UNIT-4

Normal Forms for Context- Free Grammars: Eliminating useless symbols, Eliminating Є-Productions. Chomsky Normal form Griebech Normal form.
Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications
Closure Properties of Context-Free Languages: Closure properties of CFL's, Decision Properties of CFL's.
Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine
==========

Normal forms

NORMAL FORM:

- For a grammar, the RHS of a production can be any string of variables and terminals, i.e., $(V_N \cup \Sigma)^*$.
- A grammar is said to be in normal form when every production of the grammar has some specific form.
- That means, instead of allowing any member of $(V_N \cup \Sigma)$ on the RHS of the production, we permit only specific members on the RHS of the production.

There are mainly two types of normal forms:

(a) Chomsky normal form (CNF) and

(b) Greibach normal form (GNF)

CHOMSKY NORMAL FORM (CNF):

A CFG is said to be in CNF if all the productions of the grammar are in the following form.

Non-terminal → String of exactly two non-terminals

Non-terminal → Single terminal

A CFG can be converted into CNF by the following process.

A CFG is in Chomsky Normal Form if the Productions are in the following forms:

- A → a Chomsky Normal Form
- A → BC
- S → ϵ

where A, B, and C are non-terminals and **a** is terminal

Algorithm to Convert into Chomsky Normal Form:

Step 1: If the start symbol **S** occurs on some right side, create a new start symbol **S'** and a new production **S' → S**.

Step 2: Remove Null productions. (Using the Null production removal algorithm)

Step 3: Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4: Replace each production **A → B1…Bn** where **n > 2** with **A → B1 C** where

C → B2 …Bn. Repeat this step for all productions having two or more symbols in the right side.

Step 5: If the right side of any production is in the form **A→ aB** where **a** is a terminal and **A, B** are non-terminal, then the production is replaced by **A→ XB** and **X→a**

Repeat this step for every production which is in the form **A→**

Problem: Convert the following CFG into CNF

S→ ASA | aB, A → B | S, B → b |

Solution:

(1) Since **S** appears in R.H.S, we add a new state S_0 and S_0**→S** is added to the production set and it becomes:

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB$, $A \rightarrow B \mid S$, $B \rightarrow b \mid \epsilon$

(2) Now we will remove the null productions: B → ϵ and A → ϵ

After removing B → ϵ, the production set becomes:

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a$, $A \rightarrow B \mid S \mid \epsilon$, $B \rightarrow b$

After removing A → ϵ, the production set becomes:

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S$, $A \rightarrow B \mid S$, $B \rightarrow b$

(3). Now we will remove the unit productions. After removing S→ S, the production set becomes:

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $A \rightarrow B \mid S$, $B \rightarrow b$

After removing $S_0 \rightarrow S$, the production set becomes:

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow B \mid S$, $B \rightarrow b$

After removing A→ B, the production set becomes:

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow S \mid b$

$B \rightarrow b$

After removing A→ S, the production set becomes:

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$,

$S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow b \mid ASA \mid aB \mid a \mid AS \mid SA$,

$B \rightarrow b$

(4). Now we will find out more than two variables in the R.H.S

Here, $S_0 \rightarrow ASA$, $S \rightarrow ASA$, $A \rightarrow ASA$ violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF:

$S_0 \rightarrow AX \mid aB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid aB \mid a \mid AS \mid SA$

$A \rightarrow b \mid AX \mid aB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

5). We have to change the productions $S_0 \rightarrow aB$, $S \rightarrow aB$, $A \rightarrow aB$

And the final production set becomes:

$S_0 \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$A \rightarrow b \mid AX \mid YB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

Greibach Normal Form

A CFG is in Greibach Normal Form if the Productions are in the following forms:

$A \rightarrow b$

$A \rightarrow bD_1 \ldots D_n$ $S \rightarrow \epsilon$ where $A, D_1, \ldots, D_n$

are non-terminals and **b** is a terminal

Algorithm to Convert a CFG into Greibach Normal Form:

Step 1 If the start symbol **S** occurs on some right side, create a new start symbol **S'** and a new production **S' → S**.

Step 2 Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 Remove all direct and indirect left-recursion.

Step 5 Do proper substitutions of productions to convert it into the proper form of GNF.

Problem:

Convert the following CFG into CNF

S→ XY | Xn | p

X → mX | m

Y → Xn | o **Solution**:

Here, **S** does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.

Step 4:

Now after replacing

X in S → XY | Xo | p

with mX | m

we obtain

S → mXY | mY | mXo | mo | p.

And after replacing

X in Y→ Xn | o

with the right side

of

X → mX | m

we obtain

Y→ mXn | mn | o.

Two new productions O→ o and P → p are added to the production set and then

we came to the final GNF as the following:

S → mXY | mY | mXC | mC | p

X→ mX | m

Y→ mXD | mD | o

O → o

O → o

Pumping Lemma for Context-Free Languages

There are two Pumping Lemmas, which are defined for

1. Regular Languages, and
2. Context – Free Languages

Pumping Lemma for Regular Languages

For any regular language L, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists u, v, $w \in \Sigma*$, such that x = uvw, and

(1) $|uv| \leq n$

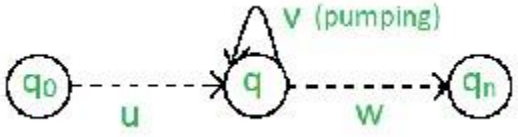
(2) $|v| \geq 1$

(3) for all $i \geq 0$: $uv^iw \in L$

In simple terms, this means that if a string v is 'pumped', i.e., if v is inserted any number of times, the resultant string still remains in L.

Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular.

The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular.



For example, let us prove $L_{01} = \{0^n1^n \mid n \geq 0\}$ is irregular.

Let us assume that L is regular, then by Pumping Lemma the above given rules follow.

Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w such that (1) – (3) hold.

We show that for all u, v, w, (1) – (3) does not hold.

If (1) and (2) hold then $x = 0^n1^n = uvw$ with $|uv| \leq n$ and $|v| \geq 1$.

So, $u = 0^a$, $v = 0^b$, $w = 0^c1^n$ where : $a + b \leq n$, $b \geq 1$, $c \geq 0$, $a + b + c =$

n But, then (3) fails for i = 0 $uv^0w = uw = 0^a0^c1^n = 0^{a+c}1^n \notin L$, since a

$+ c \neq n$.

Pumping Lemma for Context-free Languages (CFL)

Pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be 'pumped' any number of times and still be in the same language. For any language L, we break its strings into five parts and pump second and fourth substring.

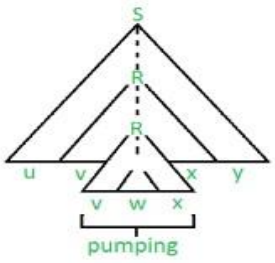
Pumping Lemma, here also, is used as a tool to prove that a language is not CFL. Because, if any one string does not satisfy its conditions, then the language is not CFL.

Thus, if L is a CFL, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists u, v, w, x, y $\in \Sigma*$, such that x = uvwxy, and

(1) $|vwx| \leq n$

(2) $|vx| \geq 1$

(3) for all $i \geq 0$: $uv^iwx^iy \in L$



For above example, 0^n1^n is CFL, as any string can be the result of pumping at two places, one for 0 and other for 1.

Let us prove, $L_{012} = \{0^n1^n2^n \mid n \geq 0\}$ is not Context-free.

Let us assume that L is Context-free, then by Pumping Lemma, the above given rules follow.

Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w, x, y such that (1) – (3) hold.

We show that for all u, v, w, x, y (1) – (3) do not hold.

If (1) and (2) hold then $x = 0^n1^n2^n = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$.

(1) tells us that vwx does not contain both 0 and 2. Thus, either vwx has no 0's, or vwx has no 2's. Thus, we have two cases to consider.

Suppose vwx has no 0's. By (2), vx contains a 1 or a 2. Thus uwy has 'n' 0's and uwy either has less than 'n' 1's or has less than 'n' 2's. But (3) tells us that $uwy = uv^0wx^0y \in L$.

So, uwy has an equal number of 0's, 1's and 2's gives us a contradiction. The case where vwx has no 2's is similar and also gives us a contradiction. Thus L is not context-free.

Source : John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2003). Introduction to Automata Theory, Languages, and Computation.

Closure Properties of Context Free Languages

Context Free languages are accepted by pushdown automata but not by finite automata. Context free languages can be generated by context free grammar which has the form :

A -> ρ (where $A \in N$ and $\rho \in (T \cup N)^*$ and N is a non-terminal and T is a terminal).

Properties of Context Free Languages

Union : If L1 and If L2 are two context free languages, their union L1 ∪ L2 will also be context free. For example,

L1 = { $a^nb^nc^m$ | m >= 0 and n >= 0 } and L2 = { $a^nb^mc^m$ | n >= 0 and m >= 0 } L3 =

L1 ∪ L2 = { $a^nb^nc^m \cup a^nb^mc^m$ | n >= 0, m >= 0 } is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their union says either of two conditions to be true. So it is also context free language.

Concatenation : If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free. For example,

L1 = { a^nb^n | n >= 0 } and L2 = { c^md^m | m >= 0 }

L3 = L1.L2 = { $a^n b^n c^m d^m$ | m >= 0 and n >= 0} is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of c's should be equal to number of d's. Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number of d's. So, we can create a PDA which will first push for a's, pop for b's, push for c's then pop for d's. So it can be accepted by pushdown automata, hence context free.

Kleene Closure : If L1 is context free, its Kleene closure L1* will also be context free. For example,

L1 = { $a^n b^n$ | n >= 0 }

L1* = { $a^n b^n$ | n >= 0 }* is also context free.

Intersection and complementation : If L1 and If L2 are two context free languages, their intersection L1 ∩ L2 need not be context free. For example,

L1 = { $a^n b^n c^m$ | n >= 0 and m >= 0 } and L2 = ($a^m b^n c^n$ | n >= 0 and m >= 0 }

L3 = L1 ∩ L2 = { $a^n b^n c^n$ | n >= 0 } need not be context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.

Similarly, complementation of context free language L1 which is ∑* – L1, need not be context free.

Deterministic Context-free Languages

Deterministic CFL are subset of CFL which can be recognized by Deterministic PDA. Deterministic PDA has only one move from a given state and input symbol. For example, L1= { $a^n b^n c^m$ | m >= 0 and n >= 0} is a DCFL because for a's, we can push on stack and for b's we can pop. It can be recognized by Deterministic PDA. On the other hand, L3 = { $a^n b^n c^m \cup a^n b^m c^m$ | n >=

0, m >= 0 } cannot be recognized by DPDA because either number of a's and b's can be equal or either number of b's and c's can be equal. So, it can only be implemented by NPDA. Thus, it is CFL but not DCFL.

Note : Out of union, concatenation, complementation, intersection and kleene-closure, DCFL are closed only under complementation.

Question : Consider the language L1,L2,L3 as given below.

$L1 = \{ a^m b^n \mid m, n >= 0 \}$

$L2 = \{ a^n b^n \mid n >= 0 \}$

$L3 = \{ a^n b^n c^n \mid n >= 0 \}$

Which of the following statements is NOT TRUE?

A. Push Down Automata (PDA) can be used to recognize L1 and L2

B. L1 is a regular language

C. All the three languages are context free

D. Turing machine can be used to recognize all the three languages

Solution : Option (A) says PDA can be used to recognize L1 and L2. L1 contains all strings with any no. of a followed by any no. of b. So, it can be accepted by PDA. L2 contains strings with n no. of a's followed by n no. of b's. It can also be accepted by PDA. So, option (A) is correct.

Option (B) says that L1 is regular. It is true as regular expression for L1 is a*b*. Option (C) says L1, L2 and L3 are context free. L3 languages contains all strings with n no. of a's followed by n no. of b's followed by n no. of c's. But it can't be accepted by PDA. So option (C) is not correct.

Option (D) is correct as Turing machine can be used to recognize all the three languages.

Question : The language $L = \{ 0^i 1 2^i \mid i \geq 0 \}$ over the alphabet {0, 1, 2} is :

A. Not recursive

B. Is recursive and deterministic CFL

C. Is regular

D. Is CFL bot not deterministic CFL.

Solution : The above language is deterministic CFL as for 0's, we can push 0 on stack and for 2's we can pop corresponding 0's. As there is no ambiguity which moves to take, it is deterministic. So, correct option is (B). As CFL is subset of recursive, it is recursive as well.

Question : Consider the following languages:

$L1 = \{ 0^n 1^n \mid n \geq 0 \}$

$L2 = \{ wcwr \mid w \varepsilon \{a,b\}^* \}$

$L3 = \{ wwr \mid w \varepsilon \{a,b\}^* \}$

Which of these languages are deterministic context-free languages?

A. None of the languages

B. Only L1

C. Only L1 and L2

D. All three languages

Solution : Languages L1 contains all strings in which n 0's are followed by n 1's. Deterministic PDA can be constructed to accept L1. For 0's we can push it on stack and for 1's, we can pop from stack. Hence, it is DCFL.

L2 contains all strings of form wcwr where w is a string of a's and b's and wr is reverse of w. For example, aabbcbbaa. To accept this language, we can construct PDA which will push all symbols on stack before c. After c, if symbol on input string matches with symbol on stack, it is popped. So, L2 can also be accepted with deterministic PDA, hence it is also DCFL.

L3 contains all strings of form wwr where w is a string of a's and b's and wr is reverse of w. But we don't know where w ends and wr starts. e.g.; aabbaa is a string corresponding to L3. For first a, we will push it on stack. Next a can be either part of w or wr where w=a. So, there can be multiple moves from a state on an input symbol. So, only non-deterministic PDA can be used to accept this type of language. Hence, it is NCFL not DCFL.

So, correct option is (C). Only, L1 and L2 are DCFL.

Question : Which one of the following grammars generate the language L = { $a^i b^j$ | $i \neq j$ }

S -> AC | CB, C -> aCb | a | b, A -> aA | ε, B -> Bb | ε

S -> aS | Sb | a | b

S -> AC | CB, C -> aCb | ε, A -> aA | ε, B -> Bb | ε

S -> AC | CB, C -> aCb | ε, A -> aA | a, B -> Bb | b

Solution : The best way to solve these type of questions is to eliminate options which do not satisfy conditions. The conditions for language L is no. of a's and no. of b's should be unequal.

In option (B), S => aS => ab. It can generate strings with equal a's and b's. So, this option is incorrect.

In option (C), S => AC => C => ε. In ε, a's and b's are equal (0), so it is not correct option.

In option (A), S will be replaced by either AC or CB. C will either generate no. of a's more than no. of b's by 1 or no. of b's more than no. of a's by 1. But one more a or one more b can be compensated by B -> bB | ε or A -> aA | ε respectively. So it may give strings with equal no. of a's and b's. So, it is not a correct option.

In option (D), S will be replaced by either AC or CB. C will always generate equal no. of a's and b's. If we replace S by AC, A with add atleast one extra a. and if we replace S by CB, B will add atleast one extra b. So this grammar will never generate equal no. of a's and b's. So, option (D) is correct.

Turing Machine

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple $(Q, X, \sum, \delta, q_0, B, F)$ where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- $\sum$ is the input alphabet
- **δ** is a transition function; $\delta : Q \times X \rightarrow Q \times X \times \{Left\_shift, Right\_shift\}$.
- $\mathbf{q_0}$ is the initial state
- **B** is the blank symbol
- **F** is the set of final states

Comparison with the previous automaton

The following table shows a comparison of how a Turing machine differs from

Finite Automaton and Pushdown Automaton.

Machine	Stack Data Structure	Deterministic?
Finite Automaton	N.A	Yes
Pushdown Automaton	Last In First Out(LIFO)	No
Turing Machine	Infinite tape	Yes

Example of Turing machine

Turing machine $M = (Q, X, \sum, \delta, q_0, B, F)$ with

- $Q = \{q_0, q_1, q_2, q_f\}$
- $X = \{a, b\}$
- $\sum = \{1\}$
- $q_0 = \{q_0\}$
- B = blank symbol
- F = $\{q_f\}$ δ is given by –Tape alphabet symbol Present State 'q0' Present State 'q1' Present State 'q2' a 1Rq1 1Lq0 1Lqf b 1Lq2 1Rq1 1Rqf

Tape alphabet symbol	Present State 'q_0'	Present State 'q_1'	Present State 'q_2'
a	$1Rq_1$	$1Lq_0$	$1Lq_f$
b	$1Lq_2$	$1Rq_1$	$1Rq_f$

Here the transition $1Rq_1$ implies that the write symbol is 1, the tape moves right, and the next state is q_1. Similarly, the transition $1Lq_2$ implies that the write symbol is 1, the tape moves left, and the next state is q_2.

Why Turing Machines?

– Robust model of computation

– Equivalence with other such models of computation, with reasonable assumptions (e.g., only finite amount of work is possible in 1 step).

– Thus, though there are several computational models, the class of algorithms (or procedures) they describe is the same.

– Can do everything a computer can do and vice versa. But takes a lot more time. Is not practical and indeed its not what is implemented in today's computer. After all who wants to write 100 lines to do subtraction or check something that a 4-year old can do?

– So then again, why Turing? Why is the top-most nobel-equivalent award in computer science called Turing award and not Bill Gates award?

Time and Space Complexity of a Turing Machine

For a Turing machine, the time complexity refers to the measure of the number of times the tape moves when the machine is initialized for some input symbols and the space complexity is the number of cells of the tape written.

Time complexity all reasonable functions −

$T(n) = O(n \log n)$

TM's space complexity −

$S(n) = O(n)$

Techniques for Turing Machine Construction

Designing a Turing machine to solve a problem is an interesting task. It is somewhat similar to programming. Given a problem, different Turing machines can be constructed to solve it. But we would like to have a Turing machine which does it in a simple and efficient manner. Like we learn some techniques of programming to deal with alternatives, loops etc, it is helpful to understand some techniques in Turing machine construction, which will help in designing simple and efficient Turing machines. It should be noted that we are using the word 'efficient' in an intuitive manner here. Introductio

1. Considering the state as a tuple. In an earlier example we considered a Turing machine which makes a copy of a given string over $\Sigma = \{a, b\}$. After reading a 'a', the machine remembers it by going to qa and after reading a 'b', it goes to qb. In general we can represent the state as [q, x] where $x \in \Sigma$ denoting that it has read a 'x'. 2. Considering the tape symbol as a tuple. Sometimes we may want to mark some symbols without destroying them or do some computation without destroying the input. In such cases it is advisable to have multiple tracks on the tape. This is equivalent to considering the tape symbol as a tuple.

There is only one tape head. In the above figure there are three tracks. The head is pointing to a cell which contains A on the first track, B on the second track and C on the third track. The tape symbol is taken a 3-tuple [A, B, C]. Some computation can be done in one track by manipulating the respective component of the tape symbol. This is very useful in checking off symbols.

2. Checking off symbols. We use one track of the tape to mark that some symbols have been read without changing them.

Extensions to Turing Machines

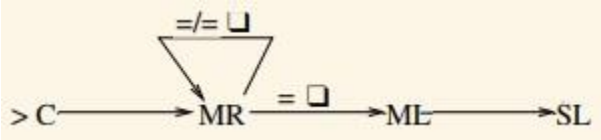
Example Turing Machines

By naming machines, we can combine complex machines into even more complex machines. • For example,

–We had a machine for copying its input. The copies are written with a space between them, and, when the machine halts, its read/write head will be over this space. Call this machine C.

– The homework exercise was a machine that shifts a string left one cell. Call this machine SL.

– Now we can construct a machine which has the effect of creating copies without a space between them (by copying and then left-shifting the rightmost copy):



– In fact, in this example, the following also works:

C −→ SL

The few examples that only hint at the power of Turing machines. But, it's natural to consider the effect of extending Turing machines: does this add any power?

• Many variations have been proposed.

E.g.: – Allow the tape to have multiple-tracks.

– Allow multiple read/write heads.

– Allow many tapes, each with its own read/write head.

– Allow a multi-dimensional tape.

– Allow non-determinism.

– Allow combinations of the above.

• They may bring greater convenience; they may speed up a computation. But none of them has increased the power. There are no problems that extended Turing machines can solve that standard Turing machines cannot solve.

• To show that a proposed extension is no more powerful than the standard model, we show anything the extension can do, the standard model can do.

I.e. we would show how a standard Turing machine would simulate the operation of the extended machine.

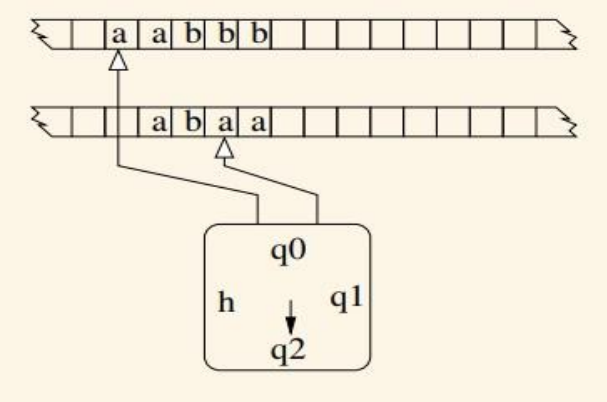
• If we wanted to show they are of equal power, we would show anything the extension can do, the standard machine can do anything the standard machine can do, the extension can do.

I.e. we would show how standard machines can simulate extended machines, and how extended machines can simulate standard machines.

• Here, we will only show the former: that the extension is no more powerful than the standard model.

- Furthermore, we won't give detailed proofs (because they're very detailed). We'll just sketch enough to get some intuitions.

Multiple Tape TM



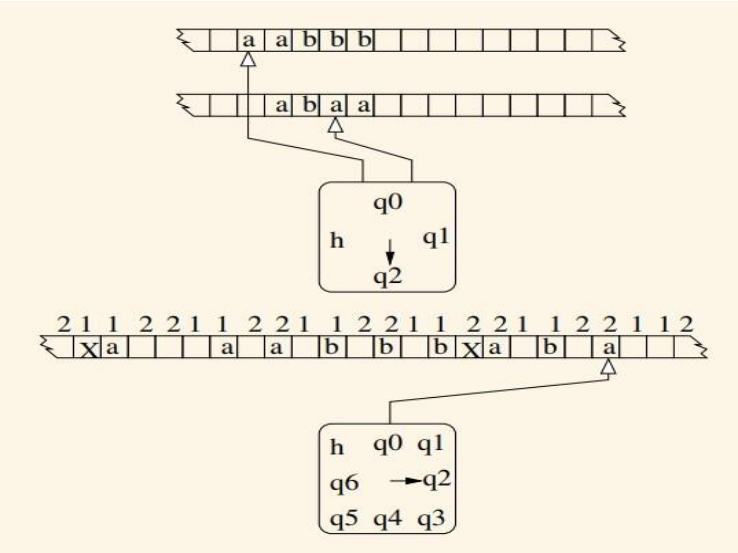
In this extension, the machine can have several tapes, each with its own read/write head. We focus on the case where there are two tapes.

- In one step, the machine reads the symbols scanned by all heads and then, depending on those symbols and its current state, each head will move or write and the control unit will enter a new state. Note that the actions of the heads are independent of each other: in one step one head might move left and another might move right or write an a.
- The transition table might now look like this:

δ	a, a	$a, \sqcup$	$\sqcup, a$	$\sqcup, \sqcup$
q_0	R, L, q_0	$\sqcup, L, q_1$	$\sqcup, \sqcup, h$	R, R, q_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

- This extension clearly brings
  - convenience — Imagine how easy it is to design a two-tape Turing machine that copies a string on one tape to the other tape; compare this with the (one-tape) Turing machine that copies its input from one part of its tape to another part of its tape.
  - efficiency — Similarly, consider the to-ing and fro-ing that can be saved by using two-tape machines.
- But it brings no extra power — as we shall now demonstrate.
- How can a Turing machine (i.e. a one-tape Turing machine) simulate a two-tape Turing machine? • We interleave the contents of the two tapes onto a single tape.

– First a symbol of tape 1, then the corresponding symbol from tape 2; – then the next symbol from tape 1 and the corresponding symbol from tape 2; – and so on.



- But we also leave an extra blank before each symbol. This can be used to record the position of the read/write heads.
- The control unit will also need a lot more states.
- The control unit of the one-tape machine will do a lot of work: whenever the two-tape machine does one step, the one-tape machine will do many steps.
- Whenever the two-tape machine does one step:
  - The one-tape machine scans left to see what symbols are next to the X's, i.e. to see what the two-tape machine would have read. Its control unit puts itself into a state that remembers this. E.g. there's one state to remember ha, ai, another to remember ⟨ ␣, a ⟩, another to remember ⟨a, ␣ ⟩, and so on. Since Σ is finite, there is a finite number of possibilities. (How many possibilities for a twotape machine? How many for a k-tape machine?) – It then scans right, back to the rightmost X.
  - The control unit will now update the X's or the symbols to their right, in accordance with what the two-tape machine would have done.

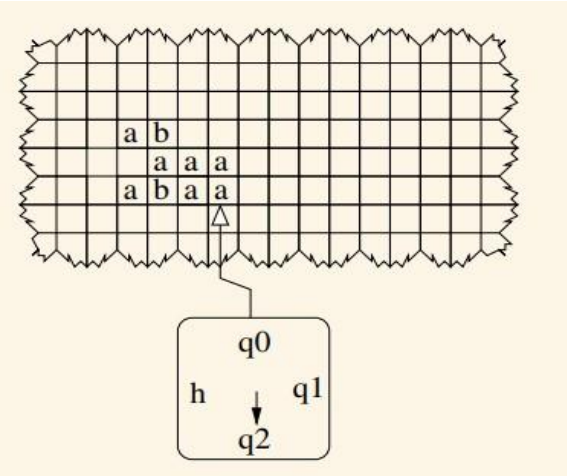
∗ E.g. if the two-tape machine would have instructed this head to write b, then go right from the X and write b.

∗ E.g. if the two-tape machine would have instructed this head to move left, then write a blank, move 4 cells left (why 4?), and write an X. It then finds the other X and repeats the above process.

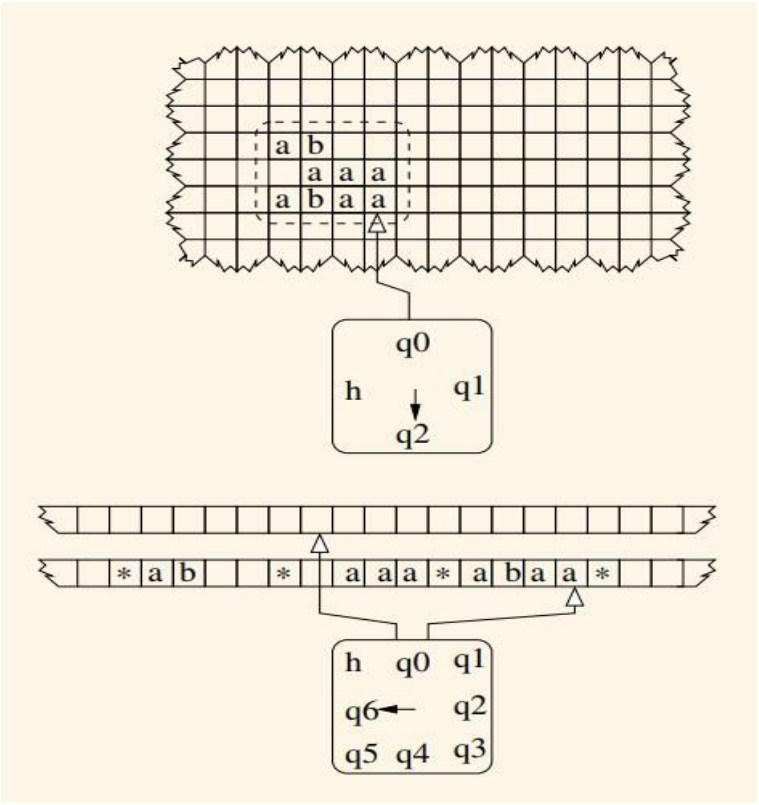
- It can be proven that this simulation correctly mimics the behaviour of the original two-tape machine.

Multidimensional Tape TM

- In this extension, the machine has a single read/write head but a multidimensional tape. We focus on the two-dimensional case:



- In one step, the machine reads the symbol scanned by the head and then, depending on the symbol and its current state the control unit will enter a new state and the head will write a new symbol or move Right, Left, Up or Down.
- This extension brings no extra power — as we shall now demonstrate.
- How can a Turing machine (i.e. a Turing machine with a one-dimensional tape or tapes) simulate a Turing machine with a two-dimensional tape?
- We simulate using a two-tape Turing machine. But we already know that two-tape machines can, in their turn, be simulated by one-tape machines.
- We can draw a rectangle around the non-blank cells. Then copy each row of the rectangle onto the second tape, separated by, e.g., *.



- Whenever the two-dimensional machine moves its head left or right, the two-tape machine moves tape 2's head left or right.
- Whenever the two-dimensional machine moves up (or down), the two-tape machine
  - scans left until it finds *;
  - as it scans, it writes symbols onto tape 1 to record how many cells it had to scan before it reached the *;
  - then it scans to the next * to the left (for up) or right (for down); – then it moves right a certain number of times, as recorded on tape 1.
  - If the result of all of this has taken it vertically out of the rectangle, then we add to tape 2 a new line of the same length to the left or right of the current lines.
  - If it has moved horizontally out of the rectangle, then we add one cell to the left or right of each line (which will require a lot of left or right shifting).
- It can be proven that this two-tape simulation correctly mimics the behaviour of the original twodimensional machine.

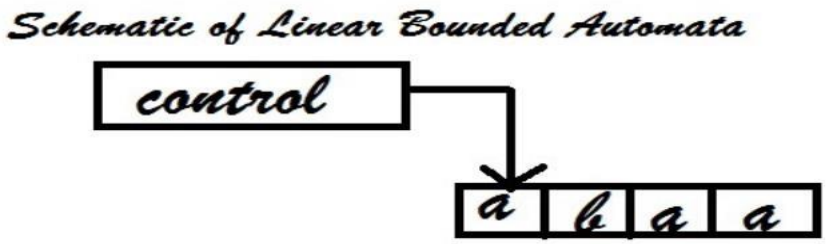
Restricted Turing Machine

1) Linear Bounded Automaton
2) Multi-stack Machines
3) Counter Machines
4) Limits on the number of states and symbols

1) Linear Bounded Automaton

is a type of turing machine wherein the tape is not permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is, in the same way that the head will not move off the left-hand end of an ordinary Turing machine's tape.

A Linear bounded automaton is a TM with a limited amount of memory. It can only solve problems requiring memory that can fit within the tape used for the input. Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor.



2) Multi-stack Machines

A deterministic two-stack machine is a deterministic TM with a read only input and two storage tapes. If a head moves left on either tape, a blank is printed on that tape.

Lemma: An arbitrary single tape TM can be simulated by a deterministic two stack machines.

Proof: the symbols to the left of the head of the TM being simulated can be stored on the stack, while the symbols on the right of the head can be placed on the other stack. On each stack, symbols closer to the TM's head are placed closer to the top of the stack than symbols farther from the TM's head.

3) Counter Machines

are offline TMs (is a multi-tape TM whose input tape is read only) whose storage tapes are semi-infinite,and whose tape alphabets contain only two symbols Z and B(Blank).

Furthermore the symbol Z,which serves as a bottom of stack marker,appears initially on the cell scanned by the tape head and may never appear on any other cell.

An integer i can be stored by moving the tape head i cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. We can test whether a number is zero by checking whether Z is scanned by the head, but we cannot directly test whether two numbers are equal.

Instantaneous description of a counter machine can be described by the state,the input tape contents,the position of the input head,and the distance of the storage heads from the symbol Z.

The counter machine can really only stores a count on each tape and tell if that count is zero.

4) Limits on the number of states & symbols

Another way to restrict a TM is to limit the size of the tape alphabet or the number of states.

If the alphabet,number of tapes,and number of states are all limited,then there is only a finite number of different TMs,so that restricted model is less powerful than the original.