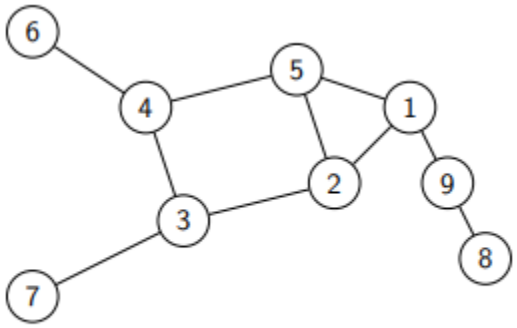
Chapter 6

6.1 : Concept and descriptions of graphs

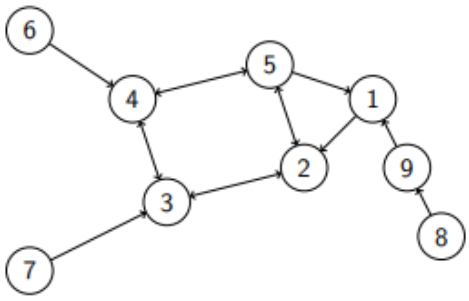
Simple graph: A graph with no loops and no parallel edges is called a simple graph. The maximum number of edges possible in a single graph with 'n' vertices is nC2(binomial) where nC2 = n(n – 1)/2. A simple railway tracks connecting different cities is an example of simple graph.

Undirected graph: An **undirected graph** data structure consists of a finite set of vertices (also called nodes), together with a set of **unordered pairs** of these vertices. These pairs are known as edges.



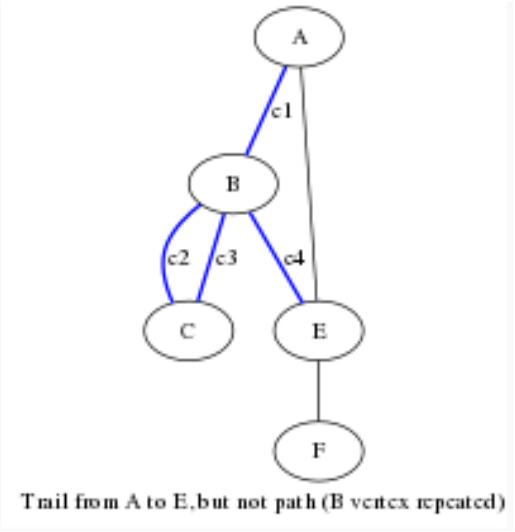
$G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and
$E = \{(1, 2), (1, 5), (1, 9), (2, 3), (2, 5), (3, 4), (3, 7), (4, 6), (8, 9)\}$.

Directed graph: A **directed graph** data structure consists of a finite set of vertices (also called nodes), together with a set of **ordered pairs** of these vertices. These pairs are known as edges, and for a directed graph are also known as arrows.



$G = (V, E)$, where
$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and
$E = \{(1, 2), (2, 3), (2, 5), (3, 2), (3, 4), (4, 3), (4, 5), (5, 1), (5, 2),$
$(5, 4), (6, 4), (7, 3), (8, 9), (9, 1)\}$.

Walk, trail, path



Trail from A to E, but not path (B vertex repeated)

- A **walk** is a finite or infinite sequence of edges which joins a sequence of vertices.

Let $G = (V, E, \phi)$ be a graph. A finite walk is a sequence of edges $(e_1, e_2, \ldots, e_{n-1})$ for which there is a sequence of vertices $(v_1, v_2, \ldots, v_n)$ such that $\phi(e_i) = \{v_i, v_{i+1}\}$ for $i = 1, 2, \ldots, n - 1$. $(v_1, v_2, \ldots, v_n)$ is the *vertex sequence* of the walk. This walk is *closed* if $v_1 = v_n$, and *open* else. An infinite walk is a sequence of edges of the same type described here, but with no first or last vertex, and a semi-infinite walk (or ray) has a first vertex but no last vertex.

- A **trail** is a walk in which all edges are distinct.
- A **path** is a trail in which all vertices (and therefore also all edges) are distinct.

Connected graph: Let G = (V, E, φ) be a graph. If for any two distinct elements u and v of V there is a path P from u to v then G is a connected graph. If |V | = 1, then G is connected.

Paired graph: New class of graphs termed Paired Threshold (PT) graphs described through vertex weights that govern the existence of edges via two inequalities. One inequality imposes the constraint that the sum of weights of adjacent vertices has to exceed a specified threshold. The second inequality ensures that adjacent vertices have a weight difference upper bounded by another threshold. We provide a conceptually simple characterization and decomposition of PT graphs, analyze their forbidden induced subgraphs and present a method for performing vertex weight assignments on PT graphs that satisfy the defining constraints

Complete graph: A **complete graph** is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.

Tree

A *tree* is an undirected graph *G* that satisfies any of the following equivalent conditions:

- *G* is connected and acyclic (contains no cycles).
- *G* is acyclic, and a simple cycle is formed if any edge is added to *G*.
- *G* is connected, but would become disconnected if any single edge is removed from *G*.
- *G* is connected and the 3-vertex complete graph K_3 is not a minor of *G*.
- Any two vertices in *G* can be connected by a unique simple path.

If *G* has finitely many vertices, say *n* of them, then the above statements are also equivalent to any of the following conditions:

- *G* is connected and has *n* − 1 edges.
- *G* is connected, and every subgraph of *G* includes at least one vertex with zero or one incident edges. (That is, *G* is connected and 1-degenerate.)
- *G* has no simple cycles and has *n* − 1 edges.

Circle graph: A Circle Graph is a graph in the form of a circle that is divided into sectors, with each sector representing a part of a set of data.

Weighted graph: A weighted graph is a graph in which each branch is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive).

6.2

GENERATIVE GRAMMER:

- The generative grammar is a universal method to define languages.

- The generative grammar (G) is the following quadruple:

  - G = (N, T, S, P) where
    • N is the set of the nonterminal symbols, also called variables, (finite nonempty alphabet),
    • T is the set of the terminal symbols or constants (finite nonempty alphabet),
    • S is the start symbol, and
    • P is the set of the production rules

- The following properties hold: $N \cap T = \emptyset$ and $S \in N$. Let us denote the union of the sets N and T by V ($V = N \cup T$). Then, the form of the production rules is $V^*NV^* \rightarrow V^*$.

- Informally, we have two disjoint alphabets, the first alphabet contains the so-called start symbol, and we also have a set of productions. The productions have two sides, both sides are words over the joint alphabet, and the word on the left-hand side must contain a nonterminal symbol.

- The word q can be derived from the word p, if q = p or there are words $r_1, r_2, \ldots, r_n$ such that $r_1 = p$, $r_n = q$ and the word r_i can be derived in one step from the word r_{i-1}, for each $2 \leq i \leq n$. (Denoted by $p \Rightarrow^* q$.)

- Let G = (N, T, S, P) be a generative grammar. The language generated by the grammar G is $L(G) = \{p \mid p \in T^*, S \Rightarrow^* p\}$.

  The above definition claims that the language generated by the grammar G contains each word over the terminal alphabet which can be derived from the start symbol S.

CHOMSKY HIERARCHY

- The Chomsky hierarchy classifies the generative grammars based on the forms of their production rules. The Chomsky hierarchy also classifies languages, based on the classes of generative grammars generating them.

- Although it is easy to find the exact position of a grammar in the Chomsky hierarchy, it is sometimes much more challenging to find the position of a language in the Chomsky hierarchy.

- Let G = (N, T, S, P) be a generative grammar.

    - Type 0 or unrestricted grammars. Each generative grammar is unrestricted.

    - Type 1 or context-sensitive grammars. G is called context-sensitive, if all of its production rules have a form

    $$p_1 A\, p_2 \rightarrow p_1 q\, p_2,$$

    or

    $$S \rightarrow \lambda,$$

    where p_1, $p_2 \in V^*$, $A \in N$ and $q \in V^+$. If $S \rightarrow \lambda \in P$ then S does not appear in the right-hand side word of any other rule.

    - Type 2 or context-free grammars. The grammar G is called context-free, if all of its productions have a form

    $$A \rightarrow p,$$

    where $A \in N$ and $p \in V^*$.
    Left hand side only one non-terminal.

    - Type 3 or regular grammars. G is called regular, if all of its productions have a form

    $$A \rightarrow r,$$

    or

    $$A \rightarrow rB,$$

    where $A, B \in N$ and $r \in T^*$.
    Right hand side at most one non-terminal in last-position. Left hand side only one non-terminal.

- A generative grammar is said to be λ-free grammar if none of its production rules contains the empty word λ on the right-hand side.
- For generative grammars generating context-sensitive languages, only one production rule form is enough to be able to generate any recursively enumerable language.
- The language L is called **recursively enumerable**, if there exists an unrestricted grammar G such that L = L(G).

- The language L is context-sensitive/ context-free/ regular if there exists a context-sensitive/ context-free/ regular grammar G such that L = L(G).

- For each context-free grammar G we can give context-free grammar G', which is context-sensitive as well, such that L(G) = L(G'). (Empty Word Theorem)

- There are context-free grammars which are not context-sensitive. Although this statement holds for grammars, we can show that in the case of languages the Chomsky hierarchy is a real hierarchy, because each context-free language is context-sensitive as well.

- A grammar G = (N, T, S, P) is regular if each of its productions has one of the following forms: $A \rightarrow a$, $A \rightarrow aB$, $S \rightarrow \lambda$, where $A, B \in N$, $a \in T$. The languages that can be generated by these grammars are the regular languages. (Alternative Definition)

- This alternative definition is usually called as regular normal form.

- The language classes defined by our original definition and by the alternative definition coincide.
- A language over an alphabet is not necessarily a finite set of words, and it is usually denoted by L.

- For a given alphabet V the language L over V is $L \subseteq V^*$.

- We have a set again, so we can use the classical set operations, if the operands are defined over the same alphabet.

- For the absolute complement operation, we use V* for universe, so L' = V*\L.

- The language L^0 contains exactly one word, the empty word. The empty language does not contain any words, $L_e = \emptyset$, and $L^0 \neq L_e$.

- We can also use the **Kleene star** and **Kleene plus** closure for languages.

FINITE AUTOMATA

Definition (Finite automata): Let A = (Q,T, q0, δ, F). It is a finite automaton (recognizer), where Q is the finite set of (inner) states, T is the input (or tape) alphabet, $q0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final (or accepting) states and δ is the transition function as follows.
• $\delta : Q \times T \rightarrow Q$ (for deterministic finite automata);
• $\delta : Q \times T \rightarrow 2^Q$ (for nondeterministic finite automata).

One can imagine a finite automaton as a machine equipped with an input tape. The machine works on a discrete time scale. At every point of time the machine is in one of its states, then it reads the next letter on the tape (the letter under the reading head), and then, according to the transition function (depending on the actual state and the letter being read,) it goes to the next state. It may happen in some variations that there is no transition defined for the actual state and letter, then the machine gets stuck and cannot continue its run. This case the automaton is called partially defined finite automaton. Otherwise the automaton is called completely defined finite automaton.

Finite automata representation – Cayley table

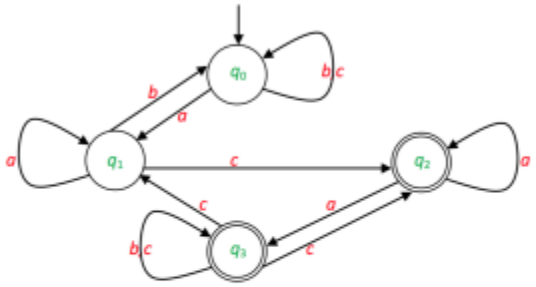
Example (Finite automata – Cayley table)

Let the finite automaton A be

T \ Q	$\to q_0$	q_1	(q_2)	(q_3)
a	q_1	q_1	q_2, q_3	–
b	q_0	q_0	–	q_3
c	q_0	q_2	–	q_1, q_2, q_3

When an automaton is given by a Cayley table, then the 0th line and the 0th column of the table are reserved for the states and for the alphabet, respectively (and it is marked in the 0th element of the 0th row).
The initial state should be the first among the states (it is advisable to mark it by a $\to$ sign also). The final states should also be marked, they should be circled.
The transition function is written into the table: the elements of the set $\delta(q, a)$ are written (if any) in the field of the column and row marked by the state q and by the letter a.

Finite automata representation – graph



Automata can also be defined in a graphical way: let the vertices (nodes, that are drawn as circles in this case) of a graph represent the states of the automaton (we may write the names of the states into the circles). The initial state is marked by an arrow going to it not from a node. The accepting states are marked by double circles. The labeled arcs (edges) of the graph represent the transitions of the automaton. If $p \in \delta(q, a)$ for some $p, q \in Q$, $a \in T$, then there is an edge from the circle representing state q to the circle representing state p and this edge is labeled by a.

Language accepted by finite automaton

Definition (Language accepted by finite automaton)

Let $A = (Q, T, q_0, \delta, F)$ be an automaton and $w \in T^*$ be an input word. We say that w is accepted by A if there is a run of the automaton, i.e., there is an alternating sequence $q_0 t_1 q_1 \dots q_{k-1} t_k q_k$ of states and transitions, that starts with the initial state q_0, ($q_i \in Q$ for every i, they are not necessarily distinct, e.g., $q_i = q_j$ is allowed even if $i \neq j$) and for every of its transition t_i of the sequence

- $t_i : q_i \in \delta(q_{i-1}, a_i)$ *in nondeterministic cases,*
- $t_i : q_i = \delta(q_{i-1}, a_i)$ *in deterministic cases,*

where $a_1 \dots a_k = w$, and $q_k \in F$. This run is called an accepting run.
All words that A accepts form $L(A)$, the language accepted (or recognized) by the automaton A.

Theorem (Determinization of finite automata):
For every finite automaton there is an equivalent (completely defined) deterministic finite automaton.

One can observe that A 0 is a completely defined deterministic automaton. Also, every run of A has an equivalent run of A' , in fact, A' simulates every possible run of A on the input at the same time. Conversely, if A' has an accepting run, then A also has at least one accepting run for the same input. Therefore, A and A' accept the same language, consequently they are equivalent.

Theorem: Every language generated by a regular grammar is accepted by a finite automaton. One can see that the successful derivations in the grammar and the accepting runs of the automaton have a one-to-one correspondence.

A language is regular, if and only if it is accepted by some finite automaton.

Finite automata – minimization

Let $A = (Q, T, q_0, \delta, F)$ be a deterministic finite automaton such that each of its states is reachable from its initial state (there are no useless states). Then we can construct the minimal deterministic finite automaton that is equivalent to A in the following way:
Let us divide the set of states into two groups obtaining the classification $C_1 = \{F, Q \setminus F\}$. (We denote the class where state q is by $C_1[q]$.)
Then, for $i > 1$ the classification C_i is obtained from C_{i-1}: the states p and q are in the same class by C_i if and only if they are in the same class by C_{i-1} and for every $a \in T$ they behave similarly: $\delta(p, a)$ and $\delta(q, a)$ are in the same class by C_i.
Set Q is finite and, therefore, there is a classification C_m such that it is the same as C_{m+1}.

Finite automata – minimization

Then, we can define the minimal completely defined deterministic automaton that is equivalent to A: its states are the groups of the classification C_m, the initial state is the group containing the initial state of the original automaton, the final states are those groups that are formed from final states of the original automaton, formally:

$$(C_m, T, C_m[q_0], \delta_{C_m}, F_{C_m}),$$

where $\delta_{C_m}(C_m[q], a) = C_m[\delta(q, a)]$ for every $C_m[q] \in C_m$, $a \in T$ and $F_{C_m} = \{C_m[q] \mid q \in F\}$.

Finite automata – minimization example

Example

Let the deterministic finite automaton A be given as follows:

T \ Q	$\to q_0$	q_1	$\overline{(q_2)}$	q_3	$\overline{(q_4)}$	$\overline{(q_5)}$	$\overline{(q_6)}$
a	q_2	q_5	q_1	q_1	q_2	q_1	q_0
b	q_1	q_0	q_3	q_4	q_5	q_3	q_2

Give a minimal deterministic finite automaton that is equivalent to A.

Example

Solution:
Before applying the algorithm we must check which states can be reached from the initial state: from q_0 one can reach the states $q_0, q_2, q_1, q_3, q_5, q_4$. Observe that the automaton cannot enter state q_6, therefore, this state (column) is deleted. The task is to minimize the following automaton by the algorithm.

T \ Q	$\to q_0$	q_1	$\overline{(q_2)}$	q_3	$\overline{(q_4)}$	$\overline{(q_5)}$
a	q_2	q_5	q_1	q_1	q_2	q_1
b	q_1	q_0	q_3	q_4	q_5	q_3

Example

We perform the first classification of the states $C_1 = \{Q_1, Q_2\}$ by separating the accepting and non-accepting states.
$Q_1 = \{q_2, q_4, q_5\}$, $Q_2 = \{q_0, q_1, q_3\}$ then we have:

Q	Q_1			Q_2		
T	$\overline{\subset q_2 \supset}$	$\overline{\subset q_4 \supset}$	$\overline{\subset q_5 \supset}$	$\to q_0$	q_1	q_3
a	Q_2	Q_1	Q_2	Q_1	Q_1	Q_2
b	Q_2	Q_1	Q_2	Q_2	Q_2	Q_1

Example

Then $C_2 = \{Q_{11}, Q_{12}, Q_{21}, Q_{22}\}$ with $Q_{11} = \{q_2, q_5\}$, $Q_{12} = \{q_4\}$, $Q_{21} = \{q_0, q_1\}$, $Q_{22} = \{q_3\}$. Then according to this classification we have:

Q	Q_{11}		Q_{12}	Q_{21}		Q_{22}
T	$\overline{\subset q_2 \supset}$	$\overline{\subset q_5 \supset}$	$\overline{\subset q_4 \supset}$	$\to q_0$	q_1	q_3
a	Q_{21}	Q_{21}	Q_{11}	Q_{11}	Q_{11}	Q_{21}
b	Q_{22}	Q_{22}	Q_{11}	Q_{21}	Q_{21}	Q_{12}

Example

Since $C_3 = C_2$ we have the solution, the minimal deterministic finite automaton equivalent to A:

T Q	$\subset Q_{11} \supset$	$\subset Q_{12} \supset$	$\to Q_{21}$	Q_{22}
a	Q_{21}	Q_{11}	Q_{11}	Q_{21}
b	Q_{22}	Q_{11}	Q_{21}	Q_{12}

Linear bounded automata – definition

Here we present a special, bounded version of the Turing machines, by which the class of context-sensitive languages can be characterized. This version of the Turing machine can work only on the part of the tape where the input is/was. These automata are called linear bounded automata (LBA).

Definition

Let $LBA = (Q, T, V, q_0, \#, \delta, F)$ be a Turing machine, where

$$\delta : Q \times (V \setminus \{\#\}) \to 2^{Q \times V \times \{Left, Right, Stay\}}$$

and

$$\delta : Q \times \{\#\} \to 2^{Q \times \{\#\} \times \{Left, Right, Stay\}}.$$

Then LBA is a (nondeterministic) linear bounded automaton.

- Finite automata can accept regular languages, so we have to extend its definition so as it could accept context-free languages. The solution for this problem is to add a stack memory to a finite automaton, and the name of this solution is “pushdown automaton”.

Definition

A pushdown automaton (PDA) is the following 7-tuple:

$$PDA = (Q, T, Z, q_0, z_0, \delta, F)$$

where

- Q *is the finite nonempty set of the states,*
- T *is the set of the input letters (finite nonempty alphabet),*
- Z *is the set of the stack symbols (finite nonempty alphabet),*
- q_0 *is the initial state,* $q_0 \in Q$,
- z_0 *is the initial stack symbol,* $z_0 \in Z$,
- δ *is the transition function having a form* $Q \times \{T \cup \{\lambda\}\} \times Z \to 2^{Q \times Z^*}$, *and*
- F *is the set of the final states,* $F \subseteq Q$.

- In order to understand the operating principle of the pushdown automaton, we have to understand the operations of finite automata and the stack memory.
- Finite automata were already introduced.
- The stack is a LIFO (last in first out) memory, which has two operations, PUSH and POP.
- When we use the POP operation, we read the top letter of the stack, and at the same time we delete it.
- When we use the PUSH operation, we add a word to the top of the stack.
- The pushdown automaton accepts words over the alphabet T. At the beginning the PDA is in state q_0, we can read the first letter of the input word, and the stack contains only z_0. In each step, we use the transition function to change the state and the stack of the PDA. The PDA accepts the input word, if and only if it can read the whole word, and it is in a final state when the end of the input word is reached.

Language accepted by pushdown automata:

Definition

$$L(PDA) = \{p \mid p \in T^*, (q_0, p, z_0) \vdash^*_{PDA} (q_f, \lambda, y), q_f \in F, y \in Z^*\}.$$

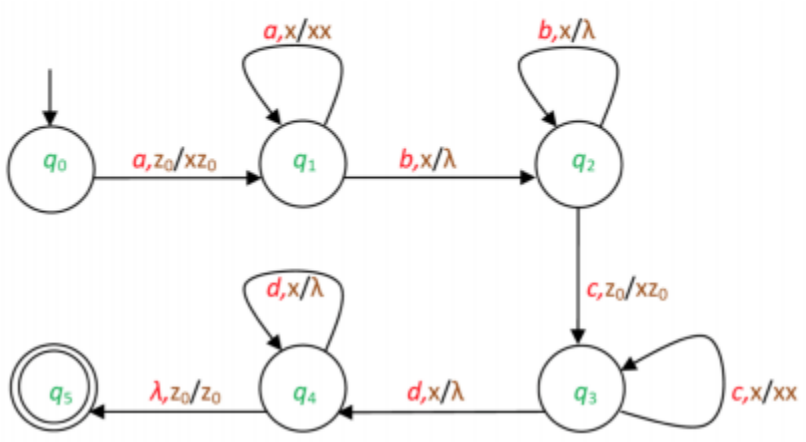
Example

This simple example shows the description of a pushdown automaton which accepts the language $L = \{a^i b^i c^j d^j \mid i, j \geq 1\}$.

$PDA = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{a, b, c, d\}, \{x, z_0\}, q_0, z_0, \delta, \{q_5\})$,
$\delta(q_0, a, z_0) = \{(q_1, xz_0)\}$,
$\delta(q_1, a, x) = \{(q_1, xx)\}$,
$\delta(q_1, b, x) = \{(q_2, \lambda)\}$,
$\delta(q_2, b, x) = \{(q_2, \lambda)\}$,
$\delta(q_2, c, z_0) = \{(q_3, xz_0)\}$,
$\delta(q_3, c, x) = \{(q_3, xx)\}$,
$\delta(q_3, d, x) = \{(q_4, \lambda\}$,
$\delta(q_4, d, x) = \{(q_4, \lambda)\}$,
$\delta(q_4, \lambda, z_0) = \{(q_5, z_0)\}$.

Pushdown automata – graph representation

The below figure shows the graphical notation of the same pushdown automaton.



- There is another method for accepting words with a pushdown automaton. It is called “acceptance by empty stack”.

- In this case, the automaton does not have any final states, and the word is accepted by the pushdown automaton if and only if it can read the whole word and the stack is empty when the end of the input word is reached. More formally:

Definition

The language accepted by automaton

$$PDA_e = (Q, T, Z, q_0, z_0, \delta)$$

by empty stack is

$$L(PDA_e) = \{p \mid p \in T^*, (q_0, p, z_0) \vdash^*_{PDA_e} (q, \lambda, \lambda), q \in Q\}.$$

- The language class accepted by pushdown automata by final states and the language class accepted by pushdown automata by empty stack are the same.

- To prove this, we use two lemmas.

Lemma

For each $PDA = (Q, T, Z, q_0, z_0, \delta, F)$ *we can give* $PDA_e = (Q', T, Z, q_0, z_0, \delta')$ *such that* $L(PDA_e) = L(PDA)$.

Proof. We are going to define a pushdown automaton PDA_e, which works the same way as the pushdown automaton PDA does, but each time when the original automaton goes into a final state, the new automaton goes into the state q_f, as well. Then, PDA_e clears out the stack, when it is in the state q_f. Formally, let $Q' = Q \cup \{q_f\}$ where $\{q_f\} \cap Q = \emptyset$, and the transition function is the following:

1. Let $(q_2, r) \in \delta'(q_1, a, z)$ if $(q_2, r) \in \delta(q_1, a, z)$, for each $q_1, q_2 \in Q, a \in T \cup \{\lambda\}, z \in Z, r \in Z^*$,
2. let $(q_f, \lambda) \in \delta'(q_1, a, z)$ if $(q_2, r) \in \delta(q_1, a, z)$, for each $q_1 \in Q, q_2 \in F, a \in T \cup \{\lambda\}, z \in Z, r \in Z^*$, and
3. let $\delta'(q_f, \lambda, z) = \{(q_f, \lambda)\}$ for each $z \in Z$.

QED.

- Second, we show the reverse case.

Lemma

For each $PDA_e = (Q, T, Z, q_0, z_0, \delta)$ *we can give* $PDA = (Q', T, Z', q_0', z_0', \delta', F)$ *such that* $L(PDA) = L(PDA_e)$.

Proof. Again, we have a constructive proof. The automaton PDA first puts the initial stack symbol of the automaton PDA_e over the new initial stack symbol. Then it simulates the original PDA_e automaton, but each time when the original automaton clears the stack completely, the new automaton goes into the new final state q_f. The automaton PDA defined below accepts the same language with final states which is accepted by the original automaton PDA_e with empty stack. Let $Q' = Q \cup \{q_0', q_f\}$, where $\{q_0'\} \cap Q = \{q_f\} \cap Q = \emptyset$, let $Z' = Z \cup \{z_0'\}$, where $\{z_0'\} \cap Z = \emptyset$, and let $F = \{q_f\}$, so $\{q_f\}$ is the only final state, q_0' is the new initial state, and z_0' is the new initial stack symbol. The transition function is the following:

1. Let $\delta'(q_0', \lambda, z_0') = \{(q_0, z_0 z_0')\}$,
2. let $(q_2, r) \in \delta'(q_1, a, z)$ if $(q_2, r) \in \delta(q_1, a, z)$, for each $q_1, q_2 \in Q, a \in T \cup \{\lambda\}, z \in Z, r \in Z^*$, and
3. let $\delta'(q, \lambda, z_0') = \{(q_f, \lambda)\}$ for each $q \in Q$.

QED

- **Theorem:** The language class accepted by pushdown automata with final states is the same as the language class accepted by pushdown automata with empty stack.

- Now we are going to prove that the languages accepted by pushdown automata are the **context-free languages**. Again, we are going to give constructive proofs.

- First, we demonstrate that for each pushdown automaton we can give a context-free grammar generating the same language as accepted by the PDA_e with empty stack, then we show how to construct a PDA_e which accepts the language generated by a context-free grammar.

Lemma

For each $PDA_e = (Q, T, Z, q_0, z_0, \delta)$ we can give a context-free grammar $G = (N, T, S, P)$ such that $L(G) = L(PDA_e)$.

The proof is difficult.

- **Theorem:** A language is context-free, if and only if it is accepted by some pushdown automaton.

Definition

The pushdown automaton $PDA_d = (Q, T, Z, q_0, z_0, \delta, F)$ is deterministic, if

1. *$\delta(q, a, z)$ has at most one element for each triple $q \in Q$, $a \in T \cup \{\lambda\}$, and $z \in Z$, and*
2. *if $\delta(q, \lambda, z)$, $q \in Q$, $z \in Z$ has an element, then $\delta(q, a, z) = \emptyset$ for each $a \in T$.*

- The language class accepted by deterministic pushdown automata (The class of deterministic context-free languages) with final states is a proper subset of the language class accepted by pushdown automata.

- We have already proven that the language class accepted by pushdown automata by final states and the language class accepted by pushdown automata by empty stack are the same.

- However, it is different for the deterministic case. The language class accepted by deterministic pushdown automata with empty stack is a proper subset of the language class accepted by deterministic pushdown automata with final states.

- Let us mark the deterministic pushdown automata accepting by empty stack with PDA_{de}. We can summarize these properties in the following expression:

    - $L(PDA_{de}) \subset L(PDA_{d}) \subset L(PDA_{e}) = L(PDA)$.

One can observe that # signs cannot be rewritten to any other symbol, therefore, the automaton can store some results of its subcomputations only in the space provided by the input, i.e., in fact, the length of the input can be used during the computation, only.
Theorem: The class of languages accepted by linear bounded automata and the class of context-sensitive languages coincide.