COMPUTABILITY THEORY

After going through this chapter, you should be able to understand :

- Chomsky hierarchy of Languages
- Linear Bounded Automata and CSLs
- LR (0) Grammar
- Decidability of problems
- UTM and PCP
- P and NP problems

8.1 CHOMSKY HIERARCHY OF LANGUAGES

Chomsky has classified all grammars in four categories (type 0 to type 3) based on the right hand side forms of the productions.

(a) Type 0

These types of grammars are also known as phrase structured grammars, and RHS of these are free from any restriction. All grammars are type 0 grammars.

Example : productions of types $AS \rightarrow aS,\ SB \rightarrow Sb, S \rightarrow \in$ are type 0 production.

(b) Type 1

We apply some restrictions on type 0 grammars and these restricted grammars are known as type 1 or **context - sensitive grammars** (CSGs). Suppose a type 0 production $\gamma\alpha\delta \rightarrow \gamma\beta\delta$ and the production $\alpha \rightarrow \beta$ is restricted such that $|\alpha| \leq |\beta|$ and $\beta \neq \in$. Then these type of productions is known as type 1 production. If all productions of a grammar are of type 1 production, then grammar is known as type 1 grammar. The language generated by a context - sensitive grammar is called context - sensitive language (CSL).

In CSG, there is left context or right context or both. For example, consider the production $\alpha A\beta \to \alpha a\beta$. In this, α is left context and β is right context of A and A is the variable which is replaced.

The production of type $S \to \in$ is allowed in type 1 if $\in$ is in L(G), but S should not appear on right hand side of any production.

Example : productions $S \to AB, S \to \in, A \to c$ are type 1 productions, but the production of type $A \to Sc$ is not allowed . Almost every language can be thought as CSL.

Note : If left or right context is missing then we assume that $\in$ is the context.

(c) Type 2

We apply some more restrictions on RHS of type 1 productions and these productions are known as type 2 or context - free productions. A production of the form $\alpha \to \beta$, where $\alpha, \beta \in (V \cup \Sigma)^*$ is known as type 2 production. A grammar whose productions are type 2 production is known as type 2 or context - free grammar (CFG) and the languages generated by this type of grammars is called context - free languages (CFL).

Example : $S \to S + S, S \to S * S, S \to id$ are type 2 productions.

(d) Type 3

This is the most restricted type. Productions of types $A \to a$ or $A \to aB | Ba$, where $A, B \in V$, and $a \in \Sigma$ are known as type 3 or regular grammar productions. A production of type $S \to \in$ is also allowed, if $\in$ is in generated language.

Example : productions $S \to aS,\ S \to a$ are type 3 productions.

Left - linear production : A production of type $A \to Ba$ is called left - linear production.

Right - linear production : A production of type $A \to aB$ is called right - linear production. A left - linear or right - linear grammar is called regular grammar. The language generated by a regular grammar is known as regular language.

8.2 LINEAR BOUNDED AUTOMATA

The Linear Bounded Automata (LBA) is a model which was originally developed as a model for actual computers rather than model for computational process. A linear bounded automaton is a restricted form of a non deterministic Turing machine.

A linear bounded automaton is a multitrack Turing machine which has only one tape and this tape is exactly of same length as that of input.

The linear bounded automaton (LBA) accepts the string in the similar manner as that of Turing machine does. For LBA halting means accepting. In LBA computation is restricted to an area bounded by length of the input. This is very much similar to programming environment where size of variable is bounded by its data type.

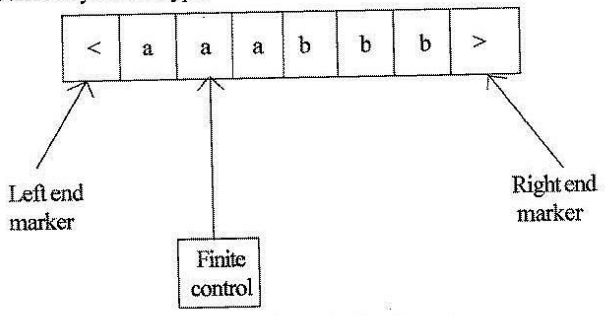


FIGURE : Linear bounded automaton

The LBA is powerful than NPDA but less powerful than Turing machine. The input is placed on the input tape with beginning and end markers. In the above figure the input is bounded by < and >.

A linear bounded automata can be formally defined as :

LBA is 7 - tuple on deterministic Turing machine with

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}) \text{ having}$$

1. Two extra symbols of left end marker and right end marker which are not elements of Γ.
2. The input lies between these end markers.
3. The TM cannot replace < or > with anything else nor move the tape head left of < or right of >.

8.3 CONTEXT SENSITIVE LANGUAGES (CSLs)

The context sensitive languages are the languages which are accepted by linear bounded automata. These type of languages are defined by context sensitive grammar. In this grammar more than one terminal or non terminal symbol may appear on the left hand side of the production rule. Along with it, the context sensitive grammar follows following rules :

i. The number of symbols on the left hand side must not exceed number of symbols on the right hand side.
ii. The rule of the form $A \rightarrow \in$ is not allowed unless A is a start symbol. It does not occur on the right hand side of any rule.

The classic example of context sensitive language is $L = \{a^n\ b^n\ c^n \mid n \geq 1\}$. The context sensitive grammar can be written as :

S	→	aBC
S	→	SABC
CA	→	AC
BA	→	AB
CB	→	BC
aA	→	aa
aB	→	ab
bB	→	bb
bC	→	bc
cC	→	cc

Now to derive the string aabbcc we will start from start symbol :

S	rule S →	SABC
$\underline{\text{S}}\text{ABC}$	rule S →	aBC
$\text{aB}\underline{\text{CA}}\text{BC}$	rule CA →	AC
$\text{aBA}\underline{\text{CB}}\text{C}$	rule CB →	BC
$\text{a}\underline{\text{BA}}\text{BCC}$	rule BA →	AB
$\underline{\text{aA}}\text{BBCC}$	rule aA →	aa
$\text{a}\underline{\text{aB}}\text{BCC}$	rule aB →	ab
$\text{aa}\underline{\text{bB}}\text{CC}$	rule bB →	bb
$\text{aab}\underline{\text{bC}}\text{C}$	rule bC →	bc
$\text{aabb}\underline{\text{cC}}$	rule cC →	cc
aabbcc		

Note : The language $a^n\ b^n\ c^n$ where $n \geq 1$ is represented by context sensitive grammar but it can not be represented by context free grammar.

Every context sensitive language can be represented by LBA.

8.4 LR (k) GRAMMARS

Before going to the topic of LR (k) grammar, let us discuss about some concepts which will be helpful understanding it.

In the unit of context free grammars you have seen that to check whether a particular string is accepted by a particular grammar or not we try to derive that sentence using rightmost derivation or leftmost derivation. If that string is derived we say that it is a valid string.

Example :

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to id \mid (E)$$

Suppose we want to check validity of a string id + id * id . Its rightmost derivation is

$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow E + T * F \\ &\Rightarrow E + T * id \\ &\Rightarrow E + F * id \\ &\Rightarrow E + id * id \\ &\Rightarrow T + id * id \\ &\Rightarrow F + id * id \\ &\Rightarrow id + id * id \end{aligned}$$

FIGURE(a) : Rightmost Derivation of id + id * id

Since this sentence is derivable using the given grammar. It is a valid string. Here we have checked the validity of string using process known as derivation.

In reduction process we have seen that we repeat the process of substitution until we get starting state. But some times several choices may be available for replacement. In this case we have to backtrack and try some other substring . For certain grammars it is possible to carry out the process in deterministic. (i. e., having only one choice at each time). LR grammars form one such subclass of context free grammars. Depending on the number of look ahead symbolized to determine whether a substring must be replaced by a non terminal or not, they are classified as LR(0) , LR(1).... and in general LR(k) grammars.

LR(k) stands for left to right scanning of input string using rightmost derivation in reverse order (we say reverse order because we use reduction which is reverse of derivation) using look ahead of k symbols.

8.4.1 LR(0) Grammar

LR(0) stands for left to right scanning of input string using rightmost derivation in reverse order using 0 look ahead symbols.

Before defining LR(0) grammars, let us know about few terms.

Prefix Property : A language L is said to have prefix property if whenever w in L, no proper prefix of w is in L. By introducing marker symbol we can convert any DCFL to DCFL with prefix property. Hence $L\$ = \{ w\$ \mid w \in L \}$ is a DCFL with prefix property whenever w is in L.

Example : Consider a language L = { cat, cart, bat, art, car } . Here, we can see that sentence cart is in L and its one of the prefixes car is also is in L. Hence, it is not satisfying property. But L$ = { cat $, cart $, bat $, art $, car $ }

Here, cart $ is in L$ but its prefix cart or car are not present in L$. Similarly no proper prefix is present in L$. Hence, it is satisfying prefix property.

Note : LR(0) grammar generates DCFL and every DCFL with prefix property has a LR(0) grammar.

LR Items

An item for a CFG is a production with dot any where in right side including beginning or end. In case of $\in$ production, suppose $A \rightarrow \in, A \rightarrow .$ is an item.

Computing Valid Item Sets

The main idea here is to construct from a given grammar a deterministic finite automata to recognize viable prefixes. We group items together into sets which give to states of DFA. The items may be viewed as states of NFA and grouped items may be viewed as states of DFA obtained using subset construction algorithm.

To compute valid set of items we use two operations goto and closure.

Closure Operation

It I is a set of items for a grammar G, then closure (I) is the set of items constructed from I by two rules.

1. Initially, every item I is added to closure (I).
2. If $A \to \alpha.B\beta$ is in closure (I) and $B \to \delta$ is production then add item $B \to \delta$ to I, if it is not already there. We apply this rule until no more new items can be added to closure (I).

Example : For the grammar,

$$S' \to S$$
$$S \to cAd$$
$$A \to a$$

If $S' \to S$ is set of one item in state I then closure of I is,

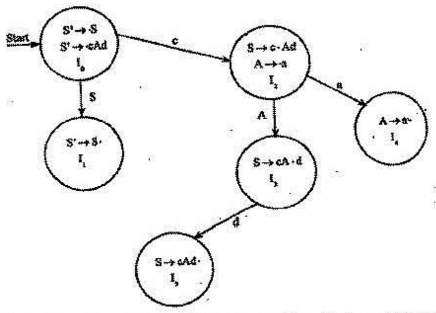
$$I_1 : S' \to .s$$
$$S \to .cAD$$

The first item is added using rule 1 and $S \to .cAd$ is added using rule 2. Because '.' is followed by nonterminal S we add items having S in LHS. In $S \to .cAd$ '.' is followed by terminal so no new item is added.

Goto Function : It is written as goto (I, X) where I is set of items and X is grammar symbol.

If $A \to \alpha.X\beta$ is in some item set I then goto (I, X) will be closure of set of all item $A \to \alpha.X.\beta$.

DFA :



FIGURE(a) : DFA whose States are the Sets of Valid Items

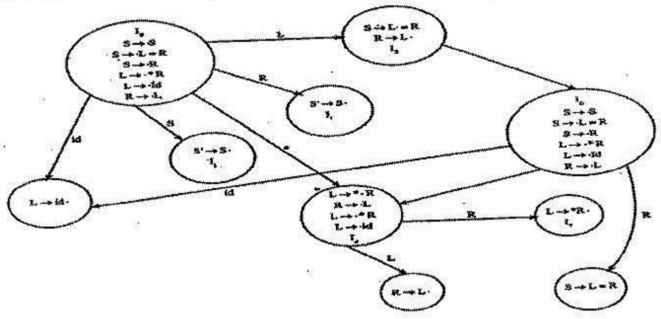
Definition of LR(0) Grammar : We say G is an LR (0) grammar if,

1. Its start symbol does not appear on the right hand side of any production and
2. For every viable prefix γ of G, whenever $A \to \alpha$ is a complete item valid for γ, then no other complete item nor any item with terminal to the right of the dot is valid for γ.

Condition 1 : For a grammar to be LR(0) it should satisfy both the conditions. The first condition can be made to satisfy by all grammars by introduction of a new production $S' \to S$ is known augmented grammar.

Condition 2 : For the DFA shown in Figure(a), the second condition is also satisfied because in the item sets I_1, I_4 and I_5 each containing a complete item, there are no other complete items nor any other conflict.

Example : Consider the DFA given in figure(b).



FIGURE(b) : DFA for the given Grammar

Each problem P is a pair consisting of a set and a question, where the question can be applied to each element in the set. The set is called the domain of the problem, and its elements are called the instances of the problem.

Example :

Domain = { All regular languages over some alphabet Σ } ,
Instance : L = { w : w is a word over Σ ending in abb} ,
Question : Is union of two regular languages regular ?

8.5.1 Decidable and Undecidable Problems

A problem is said to be decidable if

1. Its language is recursive, or
2. It has solution

Other problems which do not satisfy the above are undecidable. We restrict the answer of decidable problems to " YES" or "NO" . If there is some algorithm exists for the problem, then outcome of the algorithm is either "YES" or "NO" but not both. Restricting the answers to only "YES" or "NO" we may not be able to cover the whole problems, still we can cover a lot of problems. One question here. Why we are restricting our answers to only "YES" or "NO"? The answer is very simple ; we want the answers as simple as possible.

Now, we say " If for a problem, there exists an algorithm which tells that the answer is either "YES" or "NO" then problem is decidable."

If for a problem both the answers are possible ; some times "YES" and sometimes "NO", then problem is undecidable.

8.5.2 Decidable Problems for FA, Regular Grammars and Regular Languages

Some decidable problems are mentioned below :

1. Does FA accept regular language ?
2. Is the power of NFA and DFA same ?
3. L_1 and L_2 are two regular languages. Are these closed under following :
   (a) Union
   (b) Concatenation
   (c) Intersection
   (d) Complement

6. We have following co - theorem based on above discussion for recursive enumerable and recursive languages.

Let L and $\overline{L}$ are two languages, where $\overline{L}$ the complement of L, then one of the following is true :

(a) Both L and $\overline{L}$ are recursive languages,

(b) Neither L nor $\overline{L}$ is recursive languages,

(c) If L is recursive enumerable but not recursive, then $\overline{L}$ is not recursive enumerable and vice versa.

Undecidable Problems about Turing Machines

In this section, we will first discuss about halting problem in general and then about TM.

Halting Problem (HP)

The **halting problem** is a decision problem which is informally stated as follows :

"Given a description of an algorithm and a description of its initial arguments, determine whether the algorithm, when executed with these arguments, ever halts. The alternative is that a given algorithm runs forever without halting."

Alan Turing proved in 1936 that there is no general method or algorithm which can solve the halting problem for all possible inputs. An algorithm may contain loops which may be infinite or finite in length depending on the input and behaviour of the algorithm . The amount of work done in an algorithm usually depends on the input size. Algorithms may consist of various number of loops, nested or in sequence. The HP asks the question :

Given a program and an input to the program, determine if the program will eventually stop when it is given that input ?

One thing we can do here to find the solution of HP. Let the program run with the given input and if the program stops and we conclude that problem is solved. But, if the program doesn't stop in a reasonable amount of time, we can not conclude that it won't stop. The question is : " how long we can wait .... ?" . The waiting time may be long enough to exhaust whole life. So, we can not take it as easier as it seems to be. We want specific answer, either "YES" or "NO", and hence some algorithm to decide the answer.

Now, we analyse the following :

1. If H outputs "YES" and says that Q halts then Q itself would loop (that's how we constructed it).
2. If H outputs "NO" and says that Q loops then Q outputs "YES" and will halts.

Since , in either case H gives the wrong answer for Q. Therefore, H cannot work in all cases and hence can't answer right for all the inputs. This contradicts our assumption made earlier for HP. Hence, HP is undecidable.

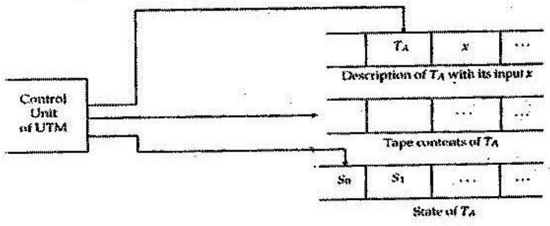
Theorem : HP of TM is undecidable.
Proof : HP of TM means to decide whether or not a TM halts for some input w. We can prove this following the similar steps discussed in above theorem.

8.6 UNIVERSAL TURING MACHINE

The Church - Turing thesis conjectured that anything that can be done on any existing digital computer can also be done by a TM. To prove this conjecture. A. M. Turing was able to construct a single TM which is the theoretical analogue of a general purpose digital computer. This machine is called a Universal Turing Machine (UTM). He showed that the UTM is capable of initiating the operation of any other TM, that is, it is a reprogrammable TM. We can define this machine in more formal way as follows :

Definition : A Universal Turing Machine (denoted as UTM) is a TM that can take as input an arbitrary TM T_A with an arbitrary input for T_A and then perform the execution of T_A on its input.

What Turing thus showed that a single TM can acts like a general purpose computer that stores a program and its data in memory and then executes the program. We can describe UTM as a 3 - tape TM where the description of TM, T_A and its input string $x \in A^*$ are stored initially on the first tape, t_1. The second tape, t_2 used to hold the simulated tape of T_A, using the same format as used for describing the TM, T_A. The third tape , t_3 holds the state of T_A



Now, suppose that a Turing machine, T_A, is consisting of a finite number of configurations, denoted by, $c_0, c_1, c_2, \ldots, c_p$ and let $\overline{c}_0, \overline{c}_1, \overline{c}_2, \ldots, \overline{c}_p$ represent the encoding of them. Then, we can define the encoding of T_A as follows :

$$* \overline{c}_0 \# \overline{c}_1 \# \overline{c}_2 \# \ldots\ldots \# \overline{c}_p *$$

Here, * and # are used only as separators, and cannot appear elsewhere. We use a pair of *'s to enclose the encoding of each configuration of TM, T_A.

The case where $\delta(s,a)$ is undefined can be encoded as follows :

$$\# \overline{s}\ 0\overline{a}\ 0\overline{B}\ \#$$

where the symbols $\overline{s}$, $\overline{a}$ and $\overline{B}$ stand for the encoding of symbols, s , a and B (Blank character), respectively.

Working of UTM

Given a description of a TM, T_A and its inputs representation on the UTM tape, t_1 and the starting symbol on tape , t_3, the UTM starts executing the quintuples of the encoded TM as follows :

1. The UTM gets the current state from tape, t_3 and the current input symbol from tape t_2.
2. then, it matches the current state - symbol pair to the state symbol pairs in the program listed on tape, t_1.
3. if no match occurs, the UTM halts, otherwise it copies the next state into the current state cell of tape, t_3, and perform the corresponding write and move operations on tape, t_2.
4. if the current state on tape, t_3 is the halt state, then the UTM halts, otherwise the UTM goes back to step 2.

8.7 POST'S CORRESPONDENCE PROBLEM (PCP)

Post's correspondence problem is a combinatorial problem formulated by Emil Post in 1946. This problem has many applications in the field theory of formal languages.

Definition :

A correspondence system P is a finite set of ordered pairs of nonempty strings over some alphabet.

Here, $u_1 = b,\ u_2 = a,\ u_3 = abc,\ \upsilon_1 = ca,\ \upsilon_2 = ab,\ \upsilon_3 = c$.

We have a solution $w = u_3\, u_2 = \upsilon_2\, \upsilon_1 = abca$.

8.8 TURING REDUCIBILITY

Reduction is a technique in which if a problem A is reduced to problem B then any solution of B solves A. In general, if we have an algorithm to convert some instance of problem A to some instance of problem B that have the same answer then it is called A reduces to B.

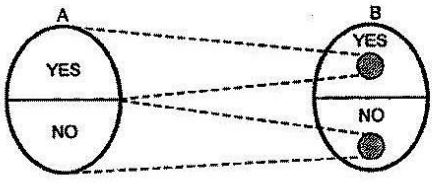


FIGURE: Reduction

Definition : Let A and B be the two sets such that $A, B \subseteq N$ of natural numbers. Then A is Turing reducible to B and denoted as $A \le_T B$.

If there is an oracle machine that computes the characteristic function of A when it is executed with oracle machine for B.

This is also called as A is B - recursive and B - computable. The oracle machine is an abstract machine used to study decision problem. It is also called as **Turing machine** with **black box.** We say that A is Turing equivalent to B and write $A \equiv_T B$ if $A \le_T B$ and $B \le_T A$.

Properties :

1. Every set is Turing equivalent to its complement.
2. Every computable set is Turing equivalent to every other computable set.
3. If $A \le_T B$ and $B \le_T C$ then $A \le_T B$.

8.9 DEFINITION OF P AND NP PROBLEMS

A problem is said to be solvable if it has an algorithm to solve it. Problems can be categorized into two groups depending on time taken for their execution.

1. The problems whose solution times are bounded by polynomials of small degree.
   Example: bubble sort algorithm obtains n numbers in sorted order in polynomial time $P(n) = n^2 - 2n + 1$ where n is the length of input. Hence, it comes under this group.

2. Second group is made up of problems whose best known algorithm are non polynomial example, travelling salesman problem has complexity of $O(n^2\ 2^n)$ which is exponential. Hence, it comes under this group.

A problem can be solved if there is an algorithm to solve the given problem and time required is expressed as a polynomial p(n), n being length of input string. The problems of first group are of this kind.

The problems of second group require large amount of time to execute and even require moderate size so these problems are difficult to solve. Hence, problems of first kind are tractable or easy and problems of second kind are intractable or hard.

8.9.1 P - Problem

P stands for deterministic polynomial time. A deterministic machine at each time executes an instruction. Depending on instruction, it then goes to next state which is unique.

Hence, time complexity of deterministic TM is the maximum number of moves made by M is processing any input string of length n, taken over all inputs of length n.

Definition : A language L is said to be in class P if there exists a (deterministic) TM M such that M is of time complexity P(n) for some polynomial P and M accepts L.
Class P consists of those problem that are solvable in polynomial time by DTM.

8.9.2 NP - Problem

NP stands for nondeterministic polynomial time.

The class NP consists of those problems that are verifiable in polynomial time. What we mean here is that if we are given certificate of a solution then we can verify that the certificate is correct in polynomial time in size of input problem.

8.10 NP - COMPLETE AND NP - HARD PROBLEMS

A problem S is said to be NP- Complete problem if it satisfies the following two conditions.

1. $S \in NP$, and
2. For every other problems $S_i \in NP$ for some $i = 1, 2, n,$ there is polynomial - time transformation from S_i to S i. e. every problem in NP class polynomial - time reducible to S.

We conclude one thing here that if S_i is NP - complete then S is also NP - Complete.

As a consequence, if we could find a polynomial time algorithm for S, then we can solve all NP problems in polynomial time, because all problems in NP class are polynomial - time reducible to each other.

"A problem P is said to be NP - Hard if it satisfies the second condition as NP - Complete, but not necessarily the first condition .".

The notion of NP - hardness plays an important role in the discussion about the relationship between the complexity classes P and NP. It is also often used to define the complexity class NP - Complete which is the intersection of NP and NP - Hard. Consequently, the class NP - Hard can be understood as the class of problems that are NP - complete or harder.

Example : An NP - Hard problem is the decision problem SUBSET - SUM which is as follows.

" Given a set of integers, do any non empty subset of them add up to zero? This is a yes / no question, and happens to be NP - complete ".

There are also decision problems that are NP - Hard but not NP - Complete , for example, the halting problem of Turing machine. It is easy to prove that the halting problem is NP - Hard but not NP - Complete. It is also easy to see that halting problem is not in NP since all problems in NP are decidable but the halting problem is not (voilating the condition first given for NP - complete languages).

In Complexity theory, the **NP - complete** problems are the hardest problems in NP class, in the sense that they are the ones most likely not to be in P class. The reason is that if we could find a way to solve any NP - complete problem quickly, then you could use that algorithm to solve all NP problems quickly.

At present time, all known algorithms for NP - complete problems require time which is exponential in the input size. It is unknown whether there are any faster algorithms for these are not.