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 $\emptyset$
- $\Lambda \notin \emptyset$
- $L + \{\Lambda\} = L$ if $\Lambda \notin L$
- $L + \emptyset = L$

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 length(w)=0, then $w = \Lambda$

Reverse

- If a is a word in some language L, then 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 $\Sigma = \{a, b\}$

PALINDROME = {Λ, and all strings x s.t. 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^*

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 $S^* = S^{**}$

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^*

$x^0 = \Lambda$ or x or x^2 or $x^3 = x^n$ 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 = {all combinations of a string from S concatenated with a string from T 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 Automation

A finite automation 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 combinations 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 labels on the loop edge and 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 FA_1 that accepts the language defined by the regular expression r_1, and there is an FA called FA_2 that accepts the language defined by the regular expression r_2, then there is an FA that we shall call FA_3 that accepts the language defined by the regular expression (r_1+r_2)

Algorithm to construct FA_3:

1) Start with 2 machines FA_1 with states $x_1, x_2, x_3, \ldots$, and FA_2 with states $y_1, y_2, y_3, \ldots$, we construct a new machine FA_3 with states $z_1, z_2, z_3, \ldots$, where each z_i is of the form $x_{\text{something}}$ or $y_{\text{something}}$

2) The combination state x_{start} or y_{start} is the start state of the new machine FA_3

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,

$z_{\text{new after reading letter p}} = (x_{\text{new after reading letter p on } FA_1})$ or $(y_{\text{new after reading letter p on } FA_2})$

Rule 3: If there is an FA_1 that accepts the language defined by the regular expression r_1, and there is an FA_2 that accepts the language defined by the regular expression r_2, then there is an FA_3 that accepts the language defined by the regular language $(r_1 r_2)$

Algorithm to construct FA_3

1) Create a state z for every state of FA_1 that we may go through before arriving at a final state.

2) For each final state x_{final} of FA_1, add a state $z = x_{\text{final}}$ or y_1, where y_1 is the start state of FA_2.

3) From the states added in step 2, add states

$$z = \begin{cases} x_{\text{something}} \text{ indicating we're still continuing on } FA_1 \\ \text{OR} \\ \text{a set of } y_{\text{something}} \text{ indicating we're on } FA_2 \end{cases}$$

4) Label every state z that contains a final state from FA_2 as a final state

Rule 4: If r is a regular expression and FA_1 is a finite automation that accepts exactly the language defined by r, then there is an FA, called FA_2, that will accept exactly the language defined by r^*

Algorithm to construct FA_2:

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 FA_2) corresponds to some collection of x-states (of FA_1). We must remember each time we reach a final state, it is possible that we have to start over again at x_1.

3) The transmissions from one collection of x-states to another based on reading certain input letters is determined by the transition rules for FA_1

4) There are only finitely many possible collections of x-states, so the machine FA_2 has only finitely many states.

Nondeterministic Finite Automata

A nondeterministic finite automation (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 +
- The $\varnothing$ 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.

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 $\varnothing$ in the FA for the a-edge (or the b-edge) to reach in this case.

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 $\{\Lambda\}$

Proof:

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

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 FA_1 and FA_2, we shall present an algorithm for constructing a union machine $FA_1 + FA_2$

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 FA_1 and FA_2 are connected to. Do not erase the start states of FA_1 and FA_2, but erase their minus signs, leaving all their edges intact. The new machine is an NFA that clearly accepts exactly language (FA_1) + language (FA_2)

2) Using the algorithm of Theorem 7, convert the NFA into an FA