ÇUKUROVA UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

Lecture 2: Chomsky Hierarchy and Automata

Motivation - Programming language syntax:

- The syntax of a programming language is the set of rules that define what valid programs are.
- A typical form for formulating these rules are so-called grammars.
- This includes in particular that analysis programs for programs (so-called parsers) can be constructed automatically.

Example #1: A grammar for arithmetic expressions

<Digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Number> → <Digit> | <Digit> <Number>

<Expression> → <Number>

<Expression> → (<Expression>)

<Expression> → <Expression> + <Expression>

<Expression> → <Expression> − <Expression>

<Expression> → <Expression> * <Expression>

<Expression> → <Expression> / <Expression>

Application example

⇒ <Expression> ∗ <Expression>

⇒ (<Expression>)∗ <Expression>

⇒ (<Expression> + <Expression>)∗ <Expression>

⇒ (<Number> + <Expression>)∗ <Expression>

⇒ (<Number> <Digit> + <Expression>)∗ <Expression>

⇒ (<Digit> <Digit> + <Expression>)∗ <Expression>

⇒ (1 <Digit> + <Expression>)∗ <Expression>

⇒ (17 + <Expression>)∗ <Expression>

⇒ (17 + <Number>)∗ <Expression>

⇒ (17+4)∗ <Expression>

⇒(17+4)∗ <Number> <Digit>

⇒(17+4)∗ <Number> <Digit> <Digit>

⇒(17+4)∗ <Digit> <Digit> <Digit>

⇒ (17+4)∗ 3 <Digit> <Digit>

⇒ (17+4)∗ 3 <Digit> <Digit>

⇒(17+4)∗37 <Digit>

⇒ (17+4)∗372

A distinction is made between the following individual symbols:

- **Metasymbols:** Characters like -> and | that are needed to formulate the rules.

- **Terminal symbols:** They correspond to the tokens of the language and are the only characters that can appear in sentences of the language. In our example these are **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), +, - and *.**

- **Nonterminal symbols:** These are the strings of characters enclosed in angle brackets. They are also sometimes called variables or syntactic categories.

- **Rules/Productions:** The grammar given above consists of a set of rules or productions.
    - Starting with a start symbol (here: <expression>), the rules can be used to create strings by replacing an occurrence of a nonterminal symbol that occurs on the left side of a rule with the right side in the string.
    - Occurs the meta symbol | on the right side on, the character rows thus separated are interpreted as alternative replacement options.

Example #2: A grammar for HTML tables

Rules:

<TAB> → "<table>" <ROWS> "</table>"

<ROWS> → <ROWS><ROWS> | "<tr>" <COLS> "</tr>"

<COLS> → <COLS><COLS> | "<td>" <TEXT> "</td>" | "<td>" <TAB> "</td>"

TEXT→ .....

Input:

<table><tr><td>Hallo</td></tr></table>

Formal Grammar Definition

A formal grammar of this type consists of a finite set of **production rules P** (left-hand side → right-hand side), where each side consists of a finite sequence of the following symbols:

- a finite set of **terminal symbols T** (indicating that no production rule can be applied)
- a finite set of **nonterminal symbols N** (indicating that some production rule can yet be applied)
- a **start symbol S** (a distinguished nonterminal symbol)

⇨ **G = (T, N, P, S)**

Chomsky Classification of Grammars

According to Noam Chomsky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other.

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Depending on the structure of the rules, grammars are different powerful. Take a look at the following illustration. It shows the scope of each type of grammar.

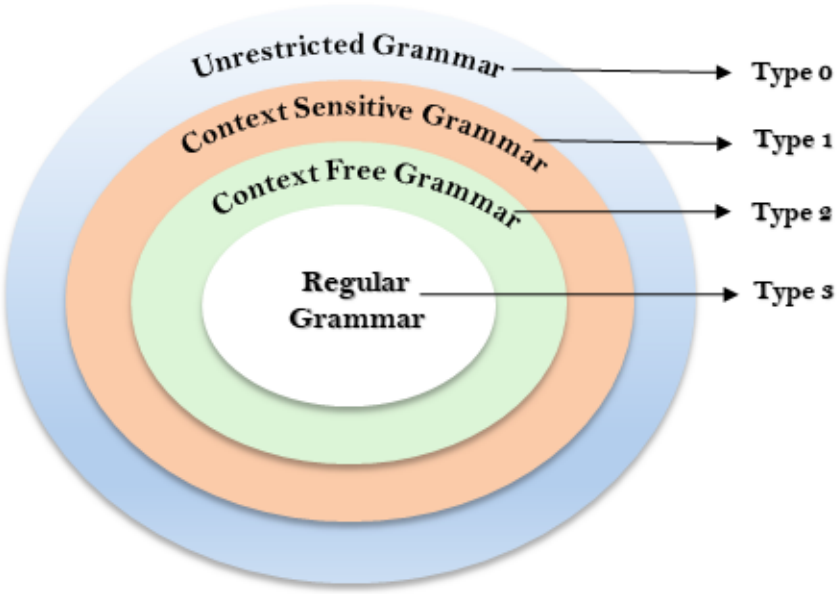


Figure 1: Chomsky Hierarchy

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

- The productions can be in the form of **α** → **β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.
- The languages generated by Type-0 grammars are recognized by a Turing machine.

Examples of rules:

S → ACaB
Bc → acB
CB → DB
aD → Db
aBC → a

Type 0 Example: $L(G) = \{a^i \mid i = 2^n,\ n \geq 1\}$ ➔ Words must be created by consecutively writing the a's by multiples of 2 ➔ aa, aaaa, aaaaaaaa, etc.

$$G = \{T, N, P, S\}$$

$$T = \{a\}$$

$$N = \{S, L, R, A, B, C\}$$

$$P: S \Rightarrow LAaR$$
$$Aa \Rightarrow aaA$$
$$AR \Rightarrow BR \mid C$$
$$aB \Rightarrow Ba$$
$$LB \Rightarrow LA$$
$$aC \Rightarrow Ca$$
$$LC \Rightarrow \varepsilon$$

$$S \Rightarrow L\boldsymbol{AaR} \Rightarrow L\boldsymbol{aa}\underline{\boldsymbol{AR}} \Rightarrow La\underline{a\boldsymbol{C}} \Rightarrow La\underline{\boldsymbol{Ca}}$$
$$\Rightarrow \underline{L\boldsymbol{C}}aa \Rightarrow aa$$

$$S \Rightarrow \underline{\boldsymbol{LAa}}\boldsymbol{R} \Rightarrow La\underline{a\boldsymbol{A}}R \Rightarrow La\underline{a\boldsymbol{B}}\boldsymbol{R}$$
$$\Rightarrow La\underline{\boldsymbol{Ba}}R \Rightarrow \underline{L\boldsymbol{Ba}}aR$$
$$\Rightarrow \underline{\boldsymbol{LA}}aaR \Rightarrow L\boldsymbol{aa}\underline{\boldsymbol{A}}aR$$
$$\Rightarrow Laa\boldsymbol{aa}\underline{\boldsymbol{AR}} \Rightarrow Laaaa\underline{\boldsymbol{C}}$$
$$\Rightarrow Laaa\underline{\boldsymbol{Ca}} \Rightarrow La\underline{a\boldsymbol{Ca}}a$$
$$\Rightarrow La\underline{\boldsymbol{Ca}}aa \Rightarrow \underline{L\boldsymbol{Ca}}aaa$$
$$\Rightarrow aaaa$$

Type-1 grammars generate context-sensitive languages.

- The productions must be in the form **α A β → α γ β**, where **A ∈ N** (Non-terminal) and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)
- The strings **α** and **β** may be empty, but **γ** must be non-empty.
- The rule **S → ε** is allowed if **S** does not appear on the right side of any rule (due to monotony property).
- The languages generated by Type-1 grammars are recognized by a linear bounded automaton.

Note:

|α A β| ≤ |α γ β| (=> Monotony property[1])

Examples of rules:

aA → ab
abCd → abcd
BA → AB
AB → AbBc
A → bcA
B → b

Typ 1 Example: $L(G) = \{a^n, b^n, c^n \mid n \geq 1\}$ => abc, aabbcc, aaabbbccc, etc.

$G = \{T, N, P, S\}$

$T = \{a, b, c\}$

$N = \{S, A, B\}$

$P: S \Rightarrow aSAB$
$S \Rightarrow aAB$
$BA \Rightarrow AB$
$aA \Rightarrow ab$
$bA \Rightarrow bb$
$bB \Rightarrow bc$
$cB \Rightarrow cc$

$S \Rightarrow \boldsymbol{a\underline{S}AB} \Rightarrow a\underline{\boldsymbol{aA}}\boldsymbol{B}AB \Rightarrow a\boldsymbol{ab}\underline{BA}B$
$\Rightarrow aab\underline{\boldsymbol{AB}}B \Rightarrow aa\boldsymbol{b}\underline{\boldsymbol{b}BB}$
$\Rightarrow aab\boldsymbol{bc}\underline{B} \Rightarrow aabb\boldsymbol{cc}$

$S \Rightarrow \boldsymbol{a\underline{S}AB} \Rightarrow a\boldsymbol{a\underline{S}AB}AB \Rightarrow aa\underline{\boldsymbol{aA}}\boldsymbol{B}ABAB$
$\Rightarrow aa\boldsymbol{ab}\underline{BA}BAB$
$\Rightarrow aaab\underline{\boldsymbol{AB}}BAB$
$\Rightarrow aaabAB\underline{\boldsymbol{AB}}B$
$\Rightarrow aaab\underline{A}\boldsymbol{B}BB$
$\Rightarrow aaa\boldsymbol{b}\underline{\boldsymbol{b}}ABBB$
$\Rightarrow aaab\boldsymbol{b}\underline{\boldsymbol{b}BB}BB$
$\Rightarrow aaabb\boldsymbol{bc}\underline{BB}$
$\Rightarrow aaabbb\boldsymbol{cc}\underline{B}$
$\Rightarrow aaabbbc\boldsymbol{cc}$

[1] The length of the rule on the left side must be smaller or equal the length of the rule on the right side. The monotnoy property logically also applies for Typ 2 and Typ 3 grammars.

Type-2 grammars generate context-free languages.

- The productions must be in the form **A → γ,** where **A ∈ N** (Non terminal) and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).
- The languages generated by Type-2 grammars are recognized by a pushdown automaton.

Example of rules:

S → Xa
X → a
X → aX
X → abc
X → ε

Typ 2 Example: L(G) for creating mathematical expressions

$$G = \{T, N, P, S\}$$

$$T = \{+, -, *, /, (,), a, b\}$$

$$N = \{S\}$$

$$P: S \Rightarrow S + S \mid S * S \mid S - S \mid S/S \mid (S) \mid a \mid b$$

$$S \Rightarrow S * \boldsymbol{S} \Rightarrow S * (\boldsymbol{S}) \Rightarrow \boldsymbol{S} * (S - S)$$
$$\Rightarrow a * (\boldsymbol{S} - S)$$
$$\Rightarrow a * (a - \boldsymbol{S})$$
$$\Rightarrow a * (a - b)$$

Type-3 grammars generate regular languages. Type-3 grammars can be left regular or right regular.

- Right regular Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.
  - The right regular Type-3 productions must be in the form **X → a or X → aY,** where **X, Y ∈ N** (Nonterminal) and **a ∈ T** (Terminal)

- Left regular Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single non-terminal followed by a single terminal.
  - The left regular Type-3 productions must be in the form **X → a or X → Ya,** where **X, Y ∈ N** (Nonterminal) and **a ∈ T** (Terminal)

- The rule **S → ε** is allowed if **S** does not appear on the right side of any rule (due to monotony property).
- The languages generated by Type-3 grammars are recognized by a finite state automaton.

Example of right regular rules:

X → ε
X → a | aY
Y → b

Example of left regular rules:

X → ε
X → a | Ya
Y → b

Typ 3 Example: L(G) = A pattern with at least two 0s before each 1. ➔ 001, 001001, 0001, 0001, 00100001, etc.

$G = \{T, N, P, S\}$

$T = \{0,1\}$

$N = \{S, A, B\}$

$P: S \Rightarrow 0S \mid 0A \mid 0 \mid \varepsilon$
$A \Rightarrow 0B$
$B \Rightarrow 1S$

$S \Rightarrow 0\boldsymbol{S} \Rightarrow 00$

$S \Rightarrow 0\boldsymbol{A} \Rightarrow 00\boldsymbol{B} \Rightarrow 001\boldsymbol{S} \Rightarrow 001$

$S \Rightarrow 0\boldsymbol{S} \Rightarrow 00\boldsymbol{A} \Rightarrow 000\boldsymbol{B} \Rightarrow 0001\boldsymbol{S} \Rightarrow 00010\boldsymbol{A} \Rightarrow 000100\boldsymbol{B} \Rightarrow 0001001\boldsymbol{S} \Rightarrow 00010010$

Chomsky Language Class	Grammar	Recognizer
3	Regular	Finite-State Automaton
2	Context-Free	Push-Down Automaton
1	Context-Sensitive	Linear-Bounded Automaton
0	Unrestricted	Turing Machine

Figure 2: Grammar types and automata

Finite State Automaton (or Finite State Machine)

Finite Automata (FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine that has five elements or tuples. It has a set of states and rules for moving from one state to another, but it depends upon the applied input symbol.

A finite automaton is a collection of **5-tuple (Σ, S, δ, s_0, F)**, where:

1. **Σ** is a finite set of the input alphabet
2. **S** is a finite set of states
3. **δ: S x Σ → S** is the transition function
4. **s_0** is the initial state **∈ S**
5. **F** is a set of final states (F ⊂ S)

Example: Representation of an acceptor; this example investigates whether a binary number has an even number of 0s, where s_1 is an *accepting state* and s_2 is a *non accepting state*. ➔ 11, 001, 0110, 001100, etc.

FA = $(\Sigma, S, \delta, s_0, F)$ with

- $\Sigma = \{1, 0\}$
- $S = \{s_1, s_2\}$
- $s_0 = \{s_1\}$
- $F = \{s_1\}$

δ is given by

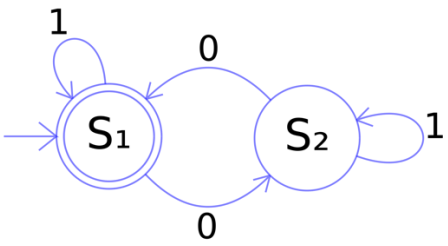


Figure 2: Transition in a FA

Pushdown Automaton

Basically, a pushdown automaton is **"Finite state machine" + "a stack".** A PDA has three components:

- an input tape,
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack. A stack does two operations −

- **Push** − a new symbol is added at the top.
- **Pop** − the top symbol is read and removed.

Note:

- A Finite State Machine can remember **a finite amount of information**, but a PDA can remember an **infinite amount of information**.
- The addition of stack is used to provide a **last-in-first-out (LIFO)** memory management capability to Pushdown automata.
- PDA can store an unbounded amount of information on the stack. But PDA can access a limited amount of information on the stack (push and pop).
- In PDA, the stack is used to store the items temporarily.
- A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

A PDA is a **7-tuple (Σ, Γ, S, δ, s_0, Z_0, F)**, where

1. **Σ** is a set of finite **input symbols**,
2. **Γ** is a set of finite **stack symbols**,
3. **S** is a set of **states**,
4. $\delta : S \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{S \times \Gamma^*}$ is a function such that $\delta(s, a, Z)$ is finite for any $s \in S$, $a \in \Sigma \cup \{\varepsilon\}$, and $Z \in \Gamma$ (the **transition function**),
5. s_0 is the **initial state** $\in S$
6. Z_0 is the **initial stack symbol** $\in \Gamma$ and
7. **F** is a set of **accepting states** ($F \subset S$)

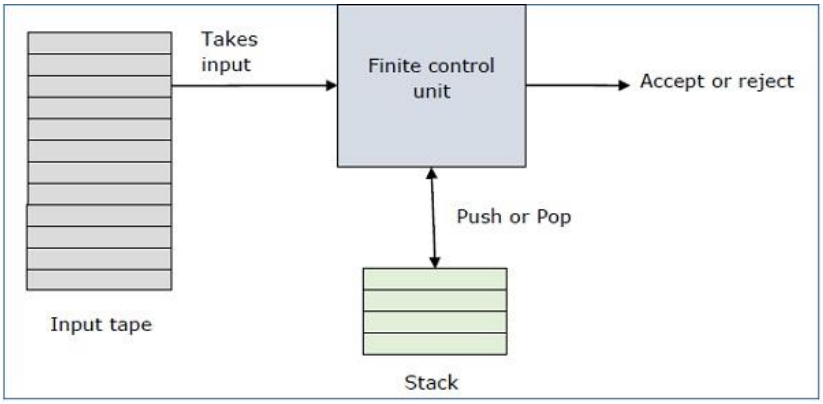


Figure 3: Design of PDA

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as a,b → c .

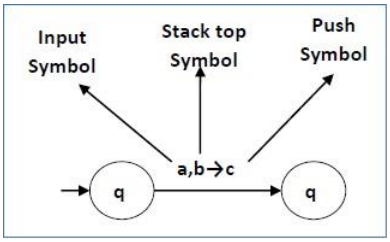


Figure 3: Transition in a PDA

This means at state $\mathbf{q_1}$, if we encounter an input string **'a'** and top symbol of the stack is **'b'**, then we pop **'b'**, push **'c'** on top of the stack and move to state $\mathbf{q_2}$.

A language can be accepted by PDA using two approaches:

1. Acceptance by Final State: The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

2. Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

PDA Example: Design a PDA for accepting a language $\{\boldsymbol{a^n b^{2n}} \mid \boldsymbol{n \geq 1}\}$. ➔ abb, aabbbb, aaabbbbbb, etc.

Solution: In this language, n number of a's should be followed by 2n number of b's. Hence, we will apply a very simple logic, and that is if we read single 'a', we will push two a's onto the stack. As soon as we read 'b', then for every single 'b', only one 'a' should get popped from the stack.

The ID can be constructed as follows:

1. $\delta(s_0, a, Z) = (s_0, aaZ)$
2. $\delta(s_0, a, a) = (s_0, aaa)$

Now, when we read b, we will change the state from s_0 to s_1 and start popping corresponding 'a'. Hence,

3. $\delta(s_0, b, a) = (s_1, \varepsilon)$

   Thus this process of popping 'b' will be repeated unless all the symbols are read. Note that popping action occurs in state s_1 only.

4. $\delta(s_1, b, a) = (s_1, \varepsilon)$

   After reading all b's, all the corresponding a's should get popped. Hence when we read ε as input symbol then there should be nothing in the stack. Hence, the move will be:

5. $\delta(s_1, \varepsilon, Z) = (s_2, \varepsilon)$

   where

   PDA = $(\{s_0, s_1, s_2\}, \{a, b\}, \{a, Z\}, \delta, s_0, Z, \{s_2\})$

We can summarize the ID as:

1. $\delta(s_0, a, Z) = (s_0, aaZ)$
2. $\delta(s_0, a, a) = (s_0, aaa)$
3. $\delta(s_0, b, a) = (s_1, \varepsilon)$
4. $\delta(s_1, b, a) = (s_1, \varepsilon)$
5. $\delta(s_1, \varepsilon, Z) = (s_2, \varepsilon)$

Now, we will simulate this PDA for the input string "aaabbbbbb".

$$\begin{aligned}\delta(s_0, aaabbbbbb, Z) &\vdash \delta(s_0, aabbbbbb, aaZ)\\ &\vdash \delta(s_0, abbbbbb, aaaaZ)\\ &\vdash \delta(s_0, bbbbbb, aaaaaaZ)\\ &\vdash \delta(s_1, bbbbb, aaaaaZ)\end{aligned}$$

$\vdash \delta(s_1, bbbb, aaaaZ)$

$\vdash \delta(s_1, bbb, aaaZ)$

$\vdash \delta(s_1, bbb, aaaZ)$

$\vdash \delta(s_1, bb, aaZ)$

$\vdash \delta(s_1, b, aZ)$

$\vdash \delta(s_1, \varepsilon, Z)$

$\vdash \delta(s_2, \varepsilon)$

ACCEPT

Turing Machine

A Turing Machine (TM), thought of by the mathematician Alan Turing in 1936, 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. **Despite its simplicity, the TM can simulate ANY computer algorithm, no matter how complicated it is!**

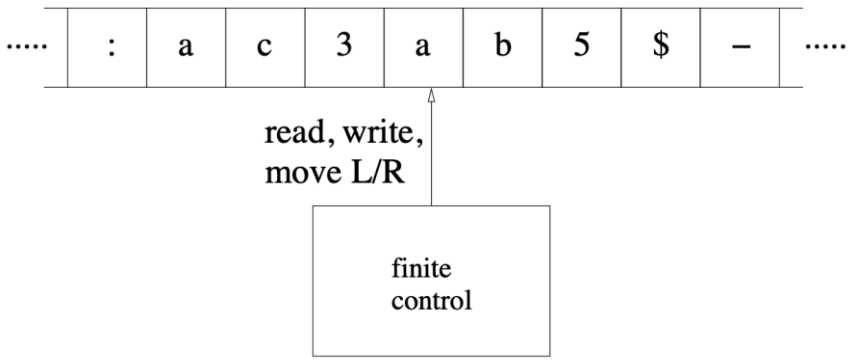


Figure 4: Design of Turing Machine

→ Recall that FAs are '**essentially memoryless'**, whilst PDAs are equipped with **memory in the form of a stack**.

→ To find the right kinds of machines for the top two Chomsky levels, we need to allow more general manipulation of memory.

→ A Turing machine consists of a finite-state control unit, equipped with a memory tape, infinite in both directions. Each cell on the tape contains a symbol drawn from a finite tape alphabet T.

A TM can be formally described as a **7-tuple (Σ, T, S, δ, s_0, #, F),** where

1. Σ is the input alphabet
2. **T** is the tape alphabet (wheras Σ ⊂ **T)**
3. **S** is a finite set of states
4. **δ** is a transition function; δ : S × T → S × T × {Left_shift, Right_shift, or no_movement}.
5. s_0 is the initial state **∈ S**
6. # is the blank symbol ∈ T − Σ
7. **F** is a set of final states (F ⊂ S)

TM Example: Bit Inversion

With the symbols "1 1 0" printed on the tape, let's attempt to convert the 1s to 0s and vice versa. This is called bit inversion, since 1s and 0s are bits in binary. This can be done by passing the following instructions to the Turing machine, utilising the machine's reading capabilities to decide its subsequent operations on its own. These instructions make up a simple program.

TM = (Σ, T, S, δ, S_0, #, F) with

- Σ = {1, 0}
- T = {1, 0, #}
- S = {s_0, s_f }
- S_0 = {s_0}
- # = blank symbol
- F = {s_f }

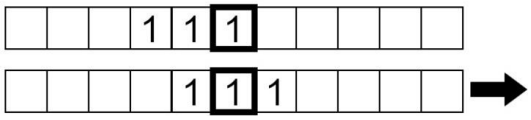
δ is given by

Current state	Symbol read		Symbol written	New state	Move instruction
s_0	Blank (#)	→	None	s_f	None
s_0	0	→	Write 1	s_0	Move tape to the right
s_0	1	→	Write 0	s_0	Move tape to the right

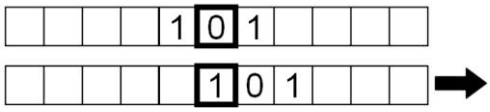
The machine will first read the symbol under the head, write a new symbol accordingly, then move the tape left or right as instructed, before repeating the read-write-move sequence again.

Let's see what this program does to our tape from the previous end point of the instructions:

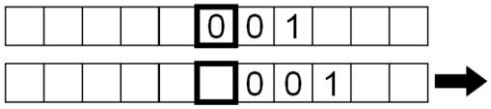
The current symbol under the head is 0, so we write a 1 and move the tape right by one square.



The symbol being read is now 1, so we write a 0 and move the tape right by one square:



Similarly, the symbol read is a 1, so we repeat the same instructions.



Finally, a 'blank' symbol is read, so the machine does nothing apart from read the blank symbol continuously since we have instructed it to repeat the read-write-move sequence without stopping.

After every instruction, we also specify a state for the machine to transition to. In the example, the machine is redirected back to its original state, State 0, to repeat the read-write-move sequence, unless a blank symbol is read. When the machine reads a blank symbol, the machine is directed to a stop state and the program terminates.

Linear-bounded automaton (LBA)

Turing machine that uses only the tape space occupied by the input is called a linear-bounded automaton (LBA).

An **LBA** can be formally described as a **9-tuple (Σ, T, S, δ, M_L, M_R, s_0, #, F)** where

1. Σ is the input alphabet
2. **T** is the tape alphabet (wheras $\Sigma \subset$ **T)**
3. **S** is a finite set of states
4. **δ** is a transition function; $\delta : S \times T \rightarrow S \times T \times$ {Left_shift, Right_shift, or no_movement}.
5. **M_L** is the left end marker
6. **M_R** is the right end marker
7. **s_0** is the initial state **$\in$ S**
8. **#** is the blank symbol $\in T - \Sigma$
9. **F** is a set of final states ($F \subset S$)

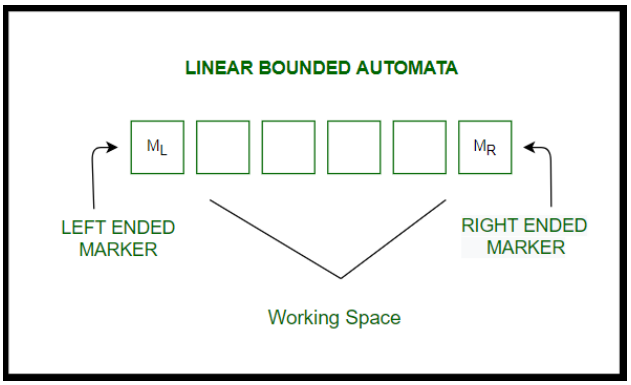


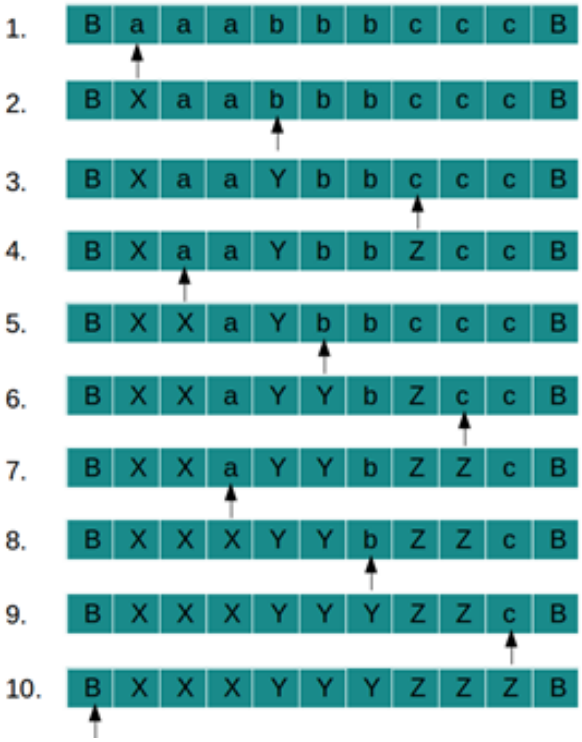
Figure 5: Diagrammatic representation of LBA

LBA Example: $a^nb^nc^n \mid n \geq 1$ ➔ abc, aabbcc, aaabbbccc, etc.

Approach for $a^nb^nc^n \mid n \geq 1$

- Mark 'a' then move right.
- Mark 'b' then move right
- Mark 'c' then move left
- Come to far left till we get 'X'
- Repeat above steps till all 'a', 'b' and 'c' are marked
- At last, if everything is marked that means string is accepted.

Tape movement for string "aaabbbccc":

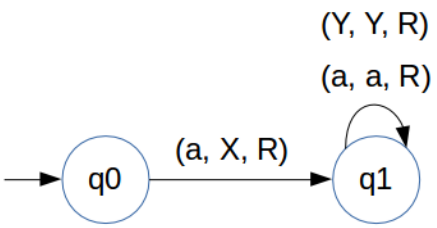


LBA = $(\Sigma, T, S, \delta, S_0, M_L, M_R, F)$ with

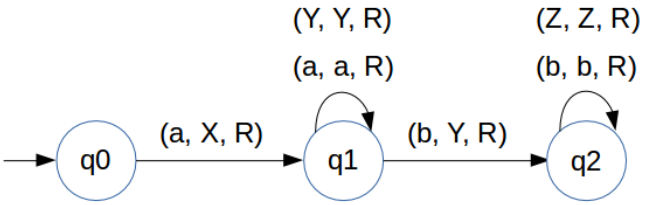
- $\Sigma = \{a, b, c\}$
- $T = \{a, b, c, X, Y, Z\}$
- $S = \{q_0, q_1, q_2, q_3, q_4, q_f\}$
- $S_0 = \{q_0\}$
- The two B's on the left and right sides of the tape correspond to M_L and M_R, respectively.
- $F = \{ q_f \}$

δ is given by the following state transition diagram:

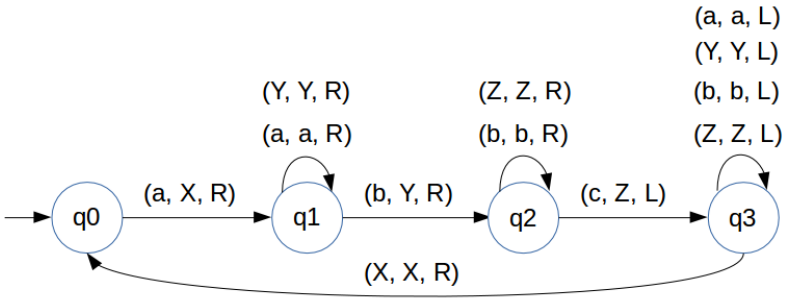
1. Following Steps:
   a. Mark 'a' with 'X' and move towards unmarked 'b'
   b. Move towards unmarked 'b' by passing all 'a's
   c. To move towards unmarked 'b' also pass all 'Y's if exist



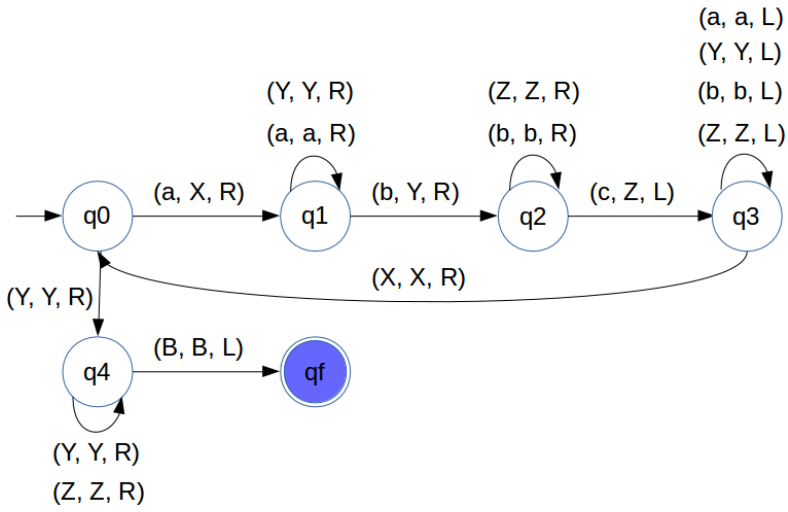
2. Following Steps:
   a. Mark 'b' with 'Y' and move towards unmarked 'c'
   b. Move towards unmarked 'c' by passing all 'b's
   c. To move towards unmarked 'c' also pass all 'Z's if exist



3. Following Steps:
   a. Mark 'c' with 'Z' and move towards first 'X' (in left)
   b. Move towards first 'X' by passing all 'Z's, 'b's, 'Y's and 'a's
   c. When 'X' is reached just move one step right by doing nothing.



4. To check all the 'a's, 'b's and 'c's are over add loops for checking 'Y' and 'Z' after "we get 'X' followed by 'Y'"
   To reach final state(qf) just replace BLANK with BLANK and move either direction



Conclusion: Compare FA, PDA, LBA, and TM

S. No.	Finite Automaton (FA)	Push Down Automaton (PDA)	Linear Bounded Automaton (LBA)	Turing Machine (TM)
1	Five tuple automata	Seven tuple automata	Nine tuple automata	Seven tuple automata
2	No auxiliary memory	Unlimited auxiliary memory in the form of stack	Limited memory corresponding to the size of the input on the tape	Unlimited memory in the form of tape
3	Less power full than PDA in terms of language acceptance	More power full than FA in terms of language acceptance	More power full than FA & PDA in terms of language acceptance	More power full than FA, PDA, and LBA in terms of language acceptance
4	Accepting only regular languages	Accepting regular and context free languages	Accepting regular, context free, and context sensitive languages	Accepting all class of languages
5	Tape head can perform only read operation	Tape head can perform only read operation	Tape head can perform both read and write operation	Tape head can perform both read and write operation
6	Tape head can travel only in forward (right) direction	Tape head can travel only in forward (right) direction	Tape head can travel in both right & left direction	Tape head can travel in both right & left direction
7	One-way infinite tape	One-way infinite tape	Two-way finite tape	Two-way infinite tape
8	FA stops its work only when the input string in the tape has been processed completely	PDA stops its work only when the input string in the tape has been processed completely	LBA stops its work immediately when it enters into the final state irrespective of the input string in the tape	TM stops its work immediately when it enters into the final state irrespective of the input string in the tape

Decidability

A language $L \subseteq \Sigma^*$ is **decidable** if and only if (iff) the characteristic function χ of L, i.e. χ_L: $\Sigma^* \rightarrow \{0, 1\}$ with

$$\chi_L(x) = \begin{cases} 1, & w \in L \\ 0, & w \notin L \end{cases} \qquad Def(\chi_L) = \Sigma^*$$

is computable.

A language $L \subseteq \Sigma^*$ is **semi-decidable** iff. the semi-characteristic function χ^l of L, *i.e.* χ_L^l: $\Sigma^* \rightarrow \{0, 1\}$ with

$$\chi_L^l(x) = \begin{cases} 1, & w \in L \\ \perp, & w \notin L \end{cases} \qquad Def(\chi_L^l) = L \rightarrow for\ \Sigma^*\text{-}L\ \chi_L^l(x)\ will\ never\ terminate$$

is computable.

Relationship between decidable and semi-decidable languages

A language $L \subseteq \Sigma^*$ is **decidable** iff. L and $\bar{L} = \Sigma^* - L$ are semi-decidable.

$$\chi_L^l(x) = \begin{cases} 1, & \chi_L(x) = 1 \\ \perp, & \chi_L(x) = 0 \end{cases}$$

$$\chi_{\bar{L}}^l(x) = \begin{cases} 1, & \chi_{\bar{L}}(x) = 0 \\ \perp, & \chi_{\bar{L}}(x) = 1 \end{cases}$$

$=>$ χ_L^l and $\chi_{\bar{L}}^l$ are computable. The two algorithms should be cleverly brought together.

Decidability in the context of word problem:

Decidability of the word problem for a language class

te_Σ: C_Σ x $\Sigma^* \rightarrow \{0, 1\}$: $te_\Sigma(L, w) = \begin{cases} 1, & w \in L \\ 0, & w \notin L \end{cases}, \ \forall L \in C_\Sigma, \forall w \in \Sigma*$

Here: C_Σ = Typ-i$_\Sigma$, $0 \leq i \leq 3$

Consider G = (Σ, N, P, S), w ∈ Σ^*

➔ Question: w ∈ L(G)?

Typ 3: A → aB | ε

Typ 2: A → BC | a

Typ 1: $(\Sigma \cup N)^* - \Sigma^* \times (\Sigma \cup N)^*$ with $|\alpha| \leq |\beta|$ **(=> Monotony property)**

Typ 0: ? -> semi-decidable, 1,2,3 -> decidable because of monotony property

Further semi-decidable Problems:

Halting Problem

In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever.

$$H = \begin{cases} w\&x \mid < T > = w \text{ and} \\ T \text{ stops when } x \text{ is entered as input} \end{cases}$$

Is χ_H *computable*?

- ⇨ Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.
- ⇨ The halting problem is semi-decidable.

Post Correspondence Problem

$$PCP: x = \langle (v_1, w_1), (v_2, w_2), \dots, (v_k, w_k) \rangle$$

$$v_1, w_1 \in \sum^+,$$

$$\exists i_1, i_2, \dots, i_l \in \{1, \dots, k\} \; with$$

$v_{i_1} v_{i_2} \dots v_{i_l} = w_{i_1} w_{i_2} \dots w_{i_l}$? (if the equality is true, then we have a **solution**!)

$$\text{PCP}_{\sum} = \{x_k \mid x_k \; has \; solution \; i_1, i_2, \dots, i_l \in \{1, \dots, k\}\}$$

Is χ_{PCP} *computable*?

- ⇨ The PCP is another undecidable decision problem introduced by Emil Post in 1946
- ⇨ The PCP is semi-decidable.

Consider the following two lists:

v_1	v_2	v_3
a	*ab*	*bba*

w_1	w_2	w_3
baa	*aa*	*bb*

A solution to this problem would be the sequence (3, 2, 3, 1), because

$v_3 v_2 v_3 v_1$ = bba ° ab ° bba ° a = bbaabbbaa = bb ° aa ° bb ° baa = $w_3 w_2 w_3 w_1$

Furthermore, since (3, 2, 3, 1) is a solution, so are all of its "repetitions", such as (3, 2, 3, 1, 3, 2, 3, 1), etc.; that is, when a solution exists, there are infinitely many solutions of this repetitive kind.

However, if the two lists had consisted of only v_2, v_3 and w_2, w_3 from those sets, then there would have been no solution (the last letter of any such v string is not the same as the letter before it, whereas w only constructs pairs of the same letter).

- ⇨ Both the halting problem and PCP are often used in proofs of undecidability.

Homeworks:

1) Consider the language L = { w#w | w $\in${0,1}* }. Design a TM that recognizes the language L.

2) A Turing Machine (TM) M that decides L = { 02^n | n ≥ 0 } which is the language consisting of all strings of 0s whose length is a power of 2.

3) A TM to add 1 to a binary number (with a 0 in front)