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^*

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

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^*

$X^* = \Lambda$ 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:

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

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:

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 $x_{something}$ or $y_{something}$

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_{new} after reading letter p = (x_{new} after reading letter p on FA_1) or (y_{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 $z = x_{final}$ or y_1 where y_1 is the start state of FA_2.

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

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

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 +

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.
- 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 odd 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 $\{\Lambda\}$

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 F.A. 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

Moore Machine

Collection of five things:

- Finite set of states $q_0, q_1, q_2, \ldots$, where q_0 is designated as start state
- Alphabet of letters for forming the input string $\Sigma = \{a,b,c,\ldots\}$
- Alphabet of possible output characters $\Gamma = \{0,1,2,\ldots\}$
- 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 Γ is printed by each state as it is entered

Mealy Machine

Collection of four things:

- A finite set of states $q_0, q_1, q_2, \ldots$, where q_0 is designated as the start state.
- An alphabet of letters for forming the input string $\Sigma = \{a,b,\ldots\}$
- An alphabet of possible output characters $\Gamma = \{0,1,\ldots\}$
- 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_4, which prints a certain character, say t
- Consider all the incoming edges to q_4. 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_4. This means we shall be printing a t on the incoming edges before we enter q_4
- We leave the outgoing edges from q_4 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 $q_0, q_1, \ldots$, 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 q_4, one that prints a 0 (labeled as $q_4'/0$), and the other that prints a 1 (labeled as $q_4^2/1$). Hence,
  - The edges a/0 and b/0 will go into $q_4'/0$
  - The edge b/1 will go into $q_4^2/1$
- The arrow coming out of each of these two states must be the same as the edges coming out of q_4 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 Me may become two edges in Mo, 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 Me, we can let any of the copies be the start state in Mo, because they all give the identical directions for proceeding to other states
- Having a choice of start states means that the conversion of Me into Mo is NOT unique
- Repeating this process for each state of Me will produce an equivalent Mo. 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+L_2, L_1L_2, and L_1^* are also regular languages

- L_1+L_2 is the language of all words in either L_1 or L_2
- L_1L_2 is the product language of all words formed by concatenating a word from L_1 with a word from L_2.
- L_1^* 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

$$xy^nz \quad \text{for } n=1,2,3,\ldots$$

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 length(x)+length(y) does not exceed N, such that $w=xyz$ and all strings of the form

$$xy^nz \quad \text{for } n=1,2,3,\ldots$$

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 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,

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

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

Ambiguity

A CFG is called ambiguous if for at least one word in the language that it generates, there are two possible derivations of the word that correspond to different syntax trees. If a CFG is not ambiguous, it is called unambiguous.

Total Language Tree

For a given CFG, we define a tree with the start symbol S as its root and whose nodes are working strings of terminals and nonterminals. The descendants of each node are all the possible results of applying every applicable production to the working string, one at a time.

A string of terminals is a terminal node in the tree.

Theorem 21

Given any FA, there is a CFG that generates exactly the language accepted by the FA. In other words, all regular languages are context-free languages.

Semi-word

For a given CFG, a semi-word is a string of terminals (may be none) concatenated with exactly one nonterminal (on the right). In general, a semi-word has the form

(terminal) ... (terminal) (Nonterminal)

Theorem 22

If every production in a given CFG fits one of the two forms:

Nonterminal → semi-word

or

Nonterminal → word (may be Λ)

then the language generated by this CFG is regular.

Regular Grammar

A CFG is called a regular grammar if each of its production is in one of the two forms

Nonterminal → semi-word

or

Nonterminal → word

where the word may be Λ.

Λ-Productions

A Λ-production is a production of the form N → Λ. They can make our lives very difficult.

Theorem 23

If L is a context-free language generated by a CFG that includes Λ-productions, then there is a different CFG that has no Λ-productions that generates either the whole language L (if L does not include Λ), or else generates the language of all words in L that are not Λ.

Killing Λ-Productions

Definition: A nonterminal N is nullable if

- There is a production N → Λ, or
- There is a derivation that starts at N and leads to Λ: N ⇒ ... ⇒ Λ.

Given a certain CFG with Λ-productions, we convert this CFG into a new CFG without Λ-productions using the following algorithm:

- Delete all Λ-productions.
- For every production of the form

  X → (old string with nullable nonterminals)

  add new productions of the form

  X → (new string)

  where the right side can be formed by deleting all possible subsets of nullable nonterminals, except that we do not allow X → Λ to be formed, even if all the characters in the old string are nullable.
- Instead of using a production with N and then dropping the N later by N → Λ to form a word w, we simply use the correct production with N already dropped when generating w.
- There is no need to remove N later, so there is no need for the Λ-production.
- The new productions added do not lead to the generation of any new words, because every new production X → (new N-deleted string) has the same effect as the application of two old productions X → (old string with N) and then N → Λ.
- The new CFG therefore generates exactly the same language as the old CFG with the possible exception of the word Λ.

Unit Productions

A production of the form

Nonterminal → one Nonterminal

is called a unit production.

Theorem 24

If there is a CFG for the language L that has no Λ-productions, then there is also a CFG for L with no Λ-productions and no unit productions

Killing Unit Productions

For every pair of nonterminals A and B, if the CFG has a unit production A → B, or if there is a chain

A → X₁ → X₂ → ... → B

where X₁, X₂, ... are nonterminals, create new productions as follows:

If the non-unit productions from B are

B → s₁ | s₂ | ⋯

where s₁, s₂, ... are strings, we create productions

A → s₁ | s₂ | ⋯

Do the same for all such pairs of A's and B's simultaneously. We can then eliminate all unit productions.

If in the derivation of some word w, the nonterminal A gets replaced by a single unit production A → B, or by a sequence of unit productions leading to B, and further B is replaced by the production B → s_4, we can accomplish the same thing by using the production A → s_4 directly in the first place.

If the grammar has no Λ-productions, it is not difficult to find all sequences of unit productions A → X₁ → X₂ → ... → B.

However, in a grammar with Λ-productions and nullable nonterminals X and Y, the production S → ZYX is a unit production but is not easy to see.

The hypothesis of the theorem does not allow Λ-productions, so no such difficulty is possible.

Theorem 25

If L is a language generated by some CFG, then there is another CFG that generates all the non-Λ words of L, all of whose productions are of one of the two basic forms:

Nonterminal → string of only nonterminals

or

Nonterminal → one terminal

Algorithm

In the given CFG, nonterminals are S, X_1, X_2, ..., and terminals are a and b.

Then we can add two new terminals A and B, and add productions A → a and B → b.

For every previous productions involving terminals, replace each a with A and each b with B.

Now, we have a new CFG whose productions are of the form

Nonterminal → string of only Nonterminals

together with two new productions of the form

Nonterminal → one terminal

Chomsky Normal Form

If a CFG has only productions of the form

Nonterminal → string of exactly 2 Nonterminals

or the form

Nonterminal → one terminal

then it is said to be in Chomsky Normal Form (or CNF)

Theorem 26

For any context-free language L, the non-Λ words of L can be generated by a grammar in which all productions are in CNF.

CNF Algorithm

We know there is a CFG for L (or L-{Λ}) that has no Λ-productions and no unit productions.

Suppose we also have made the CFG fit the form specified in Theorem 25.

For productions of the form

Nonterminal → one terminal

we leave them alone.

For each production of the form

Nonterminal → string of Nonterminals

we use the following expansion:

S → $X_1X_2X_3X_4$

S → X_1R_1

R_1 → X_2R_2

R_2 → X_3X_4

We convert a production with long string of nonterminals into a sequence of productions with exactly two nonterminals

Leftmost Derivations

The leftmost nonterminal in a working string is the first nonterminal that we encounter when we scan the string from left to right.

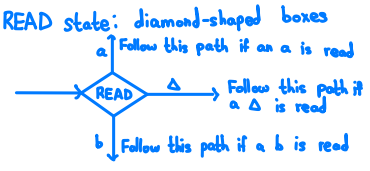
If a word w is generated by a CFG such that at each step in the derivation, a production is applied to the leftmost nonterminal in the working string, then this derivation is called a leftmost derivation.

Theorem 27

Any word that can be generated by a given CFG by some derivation also has a leftmost derivation.

Components of Pushdown Automata

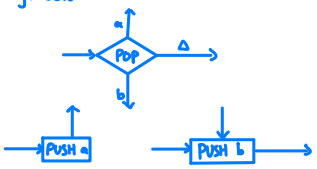
- Input tape: The part of the machine where the input string is placed. It is indefinitely long to accomodate any input. Input letters are put into cells that are named with lowercase roman numerals. We use the character Δ to denote blank cells. We read one letter at a time and never go back to a cell that was read before. We stop when we reach the first blank cell.
- START state: Begin the process here, but we read no input letters and go immediately to the next state.
- ACCEPT state: dead-end final state.
- REJECT state: dead-end state that is not final
- READ state: diamond-shaped boxes



(Δ-edge always leads to ACCEPT or REJECT)

- PUSHDOWN STACK: a place where input letters can be stored until we want to refer to them again.
  - Before the machine begins to process an input string, STACK is presumed to be empty.
  - PUSH operation adds new letter to top of the STACK. All other letters are pushed down accordingly
  - POP takes a letter out of the STACK. The rest of the letters are moved up one location each accordingly

- We can add a PUSHDOWN STACK and the operations PUSH and POP to our machine by including the following symbols:



- Edges coming out of a POP state are labeled in the same way as the edges coming out of a READ state
- Branching can occur at POP states but not at PUSH states: We can leave PUSH states only by the one indicated route, although we can enter a PUSH state from any direction

Deterministic vs Nondeterministic PDA

A deterministic PDA is one for which every input string has a unique path through the machine.

A nondeterministic PDA is one for which at certain times we may have to choose among possible paths through the machine.

- An input string is accepted by a nondeterministic PDA if some set of choices leads us to an ACCEPT state.
- If for all possible paths that a certain input string can follow, it always ends at a REJECT state, then the string must be rejected.

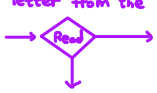
PDA Definition

A pushdown automaton, PDA, is a collection of eight things:

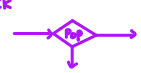
- An alphabet Σ of input letters.
- An input TAPE. Initially, the string of input letters is placed on the TAPE starting in cell i. The rest of the TAPE is blank.
- An alphabet Γ of STACK characters
- A pushdown STACK. Initially, the STACK is empty.
- One START state that has only out-edges, and no in-edges.
- Halt states of two kinds: some ACCEPT and some REJECT. They have in-edges and no out-edges.
- Finitely many non-branching PUSH states that put characters onto the top of the STACK. They are of the form

where X is any letter in Γ.

- Finitely many branching states of two kinds:
  - States that read the next unused letter from the TAPE



which may have out-edges labeled with letters from Σ and the blank character Δ, with no restrictions on duplication of labels and no requirement that there be a label for each letter of Σ, or Δ.
  - States that read the top character of the STACK



which may have out-edges labeled with the characters of Γ and the blank character Δ, again with no restrictions.

- We require that the states be connected to become a connected directed graph.
- To run a string of input letters on a PDA means that we begin from START and follow the unlabeled edges and those labeled edges, making choices of edges when necessary, to produce a path through the graph.
- This path will end either at a halt state or will crash in a branching state when there is no edge corresponding to the letter (character) being read (popped).
- When letters (characters) are read (popped) from the TAPE (STACK), they are used up and vanish.
- An input string with a path that ends in ACCEPT is said to be accepted.
- An input string that can follow a set of paths is said to be accepted if at least one of these paths leads to ACCEPT.
- The set of all input strings accepted by a PDA is called the language accepted by the PDA, or the language recognized by the PDA.

Theorem 28

For every language L, there is some PDA that accepts it.

Theorem 29

Given any PDA, there is another PDA that accepts exactly the same language with the additional property that whenever a path leads to ACCEPT, the STACK and the TAPE contain only blanks.

Theorem 30

Given a CFG that generates the language L, there is PDA that accepts exactly L.

Theorem 31

Given a PDA that accepts the language L, there exists a CFG that generates exactly L

Algorithm: CFG → PDA

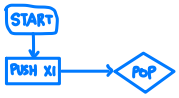
We need to first convert the CFG in CNF

Then we have the CNF is in the form

$$X_1 \to X_2X_3$$
$$X_1 \to X_3X_4$$
$$X_2 \to X_2X_2$$
$$\dots$$
$$X_3 \to a$$
$$X_4 \to a$$
$$X_5 \to b$$
$$\dots$$

with the start symbol $S=X_1$ and the other nonterminals are $X_2, X_3, \dots$

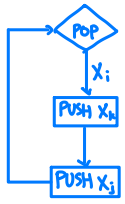
Begin with



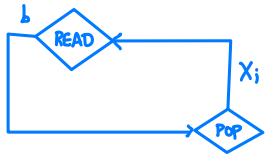
For each production of the form

$$X_i \to X_jX_k$$

We include this circuit from the POP state back to itself



For each production of the form $X_i \to b$, we include this circuit

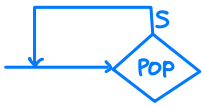


When the STACK is empty, which means that we have converted our last nonterminal to a terminal and the terminals have matched the INPUT TAPE, we add this path

We know that all words generated by the given CFG will be accepted by the PDA, and all words accepted by this PDA will have leftmost derivations in the given CFG.

At the beginning we assumed that the CFG was in CNF. But there are some CFLs that cannot be put into CNF. These are the languages that include the word Λ.

In this case, we can convert all productions into CNF and construct the PDA as described above. In addition, we must also include Λ. This can be done by adding a simple circuit at the POP:



Theorem 32

Let G be a CFG in CNF.

Let us call the productions of the form

Nonterminal → Nonterminal Nonterminal

live productions.

Let us call the productions of the form

Nonterminal → terminal

dead productions.

If we are restricted to using the live productions at most once each, we can generate only finitely many words.

Theorem 33

Let G be CNF that has p live productions and q dead productions.

If w is a word generated by G that has more than 2^p letters in it, then somewhere in every derivation tree for w, there is some nonterminal (call it Z) being used twice where the second Z is descended from the first Z.

Self-Embeddedness

In a given derivation of a word in a given CFG, a nonterminal is said to be self-embedded if it ever occurs as a tree descendant of itself.

Notation $\Rightarrow^*$

We use the symbol $\Rightarrow^*$ to stand for "can eventually replace".

Suppose in a certain CFG, the working string S_1 can produce the working string S_2, which in turn can produce the working string S_3, ..., which in turn can produce the working string S_n:

$$S_1 \Rightarrow S_2 \Rightarrow S_3 \Rightarrow \dots \Rightarrow S_n$$

Then, we can write

$$S_1 \Rightarrow^* S_n$$

Pumping Lemma for CFLs (Theorem 34)

If G is any CFG in CNF with p live productions, and w is any word generated by G with length greater than 2^p, then we can break up w into five substrings:

$$w = uvxyz$$

such that x is not Λ, and v and y are not both Λ, and such that all the words

$$uxyz,\ uvvxyyz,\ uvvvxyyyz,\ \dots$$

or in general,

$$uv^nxy^nz \quad \text{for } n=1,2,3,\dots$$

can also be generated by G.

Self-Embeddedness (Second Definition)

In a particular CFG, a nonterminal N is called self-embedded in the derivation of a word w if there are strings of terminals v and y not both null, such that

$$N \Rightarrow^* vNy$$

Theorem 35

Let L be a CFL generated by a CFG in CNF with p live productions.

Then, a word w in L with length $> 2^p$ can be broken into 5 parts:

$$w = uvxyz$$

such that

$$\text{length}(vxy) \le 2^p$$

$$\text{length}(x) > 0$$

$$\text{length}(v) + \text{length}(y) > 0$$

and such that all the words

$$uv^nxy^nz \quad \text{for } n=1,2,3,\dots$$

are in the language L.

Theorem 36

Let L_1 and L_2 be context-free languages.

Then $L_1 + L_2$ is also a context-free language

In other words, the context-free languages are closed under union.

Algorithm to find CFG for unions

Suppose L_1 and L_2 are context-free

Let the CFG for L_1 have start symbol S_1 and nonterminals $A_1, B_1, C_1, \dots$

Let the CFG for L_2 have start symbol S_2 and nonterminals $A_2, B_2, C_2, \dots$

To build a CFG for $L_1 + L_2$, the productions are the productions of the CFG for L_1 and the productions of the CFG for L_2.

We then add a new start symbol and the additional production

$$S \to S_1 | S_2$$

Theorem 37

Let L_1 and L_2 be context-free languages. Then L_1L_2 is also a context-free language.

In other words, the context-free languages are closed under product

Algorithm to find CFG for products

Let CFG_1 be the context-free grammar for L_1, having start symbol S, and nonterminals $A_1, B_1, C_1, \ldots$

Let CFG_2 be the context-free grammar for L_2, having start symbol S, and nonterminals $A_2, B_2, C_2, \ldots$

We can build a CFG for L_1L_2 by using all the productions and nonterminals of CFG_1 and CFG_2 and adding the production

$$S \longrightarrow S_1S_2$$

Theorem 38

If L is a context-free language, then L^* is also a context-free language.

In other words, the context-free languages are closed under the kleene star

Algorithm to find CFG for kleene star

Start with CFG for language L.

Change start symbol for L to S, throughout the grammar.

Keep the symbols of the nonterminals for L unchanged.

Build a new CFG for L^* by

- Using all the nonterminals and productions of the CFG for L.
- Adding a new start symbol S, and the additional production

$$S \rightarrow S_1S \mid \Lambda$$

Theorem 39

The intersection of two context-free languages may or may not be context-free

Theorem 40

The complement of a context-free language may or may not be context-free.

Union of CFL and Regular Languages

The union of a context-free language and a regular language is context free but is not always regular.

Theorem 41

The intersection of a context-free language and a regular language is always context-free.

Some Decidable Questions for CFLs

1. Given a CFG, can we tell whether or not it generates any words at all? This is the question of emptiness.
2. Given a CFG and a nonterminal X, can we tell whether or not X is ever used in the generation of words? This is the question of usefulness or uselessness.
3. Given a CFG, can we tell whether or not the language it generates is finite or infinite? This is the question of finiteness.
4. Given a CFG and a particular string of letters w, can we tell whether or not w can be generated by the CFG? This is the question of membership?

Theorem 42

Given any CFG, there is an algorithm to determine whether or not it can generate any words at all.

Algorithm to test emptiness

We can use the proof of Theorem 23 to determine if Λ is a word in the language generated by the CFG.

Now, suppose Λ is not a word generated by the CFG.

We convert the CFG into CNF. If there is a production of the form $S \rightarrow t$ where t is a terminal, then t is a word in the language.

If there are no such productions, we do the following:

Step 1:

- For each nonterminal N that has some productions of the form $N \rightarrow t$ where t is a terminal or string of terminals, we choose one of these productions and throw out all other productions for which N is on the left side.
- Replace N by t in all the productions in which N is on the right side, thus eliminating the nonterminal N altogether.
- We may have changed the grammar so that it no longer accepts the same language. It may no longer be in CNF. This is fine because every word that can be generated from the new grammar could have been generated by the old CFG. If the old CFG generated any words, then the new one does also.

Step 2:

- Repeat step 1 until either it eliminates S or it eliminates no new nonterminals
- If S has been eliminated, then the CFG produces some words; otherwise the CFG does not produce any words.

Theorem 43

There is an algorithm to decide whether or not a given nonterminal X in a given CFG is ever used in the generation of words.

Algorithm to determine usefulness of X

We will need to first see whether from X we can possibly derive a string of all terminals.

We then decide whether, starting from S, we can derive a working string involving X that will lead to a word.

We can make use of Theorem 42 to check if we can produce a string of all terminals from the nonterminal X. We can first swap X and S in all productions, then we use the algorithm to determine emptiness to see if this new grammar produces any words. If it does, then X in the original grammar can produce a string of all terminals.

We call a nonterminal that cannot ever produce a string of terminals an unproductive nonterminal

Algorithm to determine usefulness of X:

1. Find all unproductive nonterminals
2. Purify grammar by eliminating all productions involving the unproductive nonterminals
3. Paint all X's blue
4. If any nonterminal is on the left side of a production with anything blue on the right, paint that nonterminal blue, and paint all occurrences of it throughout the grammar blue as well
5. The key of this approach is that all the remaining productions are guaranteed to terminate. This means that any blue on the right gives us blue on the left. Repeat step 4 until nothing new is painted blue.
6. If S is blue, X is a useful member of CFG, because there are words with derivations that involve X-productions. If not, X is not useful.

Theorem 44

There is an algorithm to decide whether a given CFG generates a infinite language or a finite language.

Algorithm to test Finiteness

If any word in the language is long enough to apply the pumping lemma, then we can produce an infinite sequence of new words in the language.

If the language is infinite, then there must be some words long enough so that the pumping lemma applies to them.

Hence, the language of a CFG is infinite if and only if the pumping lemma can be applied.

The essence of the pumping lemma was to find a self-embedded nonterminal X.

Algorithm to determine finiteness:

1. Use the algorithm of Theorem 43 to determine which nonterminals are useless. Eliminate all productions involving them.
2. Use the following algorithm to test each of the remaining nonterminals to see whether they are self-embedded. When a self-embedded nonterminal is discovered, stop.

To test whether a nonterminal X is self-embedded we do the following:

(i) Change all X's on the left side of productions into Ψ, but leave all X's on the right side of the production alone.

(ii) Paint all X's blue.

(iii) If Y is any nonterminal that is the left side of any production with some blue on the right side, then paint all Y's blue.

(iv) Repeat step (iii) until nothing new is painted blue.

(v) If Ψ is blue, then X is self-embedded, if not, then X is not self-embedded

3. If any nonterminal left in the grammar after step 1 is self-embedded, then the language generated by the CFG is infinite. If not, then the language is finite.

Theorem 45

Given a CFG and a string x, we can decide whether or not x can be generated by the CFG.

CYK Algorithm

Suppose the nonterminals are S, N_1, N_2, ..., and the string x is composed of the following letters $x = x_1x_2x_3\cdots x_n$ where the letters are NOT different but we use different subscripts to identify their positions in the string.

In general, suppose we know that

$N_8 \Rightarrow^* x_3 \cdots x_5$ and

$N_2 \Rightarrow^* x_6 \cdots x_n$ and

$N_4 \longrightarrow N_8\,N_2$

Then we can conclude that

$N_4 \Rightarrow^* x_3 \cdots x_n$

We start by considering all substrings of length 1 of x. For each substring, we determine which nonterminals can produce them. This is easily done because all derivations come immediately from the CNF production

nonterminal $\longrightarrow$ terminal

substring	All Producing Nonterminals
x_1	N_1, N_3
x_2	N_8
...	...
x_n	N_P

Now, we look at substrings of length 2 such as x_6x_7. This can only be produced from a nonterminal N_P if the first half can be produced by N_6, and the second half by nonterminal N_7, and there is a production

$$N_P \longrightarrow N_6N_7$$

We can systematically check all the productions and our list above to determine whether the length-2 substrings can be produced:

substring	All Producing Nonterminals
x_1x_2	N...
x_2x_3	N...
...	...
$x_{n-1}x_n$	N...

Now, we move onto substrings of length 3, such as $x_3x_4x_5$.

- This substring can be derived from $N_7 \rightarrow N_9N_{10}$ where N_9 produces x_3x_4 and N_{10} produces x_5 or N_9 produces x_3 and N_{10} produces x_4x_5.

- All nonterminals producing any of these four substrings are already on our list

substring	All producing Nonterminals
$X_1 X_2 X_3$	N...
$X_2 X_3 X_4$	N...
...	...
$X_{n-2} X_{n-1} X_n$	N...

We continue this same process with substrings of length 4, 5, 6, and so on. The whole process terminates when we have all of x as the length of the substring:

substring	All Producing Nonterminals
$X_1 X_2 \dots X_n$	N...

We now examine the set of producing nonterminals:

- If S is among the producing nonterminals, then x can be generated by the CFG.
- If S is not among the producing nonterminals, then x cannot be produced by this CFG.

Turing Machines

A Turing machine, denoted by TM, is a collection of 6 things:

1. An alphabet Σ of input letters, which does not contain the blank symbol Δ.
2. A TAPE divided into cells, each containing one character or a blank Δ. The input word is presented to the machine one letter per cell, beginning in the leftmost cell, called cell i. The rest of the TAPE is filled with blanks, Δ's.
3. A TAPE HEAD that can in one step
   - read the input letter from a cell
   - replace it with some other character
   - reposition itself to the next cell to the right or to the left

   The TAPE HEAD always begins by reading the input in cell i. The TAPE HEAD can never move left from cell i. If it is given orders to do so, the machine will crash.
4. An alphabet T of character that can be printed on the TAPE by the TAPE HEAD. This can include Σ. Even though we allow the TAPE HEAD to print a Δ, we call this erasing and do not include Δ as a character in T.
5. A finite set of states including exactly one START state, from which we begin execution and which we may re-enter during execution, and some (maybe none) HALT states that cause execution to terminate when we enter them. The outer states have no function, only names: $q_1, q_2, q_3, \dots$, or 1, 2, 3, ...
6. A program, which is a set of rules that tells us, on the basis of the state we are in and the letter the TAPE HEAD has just read, how to change states, what to print on the TAPE, and where to move the TAPE HEAD.

We depict the program as a collection of directed edges, connecting the states. Each edge is labeled with a triplet of information (letter, letter, direction).

- The first letter (either Δ or from Σ or T) is the character the TAPE HEAD reads from the cell to which it is pointing.
- The second letter (either Δ or from T) is what the TAPE HEAD prints in the cell before it leaves.
- The direction tells the TAPE HEAD whether to move one cell to the right (R), or one cell to the left (L).
- There is NO requirement that every state has an edge leading from it for every possible letter on the TAPE.
- If we are in a state and read a letter that offers no choice of path to another state, we crash. A crash also occurs when we are in the first cell and try to move the TAPE HEAD left.
- To terminate execution of a certain input successfully, we must be led to a HALT state. The word on the TAPE is said to be accepted by the TM.
- By definition, all TMs are deterministic. This means that there is no state q that has two or more outgoing edges labeled with the same first letter.

Theorem 46

Every regular language has a TM that accepts it.

Algorithm (FA to TM)

- Consider any regular language L. Take an FA that accepts L.
- Change the edge labels a and b to (a,a,R) and (b,b,R), respectively.
- Change the − state to START.
- Erase the + sign out of each final state, and instead add to each of these final states an edge labeled (Δ,Δ,R) leading to a HALT state. Done.
- We read the input string, moving from state to state in the TM exactly as we would do on the FA.
  - When we come to the end of the input string, if we are not in a TM state corresponding to an FA final state, we crash when the TAPE HEAD reads the Δ in the next cell.
  - If the TM state corresponds to an FA final state, we take the edge (Δ,Δ,R) to HALT. Hence, the acceptable strings are the same for the TM and the FA.

Theorem 57

NTM = TM

- NTM stands for non-deterministic Turing machines, TM stands for deterministic Turing machines.
- In other words, this theorem says that if a language is accepted by an NTM, then it is also accepted by some TM.

Theorem 58

Every context-free language can be accepted by some TM.

However, there are some languages that are not accepted by any TM.

Recursively Enumerable Languages

A language L over the alphabet Σ is called recursively enumerable if there is a TM that accepts every word in L, and either rejects (crashes) or loops forever for every word in the language L', the complement of L.

Theorem 62

The union of two recursive enumerable languages is also recursively enumerable.

Theorem 63

The intersection of two recursive enumerable languages is also recursively enumerable.

Theorem 77

The product of two recursively enumerable languages is also recursively enumerable.

Theorem 78

If L is recursively enumerable, then L^* is also recursively enumerable.

Theorem 67

The complement of a recursively enumerable language might not be recursively enumerable.

Theorem 64

Not all languages are recursively enumerable.

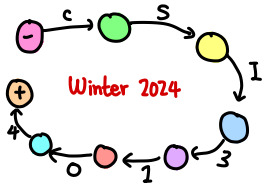
Halting Problem

Suppose we are given an input string w and a TM T. Can we tell whether or not T halts on w? This is called the halting problem for TMs.

Theorem 69

There is no TM that can accept any string w and any coded TM T and always decide correctly whether T halts on w. In other words, the halting problem cannot be decided by a TM.

THE END



You Made it

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

CSI 3104

Winter 2024

- Theory of automata
- Theory of formal languages
- Theory of turing machines

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

Good Luck on the exam

CSI 3104

Winter 2024

- Theory of automata
- Theory of formal languages
- Theory of turing machines