Formal Languages Summary

Defining Languages

Definition. An *alphabet* is a finite set of symbols.
Definition. A *language* is defined as a set of exhaustive list of words or set of rules defining accepted words over an alphabet.
Definition. A *word* or *string* is a sequence of symbols from an alphabet.
Definition. The *length* of a string is the number of letters in a string, is 0 when string is empty (Λ).
Theorem 1. For any set of strings S, we have $S^* = S^{**}$.

Recursive Definitions

A *recursive definition* is a 3 part definition:

1. First, specificy some basic objects in the set.
2. Second, give rules for constructing new objects from the basic ones.
3. Third, specify that no objects are in the set unless they are constructed in this way.

Theorems

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 contian the substring //.

Propositional Calculus. The set of all well-formed formulas (wffs) defined on the alphabet

$$\Sigma = \{\neg, \rightarrow, (,), a, b, c, d, \ldots\}$$

is defined recursively as follows:

- **Rule 1:** Any single Laten letter is a WFF (e.g. a, b, c, d, ...).
- **Rule 2:** If p is a WFF, so are (p) and $\neg p$.
- If p and q are WFFs, then so is $p \rightarrow q$.

Regular Expressions

The set of *regular expressions* is defined by the following rules:

- Rule 1: Every letter of the alphabet Σ can be made into a regular expression by writing it in **boldface**; $\mathbf{\Lambda}$ is a regular expression.
- Rule 2: If $\mathbf{r_1}$ and $\mathbf{r_2}$ are regular expressions, then so are $(\mathbf{r_1})$, $\mathbf{r_1 r_2}$, $\mathbf{r_1}+\mathbf{r_2}$, and $\mathbf{r_1}^*$.
- Rule 3: Nothing else is a regular expression.

Regular Expressions Continued

Definition. If S and T are sets of strings of letters, we define the *product set* of strings of letters as

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

Languages Associated with Regular Expressions

Definition. The following rules define the *language associated* wtih any regular expression:

- Rule 1: The language associated with the regular expression that is just a single letter is that one-letter word alone, and the language associated with $\mathbf{\Lambda}$ is just $\{\mathbf{\Lambda}\}$, a one-word language.
- Rule 2: If $\mathbf{r_1}$ is a regular expression associated with the language L_1 and $\mathbf{r_2}$ is a regular expression associated with the language L_2, then:

  (i) The regular expression $(\mathbf{r_1})(\mathbf{r_2})$ is associated with product L_1L_2, that is the language L_1 tims the language L_2.

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

  (iii) The language associated with the regular expression $(\mathbf{r_1})^*$ is L_1^*, the Kleene closure of the set L_1 as a set of words:
  $$language(\mathbf{r_1}^*) = L_1^*$$

Theorem. If L is a finite language, then L can be defined by a regular expression.

Finite Automata

A *finite automaton* is a collection of 3 things:

1. A finite set of states, one of which is designated as *start state* and some (or none) which are designated as *final states*.
2. An alphabet Σ of possible letters.
3. A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go next.

Transition Diagrams

- Directed graph with states as vertices and transitions as edges with labels being letters of the alphabet.
- Start state is indicated by a *minus sign*, final state with a *plus sign*.
- Every state has as many outgoing edges as there are letters in the alphabet, and 0 to many outgoing edges.

Transition Graphs

Definition. When an input stringg has *not* been completely read reaches a state that it cannot leave because there is no outgoing edge, we say that the input *crashes* at that state.
Definition. A transition graph (TG) is a collection of 3 things:

1. A finite set of states, *at least one of which* is designated as the *start state* and *some (or none)* of which are designated as *final states*.
2. An alphabet Σ of possible input letter from which input strings are formed.
3. A finite set of transitions that show how to go from some states to others based on reading specific substrings.

Properties of TG's

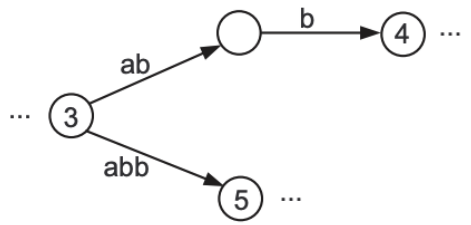
1. From part (3) of the definition, edges are labeled with some string not necessarily 1 letter.
2. There is no requirement that every state has an outgoing edge for every letter of the alphabet. There may be any number of outgoing edges.
3. A *successful path* through a TG is a series of edges forming a path from some start state to some final state.

Generalized Transition Graphs

A *generalized transition graph* (GTG) is a collection of 3 things:

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

Note: Path through a TG is not necessarily decided by the machine, human choices may be made. The machine does not make all its own determination, so we say TG's are *nondeterministic*.



Kleene's Theorem

Theorem 6

Theorem. Any language that can be defined by regular expression, or finite automaton, or transition graph can be defined by all three methods. Proving Kleene's Theorem is done in 3 parts.

- **Part 1:** Every language that can be defined by a finite automaton can also be defined by a transition graph.
- **Part 2:** Every language that can be defined by a transition graph can also be defined by a regular expression.
- **Part 3:** Every language that can be defined by a regular expression can also be defined by a finite automaton.

Proof for Part 3

Rules 1-4 show that every regular expression can be converted to a finite automaton.

- **Rule 1:** There is an FA that accepts any letter of the alphabet (including Λ).
- **Rule 2:** Given FA_1 that accepts the language defined by the regular expression $\mathbf{r}_1$ and FA_2 that accepts the language defined by $\mathbf{r}_2$, then there is an FA that accepts the language defined by the regular expression $(\mathbf{r}_1 + \mathbf{r}_2)$.
- **Rule 3:** Given FA_1 that accepts the language defined by the regular expression $\mathbf{r}_1$ and FA_2 that accepts the language defined by $\mathbf{r}_2$, then there is an FA_3 that accepts the language defined by the regular expression $(\mathbf{r}_1\mathbf{r}_2)$.
- **Rule 4:** If $\mathbf{r}$ is a regular expression and FA_1 is a finite automaton that accepts exactly the language defined by $\mathbf{r}$, then there is an FA, called FA_2, that will accept exactly the language defined by $\mathbf{r}^*$

Nondeterministic Finite Automata

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

Theorem 7

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

Proof 1 of Kleene's Theorem

Part 1: This is proved trivially since a finite automaton is a transition graph.

Part 2: Use the following algortithm:

1. Create a unqiue, unenterable start state and unique, unleaveable final state.
2. Bypass and eliminate all states that are not start or final states in any order by connecting each incoming edge with each outgoing edge, label the new edge with the concatenation of the labels.
3. When two states are joined by more than one edge, replace them with a single edge labeled with the union of the labels.
4. When all that is left is the start and end state, the label of the edge between them is the regular expression that defines the language.

Proof of Part 3

- **Rule 1:** An FA can easily be created that transitions to the final state on the single letter, then to a garbage state on any other letter.
- **Rule 2:** Algorithm: Start with 2 FA's, FA_1 with states $x_1, x_2, \ldots$, FA_2 with states $y_1, y_2, \ldots$. Start state will be $z_{start} = x_{start}$ or y_{start}. To transition to another state, next state will be z_{new} after reading letter p will be x_{new} (next state after reading p on FA_1), y_{new} (next state after reading p on FA_2). If x_i or y_1 are final states, then so is z_i.
- **Rule 3:** Algorithm: Create a state z for every state of FA_1. For every final state x_{final}, add a state $z = x_{final}$ or y_1. Then added states $z = x_i$ indicated we are continuing in FA_1 or $z = y_i$ indicated we are continuing in FA_2. Every final state in FA_2 will be a final state.
- **Rule 4:** Algorithm: Create a new start state that will also be a final state so we can accept Λ. Then, each state z_i in FA_2 will correspond to some collection of x_i states in FA_1. Set $z_1\pm = x_1-$, then the transition will be based on reading input letters determined by the transition rules in FA_1. Once we reach a final state, we have the option to return to the start state x_1.

Proof 2

Rule 1: Design an NFA to accept a, b, and λ, use Theorem 7 to construct an FA.

Rule 2: We can construct the union FA $FA_1 + FA_2$ with the following algorithm: First create a unique start state with 2 outgoing a-edges and 2 outgoing b-edges. Remove minus signs from existing start states, and use theorem 7 to convert to an FA.

Proof of Theorem 7

Proof 1

By part 2 of Kleene's theorem, we can convert an NFA into a regular expression. Then, by part 3 of Kleene's theorem, we can convert the regular expression into a finite automaton. Thus we can create an FA from an NFA.

Proof 2

The algorithm for constructing an FA from an NFA is as follows:

- Each state in the FA is a collection of states from the NFA.
- For every state z in FA, the new state that an a-edge or b-edge takes us to is the collection of possible states that result from being in x_i or x_j and taking that edge and so on.
- The start state of the FA is the same as the start state in the NFA.
- A state ϕ is added since there may be no a edges or b-edges from any particular state in the NFA, so an edge to ϕ is added.
- The state ϕ will have an a, b-edge to itself.
- A state in the FA is a final state if the collection of x states that it represents contains a final state.

Mealy and Moore Machine

A **Moore machine** is a collection of 5 things

1. A finite set of states $q_0, q_1, q_2, \ldots$ where q_0 is designated as the start state.
2. An alphabet of letters for forming the input string $\Sigma = \{a, b, c, \ldots\}$
3. An alphabet of possible output characters $\Gamma = \{0, 1, 2, \ldots\}$
4. A transition table that shows for each state and each input letter what state is reached next.

A **Mealy machine** is a collection of 4 things:

1. A finite set of states $q_0, q_1, q_2, \ldots$ where q_0 is designated as the start state.
2. An alphabet of letters forming the input string $\Sigma = \{a, b, c, \ldots\}$
3. An alphabet of possible output characters $\Gamma = \{0, 1, 2, \ldots\}$
4. A pictorial representations with states represented by nodes and directed edges indicating transitions.

Mealy and Moore Machines

The pictorial representation of a Mealy machine has:

- Each edge is labeled with a compound symbol of 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.

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

Theorem 8

Theorem 8. If Mo is a Moore machine, then there is a Mealy machine Me that is equivalent to Mo.

Proof by Constructive Algorithm:

- Consider an arbitrary state from Mo, say q_i, which outputs some character t. Suppose it has incoming edges labels with $a, b, c, \ldots$. Relabel the edges with $a/t, b/t, c/t, \ldots$, so we print t on any incoming edge to q_i. Repeat this process for all states in Mo and we end up with a Mealy machine Me.

Theorem 9

Theorem 9. For every Mealy machine Me, there is a Moore machine Mo that is equivalent to it.

Proof by Constructive Algorithm:

- Consider an arbitrary state from Me, q_i. If all incoming edges have the same printing character, then we can push the printing character into the state.
- If the incoming edges have different printing characters, then we create a new state for each printing character and connect the states with the same input letter.
- An edge that was a loop in Me may becomes 2 edges in Mo, one that is a loop and one that is not.
- If there is a state with no incoming edges, assign it any printing instruction.
- If we need to make copies of the start state in Me, any of the copies can be the start state in Mo.

Regular Languages

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

Regular Languages Continued

Theorem 10

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.

Proof 1

- If L_1 and L_2 are regular languages, then they have regular expressions $\mathbf{r}_1$ and $\mathbf{r}_2$ that define them. Then $\mathbf{r}_1 + \mathbf{r}_2$ is a regular expression that defines $L_1 + L_2$.
- Similarly, $\mathbf{r}_1\mathbf{r}_2$ is a regular expression that defines L_1L_2, and L_1^* is defined by $\mathbf{r}_1^*$.

Proof 2 Using Machines

- Since L_1 and L_2 are regular languages, there are TGs that accept them TG_1 and TG_2. Then, suppose TG_1 and TG_2 have unique start and end states. We can create a new start state that transitions to the start states of TG_1 and TG_2 on Λ. From Kleene's theorem, this TG defines $L_1 + L_2$, and has a regular expression that defines it.
- Using Kleene's theorem again, we can construct a TG that accepts L_1L_2 by having the end state of TG_1 transition to the start state of TG_2 on Λ. This TG has a regular expression that defines L_1L_2.
- Adding 2 edges from the start state to the end state and from the end state to the start state on Λ will define L_1^*.

Definition. If L is a language over the alphabet Σ, the *complement* of L' is the language of all strings over Σ that are not in L.

Theorem 11

Theorem 11. If L is a regular language, then L' is also a regular language.

Proof. If L is a regular language, then by Kleene's theorem there is an FA that accepts the language L.

- We can create a new FA that accepts the language L' by reversing each state, final states become non-final states, and vice versa. The start state becomes $\pm$.
- Strings that would end on a final state in L now end on a non-final state in L', so it rejects all strings that L accepts. Therefore this machine accepts L'.

Regular Languages: Intersections

Theorem 12

Theorem 12. If L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is also a regular language.

Proof 1

Using DeMorgan's law: $L_1 \cap L_2 = (L_1' + L_2')'$. Theorem 11 tells us L_1' and L_2' are regular, so $L_1' + L_2'$ is also regular, then the complement of that is also regular.

Proof 2

We build a machine FA_3 similarly to Kleene's theorem part 3, rule 2, which shows how to build an FA from the union of FA's. We build FA_3 the same way but only denote a state final if both corresponding states in FA_1 and FA_2 are final.

Non-regular Languages

Definition. A language that cannot be defined by a regula rexpression is called a *non-regular* language.

Theorem 13

Theorem 13. Let L be any regular language that has infinitely many words. Then there exist some three strings x, y and z (where y is not the null string) such that all the strings of the form

$$xy^n z, \ n = 1, 2, 3, \ldots$$

are words in L.

Proof. Since L is regular, there is a FA that accepts L, let w be a word in L that has more letters than there are states in the FA. Then, w must pass through some states multiple times, we can break w into parts:

1. Let x denote all the letters of w that lead to the first repeated state. This may be the null string if the circuit starts at the beginning.
2. Let y denote the substring of w that travels around the circuit back to the repeated state. This must be not the null string since there must be a circuit present in the FA.
3. Let z be the rest of w, starting after y going to the end of w.

Thus we have $w = xyz$, and we can travel the circuit as many times as we want to have $xy^n z$ be in L.

Non-regular Languages Continued

Theorem 14

Let L be an infinite language accepted by a finite automaton 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 no null and length(x)+length(y) does not exceed N, such that $w = xyz$ and all strings of the form xy^nz are in L.

Decidability

Definition. 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*. Objective is to find methods to determine if the FA $(L_1 \cap L_2') + (L_2 \cap L_1')$ accepts any words. If it accepts any words, then L_1 and L_2 are not the same.

Method 1

Convert the FA into a regular expression. Then, any regular expression excepts words by the following algorithm:

- Remove all stars from the regular expression.
- For each $+$, remove the right half and take the left.
- Remove parentheses and have just concatenations of a's and b's which forms a word.

If the FA $(L_1 \cap L_2') + (L_2 \cap L_1')$ produces a regular expression, then by method 1 it must define some words so L_1 is not the same as L_2.

Method 2

Examine the FA $(L_1 \cap L_2') + (L_2 \cap L_1')$ and see if there is a path from $-$ to $+$. If there is a path, then the machine accepts some words. Procedure for large FAs:

1. Paint the start state blue.
2. For every blue state, follow each outgoing edge and paint the destination state blue. Remove that edge.
3. Repeat step 2 until no new state is painted blue.
4. If any of the final states are blue, then the machine accepts some words. If no final states are blue, then the machine does not accept any words.

Theorem 17

Theorem 17. Let F be an FA with N states. Then, if F accepts any words, it accepts some words with fewer than N letters.

Decidability Continued

Thereom 17 Proof

Proof. If F accepts some word w, then w generates a path from $-$ to $+$. If length$(w) < N$, then we are done.

- If length$(w) \geq N$, then the path must have a circuit from $-$ to state m then back to state m and to $+$.
- Then there is a path from $-$ to state m and to $+$ without traveling through the circuit that would be shorter, and can visit each state at most once. Thus it has at most $N-1$ edges and will accept a word of at most $N-1$ letters.

Method 3

Test al wrods with fewer than N letters by running them through the FA. If the FA accepts none of them, then it accepts no words (using Theorem 17).

Theorem 18 summarizes the 3 previous results:

Theorem 18. There is an effective procedure to decide whether:

1. A given FA accepts any words.
2. Two FAs are equivalent.
3. Two regular expressions are equivalent.

Finiteness. A regular expression defines a finite language if and only if it does not contain the Kleene star.

Theorem 19

Theorem 19. Let F be a FA with N states. Then, F accepts infinitely many words if and only if it accepts some word w such that

$$N \leq \text{length}(w) \leq 2N$$

Theorem 20

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

Proof. Suppose the FA has N states over an alphabet with m letters. Then there are m^N strings of length N, and m^{N+1} strings of length $N+1$, and so on. So there are a finite number of strings of lengths between N and $2N$,

$$m^N + m^{N+1} + m^{N+2} + \cdots + m^{2N-1}$$

We can test each of these strings and if any are accepted, the language is infinite.

Context Free Grammars

Definition. A **context-free gammar** (CFG) is a collection of 3 things:

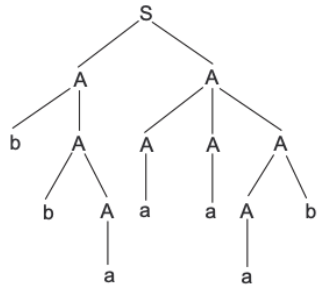
1. An alphabet Σ of letters called **terminals** from which we make strings that will be words of a language.
2. A set of symbols called **nonterminals**, one of which is the symbol S which is the start symbol.
3. A finite set of **productions** of the form

   Nonterminal $\rightarrow$ terminals and/or nonterminals

Definition. 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. A language generated by a CFG is called a *context-free language* (CFL).

Trees

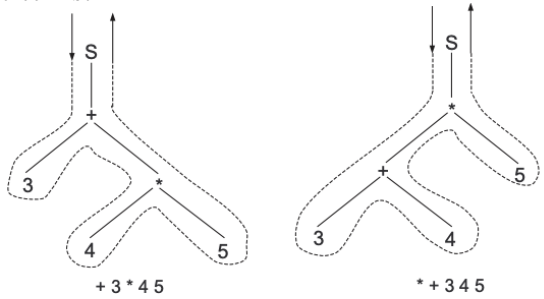
Start with the symbol S, every time we use a production to replace a nonterminal, draw downward lines from the nonterminal to each caracter in the string.



Reading the leaves from left to right gives the string generated by the CFG.

Lukasiewicz Notation

Walk around the tree and write down each symbol once that we visit, traverse down the left side of the tree first.



Then the strings are in prefix notation, which we say is *unambiguous*.

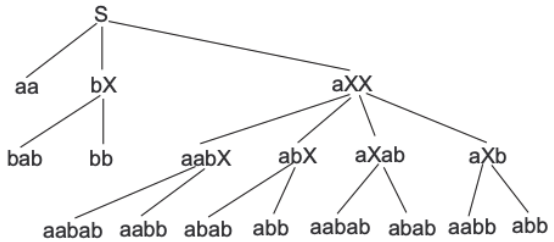
Ambiguity

Definition. A CFG is called **ambiguous** if for at least one word in the language that it generates, there are 2 possible derivation trees.

Definition. 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 possible results applying every production. This tree is called the **total language tree**.

Example: Consider the CFG:

$$S \to aa|bX|aXX$$
$$X \to ab|b$$



Grammatical Format

Definition. For a given CFG, a **semi-word** is a string of terminals (maybe none) concatenated wih exactly one nonterminal on the right.

Theorem 21

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.

Proof. Constructive Algorithm

- Step 1: Nonterminals in the CFG will be states in the FA, and S will be the start state.
- Step 2: Every edge will be a production, so $X \to aY$ would be a production if there is an edge from X to Y on a.
- For every final state F, create the production $F \to \mathbf{\Lambda}$.

Theorem 22

Theorem 22. If every production is of the form

Nonterminal $\to$ semi-word or word

Where word may be $\mathbf{\Lambda}$, then the language generated by this CFG is regular.

Grammatical Format Continued

Proof of Theorem 22

Consider a general CFG with productions $N_1 \to w_1N2$, $N_2 \to w_2N_3, \ldots, N_i \to w_i$, so wN's are semi-words. We want to construct a TG that accepts the same language so that by Kleene's theorem it is regular.

- Let $N_1 = S$, and for every production $N_x \to w_yN_z$ draw a directed edge between N_x to N_z labeled with w_y. If $N_x = N_z$, then we have a loop.
- For every production of the form $N_p \to w_q$, then we draw an edge from the state N_p to the $+$ state and label it w_q.
- Any path from $-$ to $+$ is a word in the language of the TG and is also a sequence of productions in the CFG, and on the otherhand any derivation in this CFG

$$S \Rightarrow w_iN_x \Rightarrow w_iw_jN_y \Rightarrow \cdots \Rightarrow w_iw_j \cdots w_q$$

corresponds to a path in the TG.

Therefore the language of this TG is the same as that of the CFG and the CFG is regular.

Definition. A CFG is called a **regular grammar** if each of its productions is as described in Theorem 22.

Definition. In a given CFG, we call a nonterminal N **nullable** if there is a derivation of the form $N \Rightarrow \Lambda$.

Theorem 23

Theorem 23. If L is a context-free language generated by a CFG that includes $\mathbf{\Lambda}$-productions, then there is CFG that generates L without $\mathbf{\Lambda}$

Proof. Given a CFG with Λ-productions, we can convert it using the algorithm:

1. Delete all Λ-productions.
2. For every production $N \to$ nullable string, we replace it with a new string that can be formed by deleting all possible subsets of nullable nonterminals.

Definition. A production of the form

Nonterminal $\to$ One Nonterminal

is callued a **unit production**.

Theorem 24

Theorem 27. If there is a CFG for the language L that no $\mathbf{\Lambda}$-productions, then there is also a CFG for L with no null productions or unit productions.

Grammatical Format Continued

Proof of Theorem 24

Proof. Constructive Algorithm. For every pair of nonterminals A and B, if the CFG has a unit production $A \to B$, or a chain

$$A \to X_1 \to X_2 \to \cdots \to B$$

where $X_1, \ldots,$ are nonterminals, create new productions as follows: If the non-unit productions from B are

$$B \to s_1|s_2| \cdots$$

where $s_1, s_2, \ldots$ are strings, we create the productions

$$A \to s_1|s_2| \cdots$$

Chomsky Normal Form (Theorem 25)

A method of seperating terminals from nonterminals in productions.

Theorem 25. If L is a language generated by some CFG, then there is another CFG that generates all the non-$\mathbf{\Lambda}$ words of L, all of whose products are of the form

Nonterminal $\to$ String of Nonterminals

or

Nonterminal $\to$ One Terminal

Proof. Constructive Algorithm.

- Suppose the nonterminals in a CFG are $S, X_1, X_2, \ldots$ and the terminals are a and b. Then create 2 new nonterminals A and B with the productions $A \to a$ and $B \to b$.
- Then for each of the existing productions, replace any terminals a and b with the nonterminals A and B.
- Then each production will be a string of nonterminals, along with the 2 productions which are of the form Nonterminal $\to$ One Terminal.

Definition. If a CFG has only productions of the form

Nonterminal $\to$ String of Exactly 2 Nonterminals

or of the form

Nonterminal $\to$ One Terminal

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

Theorem 26

For any context-free language L, the non-$\mathbf{\Lambda}$ words of L can be generated by a grammar in which all productions are in CNF.

Grammatical Format Continued

Proof of Theorem 26

Proof. Constructive Algorithm. From theorem 23 and 24, we know that there is a CFG for L (not including $\mathbf{\Lambda}$) that has no $\mathbf{\Lambda}$-productions or unit productions. Then we can make the CFG fit the form specified in Theorem 25.

- For the productions

$$\text{Nonterminal} \rightarrow \text{One Terminal}$$

we leave them alone. For the other productions of the form

$$\text{Nonterminal} \rightarrow \text{String of Nonterminals}$$

we introduce new nonterminals $R_1, R_2, \ldots$ so that each string of nonterminals is of length 2. For example, $S \rightarrow X_1X_2X_3X_4$ becomes

$$S \rightarrow X_1R_1, R_1 \rightarrow X_2R_2, R_2 \rightarrow X_3X_4$$

So all productions which are a string of nonterminals become a sequence of productions with exactly 2 nonterminals.

Definition. The **leftmost nonterminal** in a woroking string is the first nonterminal that we encounter when we scan the string from left to right.
Example. In $abNbaXYa$, leftmost nonterminal is N.
Definition. IF a word w is generated by a CFG such that at each step in the derivation, a production is applied to the leftmost non terminal in the working tree, then this derivation is called a **leftmost derivation**.

Theorem 27

Theorem 27. Any word that can be generated by a given CFG by some derivation also has a leftmost derivation.

Pushdown Automata

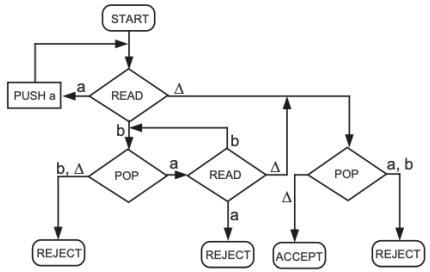
Definition. The **input tape** is part of the machine where the input is placed. It is divided into cells containing characters and is terminiated with Δ.

Symbols. In a PDA, start is denoted by a node with "START", dead-end final states are denoted "ACCEPT", and non-final dead-end states are denoted "REJECT". "READ"states are diamond-shaped boxes with branches for each letter and Δ for end of string.

Pushdown Stack: A place where input letters can be stored and retrieved. Has 2 operations, "PUSH"which adds a new letter to the top of the stack, and "POP"which takes a letter out from the top.

Pushdown Automata Continued

Branching is only allowed on POP and READ states where there can be different paths depending on the letter read.
Example. This PDA accepts the language $\{a^n b^n\}$

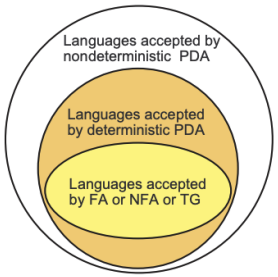


Deterministic vs Nondeterministic PDA

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 lead us to an ACCEPT state.
- If *all possible* paths end in a REJECT state, then the string is rejected.

Note that nondeterminism does increase the power of the PDA to accept new languages unlike with FAs.



Definition. A **pushdown automata** (PDA) is a collection of 8 things:

1. An alphabet Σ of input letters.
2. An inpute TAPE.
3. An alphabet Γ of STACK characters.
4. A pushdown STACK.
5. One START state with only out-edges.
6. Halt states, ACCEPT and REJECT (no out-edges).
7. Finitely many non-branching PUSH states.
8. Finitely many branching states, READ and POP.

Pushdown Automata Continued

Theorem 28

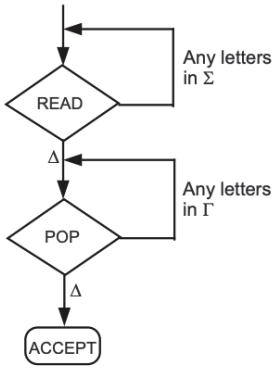
Theorem 28. For every regular language L, there is some PDA that accepts it.

Proof. Since L is regular, there is an FA that accepts it. We can convert this FA into a PDA directly by using START, READ, ACCEPT, and REJECT states.

Theorem 29

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.

Proof. Constructive Algorithm. Each time we see an ACCEPT we replace it with



CFG = PDA

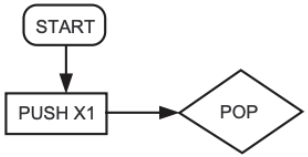
Theorem 30. Given a CFG that generates the language L, there is a PDA that accepts exactly L.

Theorem 31. Given a PDA that accepts the language L, there exists a CFG that generates exactly L.

Proof. Constructive Algorithm. Given a CFG in CNF as

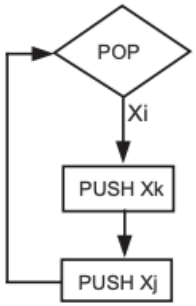
$$X_1 \rightarrow X_2X_3,\ X_1 \rightarrow X_3X_4,\ X_2 \rightarrow X_2X_2, \cdots$$
$$X_3 \rightarrow a,\ X_4 \rightarrow a,\ X_5 \rightarrow b, \ldots$$

We construct the PDA as follows, start with:

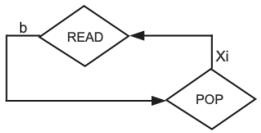


Proof of CFG = PDA Continued

For each production of the form $X_i \to X_j X_k$, we include this circuit from the POP state back to itself



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



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

- At the beginning we assumed that the CFG was in CNF, but there are CFLs that cannot be put into CNF because they include the word $\mathbf{\Lambda}$.
- In this case, convert all productions into CNF and construct the PDA as described above, then add this circuit at the POP to kill the nonterminal S without replacing it with anything.

Proof of Theorem 31 is not included, together Theorems 30 and 31 prove CFG = PDA.

Context-Free Languages

Theorem 36

Let L_1 and L_2 be context-free languages. Then their union $L_1 + L_2$ is also a context free language.

Proof. Constructive Algorithm. Let the CFG for L_1 have start symbol S_1, and L_2 has start symbol S_2.

- Define new start symbol $S \to S_1|S_2$.
- Then for if we begin with $S \Rightarrow S_1$, all words derived will belong to L_1. Similarly for S_2.

Context-Free Languages Continued

Proof 2 of Theorem 36 Using Machines

From Theorem 30 we know that there is a PDA that accepts L_1 and L_2, we can then construct a third PDA which has replaces the 2 START states with 1 new START state with 2 outgoing edges, one to the PDA corresponding to L_1 and the other to the PDA corresponding to L_2.

Theorem 37

Theorem 37. Let L_1 and L_2 be context-free languages. Then, L_1L_2 is also a context-free language.

Proof. Constructive Algorithm.

- Let CFG_1 be the context-free grammar for L_1 having start state S_1 and nonterminals $A_1, B_1, C_1, \ldots$.
- Let CFG_2 be the context-free grammar for L_2 having start state S_2 and nonterminals $A_2, B_2, C_2, \ldots$.
- Then, we use all the nonterminals and productions from CFG_1 and CFG_2, adding a new start symbol S and the production $S \to S_1S_2$.

Theorem 38

Theorem 38. If L is a context-free language, then L^* is also a context-free language.

Proof. We start with a CFG for the language L with the start state S_1. Then, using all the nonterminals and productions of the CFG for L, we add a new start symbol S and production $S \to S_1S|\mathbf{\Lambda}$. Using this gives us the derivation

$$S \Rightarrow S_1S \Rightarrow S_1S_1 \Rightarrow \cdots \Rightarrow S_1S_1S_1 \cdots S_1$$

So in general we can derive $S \Rightarrow S_1^n$, which allows us to generate any word in L^*.

Theorem 39

Theorem 39. The intersection of two context-free languages may or may not be context-free.

Proof. We know from Theorem 21 that all regular languages are context-free, when L_1 and L_2 are regular (thus context-free), in this case the intersection will be regular and thus context free.

Counter example: $L_1 = \{a^nb^na^m\}$, and $L_2 = \{a^nb^ma^m\}$ we can show these are both context-free. Then,

$$L_3 = L_1 \cap L_2 = \{a^nb^na^n\}$$

This language is not context-free.

Context-Free Languages Continued

Theorem 40.

Theorem 40. The complement of a context-free language may or may not be context-free.

Proof. We can see again that if L_1 were a regular language (and thus context-free), then it would be closed under complementation and will remain context-free. However, in general, for a contradiction:

- Suppose the complement of every CFL is context-free, let L_1 and L_2 be CFL's.
- Then $L_1' + L_2'$ is context-free by Theorem 36, and $(L_1' + L_2')'$ is context-free by assumption.
- But $(L_1' + L_2')' = L_1 \cap L_2$ which contradicts Theorem 39 since L_1 and L_2 are any CFL's.

Theorem 41

Theorem 41. The intersection of a context-free language and a regular language is always context-free.

Proof. Constructive Algorithm. We have a PDA for the CFL and an FA for the regular language. Then, we can assume the PDA terminates with a blank STACK and TAPE (using Theorem 29).

- Label each state of the PDA with the name of the particular x-state in the FA that the input string would be in if it were being processed on the FA.
- We label the START state of the PDA with x_1 which is the start state of the FA.
- If from START we go to PUSH or POP, then we stay in x_1 since we have not read any input yet.
- If we enter a READ state, we stay in FA, but when we leave the READ state by either the a or b edge, it will correspond to entering a new state in the FA.
- The PDA may be nondeterministic, so there may be several a-edges leaving a READ state, and we must label each of the states it takes us to with the x-state from the FA that an a-edge would take us.
- Special case: If in the FA, an a-edge takes us to x_3, and a b-edge takes us to x_8; but in the PDA both the a-edge and the b-edge takes us to the same state, we must make 2 copies of the state in the PDA state with identical set of exiting edges; One copy entered by the a-edge and labeled x_3, the other entered by the b-edge and labeled x_8.

Proof of Theorem 41 Continued

- If we revist a state in the PDA that is already labeled, we may need to create another copy of that state.
- If the processing of an input string ends in an ACCEPT state that is labeled with x_m where x_m is not a final state in the FA, then that input string would not be accepted.
- All ACCEPT states labaled with non-final x-states must be REJECT states.