short, z_{new} after reading letter p = (x_{new} after reading letter p on FA₁) or (y_{new} after reading letter p on FA₂)

Rule 3: If there is an FA₁ that accepts the language defined by the regular expression r₁, and there is an FA₂ that accepts the language defined by the regular expression r₂, then there is an FA₃ that accepts the language defined by the regular language (r₁r₂).

Algorithm to construct FA₃:

- 1) Create a state z for every state of FA₁ that we may go through before arriving at a final state.
- 2) For each final state x_{final} of FA₁, add a state z = x_{final} or y_i, where y_i is the start state of FA₂.
- 3) From the states added in step 2, add states

$$z = \begin{cases} x \text{ something indicating we are still continuing on FA}_1 \\ \text{OR} \\ a \text{ set of } y \text{ something indicating we are on FA}_2 \end{cases}$$

- 4) Label every state z that contains a final state from FA₂ as a final state

Rule 4: If r is a regular expression and FA₁ is a finite automaton that accepts exactly the language defined by r, then there is an FA, called FA₂, that will accept exactly the

language defined by r*

Algorithm to construct FA₂:

- 1) Language defined by r* must always contain Λ . To accept Λ , we must indicate that the start state is also the final state. Must be done carefully.

- 2) Each z-state (of FA₂) corresponds to some collection of x-states (of FA₁). We must remember each time we reach a final state, it is possible that we have to start over again at x_i.
- 3) The transmissions from one collection of x-states to another based on reading certain input letters is determined by the transition rules for FA₁.
- 4) There are only finitely many possible collections of x-states, so the machine FA₂ has only finitely many states.

Nondeterministic Finite Automata

A nondeterministic finite automaton (or NFA) is a TG with a unique start state and with the property that each of its edge labels is a single alphabet letter.

- The regular deterministic finite automata are referred to as DFAs, to distinguish them from NFAs.
- As a TG, an NFA can have arbitrarily many a-edges and arbitrarily many b-edges coming out of each state.
- An input string is accepted by an NFA if there exists any possible path from - to +

Theorem 7

For every NFA, there is some FA that accepts exactly the same language.

Algorithm (FA from NFA)

- Each state in FA is a collection of states from the original NFA
- For every state z in the FA, the new state that an a-edge (or b-edge) will take us to is just the collection of possible states that result from being in x_i and taking the a-edge, or being in x_j and taking the a-edge and so on.
- The start state of the FA that we are constructing is the same old start state we had to begin with in the NFA. Its a-edge (or b-edge) goes to the collection of the x-states that can be reached by an a-edge (or a b-edge) from the start state in the NFA.
- Since there may be no a-edges (or no b-edges) leaving the start state in the NFA (or leaving any particular state), we must add a state ϕ in the FA for the a-edge (or the b-edge) to read in this case.

- The ϕ state in the FA must have an a,b-loop going back to itself
- A state in the FA is a final state if the collection of the x-states that it represents has an old final state in it.

NFAs and Kleene's Theorem

We can use NFAs to prove rules 1-4 of part 3 of Kleene's Theorem

Rule 1: There are FAs for the languages {a}, {b}, and {Λ}

Proof:

Step 1: The above three languages can all be accepted respectively by the NFAs below

$$\textcircled{a} \xrightarrow{\quad} \textcircled{+} \quad \textcircled{b} \xrightarrow{\quad} \textcircled{+} \quad \textcircled{\Lambda} \xrightarrow{\quad} \textcircled{+}$$

Step 2: By Theorem 7, for every NFA, there is an equivalent FA. Hence, there must be FAs for these three languages as well.

Rule 2: Given FA₁ and FA₂, we shall present an algorithm for constructing a union machine FA₁ + FA₂

Algorithm:

- 1) Introduce a new and unique start state with two outgoing a-edges and two outgoing b-edges but no incoming edges. Connect them to the states that the start states of FA₁ and FA₂ are connected to. Do not erase the start states of FA₁ and FA₂, but erase their minus signs, leaving all their edges intact. The new machine is an NFA that clearly accepts exactly language(FA₁) + language(FA₂)
- 2) Using the algorithm of Theorem 7, convert the NFA into an FA

Moore Machine

Collection of five things:

- Finite set of states q₀, q₁, q₂, ..., where q₀ is designated as start state
- Alphabet of letters for forming the input string $\Sigma = \{a, b, c, \dots\}$
- Alphabet of possible output characters $T = \{0, 1, 2, \dots\}$
- A transition table that shows for each state and each input letter what state is reached next
- An output table that shows what character from T is printed by each state as it is entered

Mealy Machine

Collection of four things:

- A finite set of states q₀, q₁, q₂, ..., where q₀ is designated as the start state.
- An alphabet of letters for forming the input string $\Sigma = \{a, b, \dots\}$
- An alphabet of possible output characters $T = \{0, 1, \dots\}$
- A pictorial representation with states represented by small circles and directed edges indicating transition between states

- Each edge is labeled with a compound symbol of the form i/o, where i is an input letter and o is an output character

- Every state must have exactly one outgoing edge for each possible input letter

- The edge we travel is determined by the input letter i. While traveling on the edge, we must print the output character o.

Moore = Mealy

Given the Mealy machine Me and the Moore machine Mo (which prints the automatic start state character x), we say that these two machines are equivalent if for every input string, the output string from Mo is exactly x concatenated with the output string from Me

Theorem 8

If Mo is a Moore machine, then there is a Mealy machine Me that is equivalent to Mo.

Algorithm to turn Mo into a Me

- Consider a particular state in Mo, say state q₊, which prints a certain character, say t
- Consider all the incoming edges to q₊. Suppose these edges are labeled with a, b, c, ...
- Relabel edges as a/t, b/t, c/t, ... and erase the t from the state q₊. This means we shall be printing a t on the incoming edges before we enter q₊
- We leave the outgoing edges from q₊ alone. They will be relabeled to print the character associated with the state to which they lead.
- If we repeat this procedure for every state q₀, q₁, ..., we turn Mo into its equivalent Me.

Theorem 9

For every Mealy machine Me, there is a Moore machine Mo that is equivalent to it.

Algorithm to turn Me into Mo

- We cannot just do the reverse of the previous algorithm. If we try to push the printing instruction from the edge (as it is in Me) to the inside of the state (as it should be for Mo), we may end up with a conflict: Two edges may come into the same state but have different printing instructions.
- What we need are 2 copies of q₀, one that prints a 0 (labeled as q₀'/0), and the other that prints a 1 (labeled as q₀'/1). Hence,
 - The edges a/0 and b/0 will go into q₀'/0
 - The edge b/1 will go into q₀'/1
- The arrow coming out of each of these two states must be the same as the edges coming out of q₀ originally
- If all the edges coming into a state have the same printing instruction, we simply push that printing instruction into the state

An edge that was a loop in M_e may become two edges in M_o , one that is a loop and one that is not.

If there is ever a state that has no incoming edges, we can assign it any printing instruction we want, even if this state is the start state.

If we have to make copies of the start state in M_e , we can let any of the copies be the start state in M_o , because they all give the identical directions for proceeding to other states.

Having a choice of start states means that the conversion of M_e into M_o is NOT unique.

Repeating this process for each state of M_e will produce an equivalent M_o . The proof is completed.

Regular Languages

A language that can be defined by a regular expression is called a regular language.

Not all languages are regular.

Theorem 10

If L_1 and L_2 are regular languages, then $L_1 \cup L_2$, $L_1 L_2$, and L^* are also regular languages.

$L_1 \cup L_2$ is the language of all words in either L_1 or L_2 .

$L_1 L_2$ is the product language of all words formed by concatenating a word from L_1 with a word from L_2 .

L^* is the language of all words that are the concatenation of arbitrarily many factors from L_1 .

Often expressed as "The set of regular languages is closed under union, concat, and Kleene closure."

Complements

If L is a language over the alphabet Σ , we define its complement L' to be the language of all strings of letters from Σ that are not words in L .

Theorem 11

If L is a regular language, then L' is also a regular language. In other words, the set of regular languages is closed under complementation.

Theorem 12

If L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is also a regular language. In other words, the set of regular languages is closed under intersection.

Non-Regular Language

A non-regular language is a language that cannot be defined by a regular expression.

- By Kleene's theorem, they also cannot be accepted by any FA or TG.

Pumping Lemma (Theorem 13)

Let L be any regular language that has infinitely many words. Then there exists some three strings x, y , and z ($y \neq \lambda$) such that all the strings of the form

$$xy^n z \text{ for } n=1,2,3,\dots$$

are words in L .

Pumping Lemma V2 (Theorem 14)

Let L be an infinite language accepted by an FA with N states. Then, for all words w in L that have more than N letters, there are strings x, y , and z , where y is not null and $\text{length}(x) + \text{length}(y)$ does not exceed N , such that $w = xyz$ and all strings of the form

$$xy^n z \text{ for } n=1,2,3,\dots$$

are in L .

Equivalence

Objective: We want to know whether two regular expressions define the same language and whether two FAs accept the same language.

The above questions are the same. We can convert FAs into regular expressions and use a procedure to decide if two regular expressions define the same language. We can do it the other way too.

Effective Solution and Decision Procedure

We say that a problem is effectively solvable if there is an algorithm that provides the answer in a finite number of steps, no matter what the particular inputs are.

An effective solution to a problem that has a yes or no answer is called a decision procedure. A problem that has a decision procedure is called decidable.

Decision Procedure for Equivalence

Given L_1 and L_2 that are defined by either FAs or regular expressions. We can construct an FA that accepts the language

$$(L_1 \cap L_2') + (L_2 \cap L_1')$$

This FA accepts words that are in L_1 but not L_2 , or in L_2 but not L_1 .

If L_1 and L_2 are the same language, then this FA accepts no words at all. Otherwise, L_1 and L_2 are not the same languages.

Question: How to determine if an FA accepts any words at all.

Method 1 (Decide whether an FA accepts words)

- Convert FA into regular expression
- Every regular expression defines some words:
 - Delete all stars
 - For each $+$, remove right half and the $+$ sign
 - When there are no more $+$ signs, we remove parentheses. We get some word.

To decide whether two languages L_1 and L_2 are the same, build an FA for

$$(L_1 \cap L_2') + (L_2 \cap L_1')$$

Then convert FA into regular expression

- If process is completed and produces a regular expression, then the regular expression defines some words
- If process breaks down, then $L_1 = L_2$

3 ways process may break down:

- Machine has no final states
- Final state disconnected from start state
- Final state unreachable from start state

Method 2

We want to see whether or not there is any path from $-$ to $+$. If there is any path, the machine must accept some words.

We can use the following procedure:

- Paint the start state blue
- From every blue state, follow each outgoing edge and paint the destination edge blue, then delete the edge from the machine.
- Repeat step 2 until no new state is painted blue, then stop.
- When the procedure has stopped, if any of the final states are painted blue, then the machine accepts some words; if not, it does not accept any word.

Theorem 17

Let F be an FA with N states.

Then, if F accepts any words at all, it accepts some word with fewer than N letters.

Method 3

Test all the words with fewer than N letters by running them on the FA.

If the FA accepts none of them, then it accepts no words at all.

Theorem 18

There is an effective procedure to decide whether:

- A given FA accepts any words
- Two FAs are equivalent
- Two regular expressions are equivalent

Finiteness

Question: How can we tell whether an FA, or a regular expression, defines a finite language or an infinite language?

With regular expressions, the closure of any nonempty set is itself infinite. Thus, if a regular expression contains the symbol $*$, then the language it defines is infinite.

- The exception is Λ^* , which is Λ .

If the expression does not contain the symbol $*$, then the language is finite.

If we want to answer this question for an FA, we can first convert it into a regular expression. However, there are ways to determine whether an FA accepts an infinite language without having to perform the conversion.

Theorem 19

Let F be an FA with N states. Then,

- If F accepts an input string w such that $N \leq \text{length}(w) < 2N$ then F accepts an infinite language.

- If F accepts infinitely many words, then F accepts some word w such that $N \leq \text{length}(w) < 2N$

Theorem 20

There is an effective procedure to decide whether a given FA accepts a finite or an infinite language.

We test all words with lengths in the range $[N, 2N]$ on the machine.

If any are accepted, the language is infinite. Otherwise, the language is finite.

Context-Free Grammar

A context-free grammar (CFG) is a collection of three things:

- An alphabet Σ of letters called terminals from which we are going to make strings that will be the words of a language.
- A set of symbols called nonterminals, one of which is the symbol S , standing for "start here".
- A finite set of productions of the form:

One nonterminal \rightarrow finite string of terminals and/or nonterminals where the strings of terminals and nonterminals can consist of only terminals, or of only nonterminals, or of any mixture of terminals and nonterminals, or even the empty string. We require that at least one production has the nonterminal S as its left side.

The language generated by a CFG is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions. Context-free language

The Disjunction Symbol |

We use this symbol to combine all the productions that have the same left side.

Trees

We can use a tree diagram to show the derivation process of words.

Start with symbol S. Every time we use a production to replace a nonterminal by a string, we draw downward lines from the nonterminal to each character in the string.

Also known as syntax trees, parse trees, generation trees, production trees, or derivation trees.