

DIWAKAR EDUCATION HUB

Theory of Computation and Compilers Unit – 8

As per updated syllabus

DIWAKAR EDUCATION HUB

2020

THE LEARN WITH EXPERTIES

Theory of Computation

Automata theory (also known as Theory Of Computation) is a theoretical branch of Computer Science and Mathematics, which mainly deals with the logic of computation with respect to simple machines, referred to as automata.

Automata* enables the scientists to understand how machines compute the functions and solve problems. The main motivation behind developing Automata Theory was to develop methods to describe and analyse the dynamic behavior of discrete systems.

Automata is originated from the word “Automaton” which is closely related to “Automation”.

Now, understand the basic terminologies, which are important and frequently used in Theory of Computation.

Symbol: Symbol is the smallest building block, which can be any alphabet, letter or any picture.

a, b, c, 0, 1, ...

•

Alphabets (Σ): Alphabets are set of symbols, which are always **finite**.

Examples:

$\Sigma = \{0, 1\}$ is an alphabet of binary digits

$\Sigma = \{0, 1, \dots, 9\}$ is an alphabet of decimal digits

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

$\Sigma = \{A, B, C, \dots, Z\}$

Theory of Computation and Compilers Unit – 8

- **String:** String is a **finite** sequence of symbols from some alphabet. String is generally denoted as **w** and length of a string is denoted as **|w|**.
- Empty string is the string with
- zero occurrence of symbols,

represented as ϵ .

Number of Strings (of length 2)

that can be generated over the alphabet {a, b} -

- -

a a

a b

b a

b b

Length of String $|w| = 2$

Number of Strings = 4

Conclusion:

For alphabet {a, b} with length **n**, number of strings can be generated = 2^n .

Note – If the number of Σ 's is represented by $|\Sigma|$, then number of strings of length **n**, possible over Σ is $|\Sigma|^n$.

- **Language:** A language is a set of strings, chosen from some Σ^* or we can say- 'A language is a subset of Σ^* '. A language which can be formed over ' Σ ' can be **Finite** or **Infinite**.

Theory of Computation and Compilers Unit – 8

Example-

$L_1 = \{ \text{Set of all strings of length 2} \}$

$= \{ aa, ab, ba, bb \} \text{ -----} \rightarrow \text{Finite Language}$

$L_2 = \{ \text{Set of all strings which starts with 'a'} \}$

$= \{ a, aa, ab, aba, abaa, aab, ... \} \text{ -----} \rightarrow \text{Infinite Language}$

Powers of ' Σ ' :

Say $\Sigma = \{a, b\}$ then

$\Sigma^0 = \text{Set of all strings over } \Sigma \text{ of length 0. } \{ \epsilon \}$

$\Sigma^1 = \text{Set of all strings over } \Sigma \text{ of length 1. } \{a, b\}$

$\Sigma^2 = \text{Set of all strings over } \Sigma \text{ of length 2. } \{aa, ab, ba, bb\}$

i.e. $|\Sigma^2| = 4$ and Similarly, $|\Sigma^3| = 8$

Cardinality : Number of elements in a set, which is basically $|\Sigma|^n$.

Σ^* is a Universal Set.

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$

$= \{ \epsilon \} \cup \{a, b\} \cup \{aa, ab, ba, bb\}$

$= \dots \text{ //infinite language.}$

Theory of Computation

The theory of computation is a branch of computer science and mathematics combined that "deals with how efficiently problems can be solved on a model of computation, using an algorithm".

It studies the general properties of computation which in turn, helps us increase the efficiency at which computers solve problems. This is done when we estimate the validity of the solutions given by the computer through the

theory of computation and then alternate the algorithms so that we can obtain a

Theory of Computation and Compilers Unit – 8

more reliable solution.

This is a fast-growing branch that has helped solving problems in many fields beside computer science such as Physics, Economy, Biology and many others.

Before introducing you to the basics of the theory of computation, I would like to talk about the Turing machine, as it plays a good role in explaining this theory:

It is theoretical abstract machine used as a model of computation . It uses an infinite memory tape where the information obtained are saved, and analyze these information to determine whether the operation is feasible or not.

This machine was the creation of Alan Turing in 1936 by, he used it to prove properties of computation in general and answer these questions :

- "Does a machine exist that can determine whether any arbitrary machine on its tape is "circular""
- "Does a machine exist that can determine whether any arbitrary machine on its tape ever prints a given symbol"

Back to the computational theory, This theory is approached through three main fields:

- 1- Automata theory**
- 2- Computability theory**
- 3- Computational complexity theory**

These three branches basically uses algorithms to exploit the powers of computers in solving problems.

Here is an illustration of these fields :

Automata theory:

This branch was established in the 20th century by mathematicians. The main aim in this branch is to analyze the behavior of machines and how they solve a problem. The most powerful model of automata is the Turing machine.

Computability theory:

It is when we are able to formulate the problem using the Turing machine easily, but we can never solve it.. In other words, It is when the computer is able to address the problem but unable to come up with the solution

Computational complexity theory:

The complexity theory discusses the efficiency at which a problem could be solved. This is done considering two major aspects: time complexity and space complexity, which are the measures of the number of steps needed to analyze and solve the problem and thus determining the memory space needed to solve the problem.

In order to pre-determine the space and time that will be needed, computer scientists linked these two factors to the amount of input that was received, as the time and space needed increase linearly as the amount of input increases.

Languages :

A general definition of language must cover a variety of distinct categories: natural languages, programming languages, mathematical languages, etc. The notion of natural languages like English, Hindi, etc. is familiar to us. Informally, language can be defined as a system suitable for expression of certain ideas, facts, or concepts, which includes a set of symbols and rules to manipulate these. The languages we consider for our discussion is an abstraction of natural languages. That is, our focus here is on formal languages that need precise and formal

Theory of Computation and Compilers Unit – 8

definitions. Programming languages belong to this category. We start with some basic concepts and definitions required in this regard.

Symbols :

Symbols are indivisible objects or entity that cannot be defined. That is, symbols are the atoms of the world of languages. A symbol is any single object such as ♣, a, 0, 1, #, begin, or do. Usually, characters from a typical keyboard are only used as symbols.

Formal language theory

A **formal language** is a set of strings over a finite alphabet. **Formal language theory** is the study of formal languages, or often more accurately the study of families of formal languages. It deals with hierarchies of language families defined in a wide variety of ways. Formal language theory is concerned with the purely syntactical aspects, rather than a semantics or meaning of the strings^[1].

It is closely related to automata theory, which deals with formally defined machines that accept (or generate, according to the viewpoint) formal languages.

It is also related to complexity theory, because once a formal system capable of universal computation is fixed, a computational problem is nothing but a formal language containing all the descriptions of problem instances, and the answer to the problem is the subset of instances that would be accepted (could be derived) -- which need not be effective, and need not be a computable set, of course.

Computable Problems –

You are familiar with many problems (or functions) that are computable (or decidable), meaning there exists some algorithm that computes an answer (or output) to any instance of the problem (or for any input to the function) in a finite number of simple steps. A simple example is the integer increment operation:

$$f(x) = x + 1$$

Theory of Computation and Compilers Unit – 8

It should be intuitive that given any integer x , we can compute $x + 1$ in a finite number of steps. Since x is finite, it may be represented by a finite string of digits. Using the addition method (or algorithm) we all learned in school, we can clearly compute another string of digits representing the integer equivalent to $x + 1$.

Yet there are also problems and functions that are non-computable (or undecidable or uncomputable), meaning that there exists no algorithm that can compute an answer or output for all inputs in a finite number of simple steps. (Undecidable simply means non-computable in the context of a decision problem, whose answer (or output) is either “true” or “false”).

Non-Computable Problems –

A non-computable is a problem for which there is no algorithm that can be used to solve it. Most famous example of a non-computability (or undecidability) is the **Halting Problem**. Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes).

The alternative is that it runs forever without halting. The halting problem is about seeing if a machine will ever come to a halt when a certain input is given to it or if it will finish running. This input itself can be something that keeps calling itself forever which means that it will cause the program to run forever.

Other example of an uncomputable problem is: determining whether a computer program loops forever on some input. You can replace “computer program” by “Turing machine or algorithm” if you know about Turing machine.

Proving Computability or Non-Computability –

We can show that a problem is computable by describing a procedure and proving that the procedure always terminates and always produces the correct answer. It is enough to provide a convincing argument that such a procedure exists finding the actual procedure is not necessary (but often helps to make the argument more convincing).

Theory of Computation and Compilers Unit – 8

To show that a problem is not computable, we need to show that no algorithm exists that solves the problem. Since there are an infinite number of possible procedures, we cannot just list all possible procedures and show why each one does not solve the problem. Instead, we need to construct an argument showing that if there were such an algorithm it would lead to a contradiction.

The core of our argument is based on knowing the Halting Problem is noncomputable. If a solution to some new problem P could be used to solve the Halting Problem, then we know that P is also noncomputable. That is, no algorithm exists that can solve P since if such an algorithm exists it could be used to also solve the Halting Problem which we already know is impossible. The proof technique where we show that a solution for some problem P can be used to solve a different problem Q is known as a **reduction**. A problem Q is **reducible** to a problem P if a solution to P could be used to solve Q. This means that problem Q is no harder than problem P, since a solution to problem Q leads to a solution to problem P.

Some Examples On Computable Problems –

These are four simple examples of computable problem:

1. Computing the greatest common divisor of a pair of integers.
2. Computing the least common multiple of a pair of integers.
3. Finding the shortest path between a pair of nodes in a finite graph.
4. Determining whether a propositional formula is a tautology.

Some Examples On Computable Problems – State Entry Problem.

Consider the problem of determining if a string 'w' is given to some Turing machine 'M' will it enter some state 'q' (where q belongs to set of all states in Turing machine M and string w is not equal to empty string). Is it computable or non-computable?

Explanation –

We show the State Entry Problem is non-computable by showing that it is as hard as the Halting Problem, which we already know is non-computable.

State Entry Problem is asking us on given string w if we start from initial state of Turing machine will it reach to a state q . Now this state entry problem can be converted to halting problem. Halting problem is whether our Turing machine ever halts and state entry problem is asking same thing whether this Turing machine halts at some state q if we give string w as input to the Turing machine M . So the state entry problem is non-computable as we converted it to halting problem which we already know is non-computable problem. So in this way we can prove non-computability.

The Diagonal Argument

The Cantor diagonal method, also called the Cantor diagonal argument or Cantor's diagonal slash, is a clever technique used by Georg Cantor to show that the integers and reals cannot be put into a one-to-one correspondence (i.e., the uncountably infinite set of real numbers is "larger" than the countably infinite set of integers). However, Cantor's diagonal method is completely general and applies to any set as described below.

Given any set S , consider the power set $T = P(S)$ consisting of all subsets of S . Cantor's diagonal method can be used to show that T is larger than S , i.e., there exists an injection but no bijection from S to T . Finding an injection is trivial, as can be seen by considering the function from S to T which maps an element s of S to the singleton set $\{s\}$. Suppose there exists a bijection ϕ from S to T and consider the subset D of S consisting of the elements d of S such that $\phi(d)$ does not contain d . Since ϕ is a bijection, there must exist an element x of S such that $\phi(x) = D$. But by the definition of D , the set D contains x if and only if $\phi(x) = D$ does not contain x . This yields a contradiction, so there cannot exist a bijection from S to T .

Theory of Computation and Compilers Unit – 8

Cantor's diagonal method applies to any set S , finite or infinite. If S is a finite set of cardinality n , then $T = P(S)$ has cardinality 2^n , which is larger than n . If S is an infinite set, then $T = P(S)$ is a bigger infinite set. In particular, the cardinality c of the real numbers $\mathbb{R}$, which can be shown to be isomorphic to $P(\mathbb{N})$, where $\mathbb{N}$ is the set of natural numbers, is larger than the cardinality $\aleph_0$ of $\mathbb{N}$. By applying this argument infinitely many times to the same infinite set, it is possible to obtain an infinite hierarchy of infinite cardinal numbers.

Another

This is just a second look at the question of the relative magnitudes of a set and the set of its subsets.

Let R be a set, and F a function that maps $x \in R$ to a subset of R , $F(x) \subset R$. In other words, $F: R \rightarrow 2^R$.

Such a function can be visualized in a square $R \times R$. For every $x \in R$, $F(x)$ can be depicted in the vertical unit segment just over x . We'll call it $G(x)$, $G(x) = \{x\} \times F(x)$.

Now, for some x , $G(x)$ crosses the diagonal $D = \{(x, x)\}$, $G(x) \cap D \neq \emptyset$; and for some x , $G(x)$ does not cross the diagonal, $G(x) \cap D = \emptyset$. Of course, $G(x) \cap D \neq \emptyset$ is equivalent to $x \in F(x)$, and $G(x) \cap D = \emptyset$ is equivalent to $x \notin F(x)$.

Introduce set Z of all x with the latter property:

$$Z = \{x \in R: G(x) \cap D = \emptyset\} = \{x \in R: x \notin F(x)\}.$$

(Quite obviously the introduction of the diagonal D was not strictly necessary. But its presence illuminates the reason why the proof of the following theorem is known as the diagonal argument.)

Theorem

For no $x \in R$, $F(x) = Z$.

Theory of Computation and Compilers Unit – 8

Proof

Suppose on the contrary that, for some a , $F(a) = Z$. We may ask then whether or not $a \in Z$. We shall see that both possibilities lead to a contradiction.

Assume first that $a \in Z$. This would mean $G(a) \cap D = \emptyset$, implying that $a \notin F(a)$, i.e., $a \notin Z$.

Assume now that $a \notin Z$. This would mean $G(a) \cap \neq \emptyset$, implying that $a \in F(a)$, i.e., $a \in Z$.

Corollary 1

No function $R \rightarrow 2^R$ can be onto.

Corollary 2

For any set R , the cardinality of 2^R is strictly greater than the cardinality of R :

$$|2^R| > |R|.$$

Corollary 3

The set of the reals is uncountable.

Proof

Let $R = [0, 1]$ and N is the set of positive integers. If we think of the reals as represented by their binary expansions, then, with some caveats, each such expansion is a characteristic function of a subset of N which associates with every $x \in R$ a subset of N , viz., the subset of the indices whose corresponding digits in the binary expansion of x are 1. Such an association tells us that $|R| = |2^N|$. And Corollary 2 then implies that $|R| > |N|$.

It is clear that uncountability of the reals and Russell's Paradox are of the same origin.

Russell's Paradox

Russell's paradox is the most famous of the logical or set-theoretical paradoxes. Also known as the Russell-Zermelo paradox, the paradox arises within naïve set theory by considering the set of all sets that are not members of themselves. Such a set appears to be a member of itself if and only if it is not a member of itself. Hence the paradox.

Some sets, such as the set of all teacups, are not members of themselves. Other sets, such as the set of all non-teacups, are members of themselves. Call the set of all sets that are not members of themselves "R." If R is a member of itself, then by definition it must not be a member of itself. Similarly, if R is not a member of itself, then by definition it must be a member of itself.

Although also noticed by Ernst Zermelo, the contradiction was not thought to be important until it was discovered independently by Bertrand Russell in the spring of 1901. Since then, the paradox has prompted a great deal of work in logic, set theory and the philosophy and foundations of mathematics.

The Paradox

Central to any theory of sets is a statement of the conditions under which sets are formed. In addition to simply listing the members of a set, it was initially assumed that any well-defined condition (or precisely specified property) could be used to determine a set. For example, if T is the property of being a teacup, then the set, S, of all teacups might be defined as $S = \{x: T(x)\}$, the set of all individuals, x, such that x has the property of being T. Even a contradictory property might be used to determine a set. For example, the property of being both T and not-T would determine the empty set, the set having no members.

More precisely, naïve set theory assumes the so-called naïve or unrestricted Comprehension Axiom, the axiom that for any formula $\phi(x)$ containing x as a free variable, there will exist the set $\{x: \phi(x)\}$ whose members are exactly those objects that satisfy $\phi(x)$. Thus, if the formula $\phi(x)$ stands for "x is prime", then $\{x: \phi(x)\}$ will be the set of prime numbers. If $\phi(x)$ stands for " $\sim(x = x)$ ", then $\{x: \phi(x)\}$ will be the empty set.

Theory of Computation and Compilers Unit – 8

But from the assumption of this axiom, Russell's contradiction follows. For example, if we let $\phi(x)$ stand for $x \in x$ and let $R = \{x: \sim\phi(x)\}$, then R is the set whose members are exactly those objects that are not members of themselves.

Is R a member of itself? If it is, then it must satisfy the condition of not being a member of itself and so it is not. If it is not, then it must not satisfy the condition of not being a member of itself, and so it must be a member of itself. Since by classical logic one case or the other must hold – either R is a member of itself or it is not – it follows that the theory implies a contradiction.

As Russell tells us, it was after he applied the same kind of reasoning found in Cantor's diagonal argument to a "supposed class of all imaginable objects" that he was led to the contradiction:

The comprehensive class we are considering, which is to embrace everything, must embrace itself as one of its members. In other words, if there is such a thing as "everything," then, "everything" is something, and is a member of the class "everything." But normally a class is not a member of itself. Mankind, for example, is not a man. Form now the assemblage of all classes which are not members of themselves. This is a class: is it a member of itself or not? If it is, it is one of those classes that are not members of themselves, i.e., it is not a member of itself. If it is not, it is not one of those classes that are not members of themselves, i.e. it is a member of itself. Thus of the two hypotheses – that it is, and that it is not, a member of itself – each implies its contradictory. This is a contradiction. (1919, 136)

Standard responses to the paradox attempt to limit in some way the conditions under which sets are formed. The goal is usually both to eliminate R (and similar contradictory sets) and, at the same time, to retain all other sets needed for mathematics. This is often done by replacing the unrestricted Comprehension Axiom with the more restrictive Separation Axiom, namely the axiom that given any (consistent) set S and any formula $\phi(x)$ with x free, there will be a set $\{x \in S: \phi(x)\}$ whose members are exactly those members of S that satisfy $\phi(x)$. If we now let $\phi(x)$ stand for the formula $x \notin x$, it turns out that the corresponding set,

$\{x \in S: x \notin x\}$ will not be contradictory since it consists only of those members found within S that are not members of themselves. Hence the set fails to include itself.

A variety of related paradoxes is discussed in the second chapter of the Introduction to Whitehead and Russell (1910, 2nd edn 60-65), as well as in the entry on paradoxes and contemporary logic in this encyclopedia.

2. History of the Paradox

Russell appears to have discovered his paradox in the late spring of 1901, while working on his Principles of Mathematics (1903). Exactly when the discovery took place is not clear. Russell initially states that he came across the paradox “in June 1901” (1944, 13). Later he reports that the discovery took place “in the spring of 1901” (1959, 75). Still later he reports that he came across the paradox, not in June, but in May of that year (1969, 221). Cesare Burali-Forti, an assistant to Giuseppe Peano, had discovered a similar antinomy in 1897 when he noticed that since the set of ordinals is well-ordered, it too must have an ordinal. However, this ordinal must be both an element of the set of all ordinals and yet greater than every such element.

Unlike Burali-Forti’s paradox, Russell’s paradox does not involve either ordinals or cardinals, relying instead only on the primitive notions of set and set inclusion. Zermelo noticed a similar contradiction sometime between 1897 and 1902, possibly anticipating Russell by some years (Ebbinghaus and Peckhaus 2007, 43–48; Tappenden 2013, 336), although Kanamori concludes that the discovery could easily have been as late as 1902 (Kanamori 2009, 411). In any case, the paradox was thought to be of minor importance until it was realized how detrimental it was to Gottlob Frege’s foundations for mathematics.

Russell wrote to Frege with news of his paradox on June 16, 1902. (For the relevant correspondence, see Russell (1902) and Frege (1902) in van Heijenoort (1967).) The paradox was of significance to Frege’s logical work since, in effect, it showed that the axioms Frege was using to formalize his logic were inconsistent.

Specifically, Frege's Axiom V requires that an expression such as $\phi(x)$ be considered both a function of the argument x and a function of the argument ϕ . (More precisely, Frege's Law states that the course-of-values of a concept f is identical to the course-of-values of a concept g if and only if f and g agree on the value of every argument, i.e., if and only if for every object x , $f(x) = g(x)$. See section 2.4.1 of the entry on Gottlob Frege in this encyclopedia for more discussion.) In effect, it was this ambiguity that allowed Russell to construct R in such a way that it could both be and not be a member of itself.

Russell's letter arrived just as the second volume of Frege's *Grundgesetze der Arithmetik* (The Basic Laws of Arithmetic, 1893, 1903) was in press. Immediately appreciating the difficulty the paradox posed, Frege added to the *Grundgesetze* a hastily composed appendix discussing Russell's discovery. In the appendix Frege observes that the consequences of Russell's paradox are not immediately clear. For example, "Is it always permissible to speak of the extension of a concept, of a class? And if not, how do we recognize the exceptional cases? Can we always infer from the extension of one concept's coinciding with that of a second, that every object which falls under the first concept also falls under the second? These are the questions," Frege notes, "raised by Mr Russell's communication" (1903, 127). Because of these worries, Frege eventually felt forced to abandon many of his views about logic and mathematics.

Even so, as Russell points out, Frege met the news of the paradox with remarkable fortitude:

As I think about acts of integrity and grace, I realise that there is nothing in my knowledge to compare with Frege's dedication to truth. His entire life's work was on the verge of completion, much of his work had been ignored to the benefit of men infinitely less capable, his second volume was about to be published, and upon finding that his fundamental assumption was in error, he responded with intellectual pleasure clearly submerging any feelings of personal disappointment. It was almost superhuman and a telling indication of that of which men are

capable if their dedication is to creative work and knowledge instead of cruder efforts to dominate and be known. (Quoted in van Heijenoort (1967), 127)

Of course, Russell too was concerned about the consequences of the contradiction. Upon learning that Frege agreed with him about the significance of the result, he immediately began writing an appendix for his own soon-to-be-released *Principles of Mathematics*. Entitled “Appendix B: The Doctrine of Types,” the appendix represents Russell’s first attempt at providing a principled method for avoiding what soon was to become known as “Russell’s paradox.”

3. Early Responses to the Paradox

The significance of Russell’s paradox can be seen once it is realized that, using classical logic, all sentences follow from a contradiction. For example, assuming both P and $\sim P$, any arbitrary proposition, Q , can be proved as follows: from P we obtain $P \vee Q$ by the rule of Addition; then from $P \vee Q$ and $\sim P$ we obtain Q by the rule of Disjunctive Syllogism. Because set theory underlies all branches of mathematics, many people began to worry that the inconsistency of set theory would mean that no mathematical proof could be completely trustworthy. Only by eliminating Russell’s paradox could mathematics as a whole regain its consistency.

Russell’s paradox ultimately stems from the idea that any condition or property may be used to determine a set. For example, the property of being evenly divisible by only itself and the number one distinguishes the set of prime numbers from the set of whole numbers. The property of having mammary glands distinguishes the set of mammals from reptiles, birds and other living organisms. The property of being both square and not square (or any other conjunction of contradictory properties) determines the empty set, and so on.

One early skeptic concerning an unrestricted Comprehension (or Abstraction) axiom was the originator of modern set theory, Georg Cantor. Even prior to Russell’s discovery, Cantor had rejected unrestricted Comprehension in favour of what was, in effect, a distinction between sets and classes, recognizing that some

properties (such as the property of being an ordinal) produced collections that were simply too large to be sets, and that any assumption to the contrary would lead to inconsistency. (Details can be found in Moore (1982), Hallett (1984) and Menzel (1984).)

Russell's own response to the paradox came with his aptly named theory of types. Believing that self-application lay at the heart of the paradox, Russell's basic idea was that we can avoid commitment to R (the set of all sets that are not members of themselves) by arranging all sentences (or, more precisely, all propositional functions, functions which give propositions as their values) into a hierarchy. It is then possible to refer to all objects for which a given condition (or predicate) holds only if they are all at the same level or of the same "type."

This solution to Russell's paradox is motivated in large part by adoption of the so-called vicious circle principle. The principle in effect states that no propositional function can be defined prior to specifying the function's scope of application. In other words, before a function can be defined, one must first specify exactly those objects to which the function will apply (the function's domain). For example, before defining the predicate "is a prime number," one first needs to define the collection of objects that might possibly satisfy this predicate, namely the set, N , of natural numbers.

As Whitehead and Russell explain,

An analysis of the paradoxes to be avoided shows that they all result from a kind of vicious circle. The vicious circles in question arise from supposing that a collection of objects may contain members which can only be defined by means of the collection as a whole. Thus, for example, the collection of propositions will be supposed to contain a proposition stating that "all propositions are either true or false." It would seem, however, that such a statement could not be legitimate unless "all propositions" referred to some already definite collection, which it cannot do if new propositions are created by statements about "all propositions." We shall, therefore, have to say that statements about "all propositions" are meaningless. ... The principle which enables us to avoid illegitimate totalities may

be stated as follows: “Whatever involves all of a collection must not be one of the collection”; or, conversely: “If, provided a certain collection had a total, it would have members only definable in terms of that total, then the said collection has no total.” We shall call this the “vicious-circle principle,” because it enables us to avoid the vicious circles involved in the assumption of illegitimate totalities. (1910, 2nd edn 37)

If Whitehead and Russell are right, it follows that no function’s scope of application will ever be able to include any object presupposed by the function itself. As a result, propositional functions (along with their corresponding propositions) will end up being arranged in a hierarchy of the kind Russell proposes.

Although Russell first introduced his theory of types in his 1903 *Principles of Mathematics*, he recognized immediately that more work needed to be done since his initial account seemed to resolve some but not all of the paradoxes. Among the alternatives he considered was a so-called substitutional theory (Galaugher 2013). This in turn led to type theory’s more mature expression five years later in Russell’s 1908 article, “Mathematical Logic as Based on the Theory of Types,” and in the monumental work he co-authored with Alfred North Whitehead, *Principia Mathematica* (1910, 1912, 1913). Russell’s type theory thus appears in two versions: the “simple theory” of 1903 and the “ramified theory” of 1908. Both versions have been criticized for being too ad hoc to eliminate the paradox successfully.

In response to Russell’s paradox, David Hilbert also expanded his program of building a consistent, axiomatic foundation for mathematics so that it included an axiomatic foundation for logic and set theory (Peckhaus 2004). Underlying this formalist approach was the idea of allowing the use of only finite, well-defined and constructible objects, together with rules of inference deemed to be absolutely certain.

Finally, Luitzen Brouwer developed intuitionism, whose basic idea was that one cannot assert the existence of a mathematical object unless one can define a procedure for constructing it.

Together, all of these responses helped focus attention on the connections between logic, language and mathematics. They also helped logicians develop an explicit awareness of the nature of formal systems and of the kinds of metalogical and metamathematical results that have proved to be central to research in the foundations of logic and mathematics over the past one hundred years.

4. Russell's Paradox in Contemporary Logic

Russell's paradox is sometimes seen as a negative development – as bringing down Frege's Grundgesetze and as one of the original conceptual sins leading to our expulsion from Cantor's paradise. W.V. Quine describes the paradox as an "antinomy" that "packs a surprise that can be accommodated by nothing less than a repudiation of our conceptual heritage" (1966, 11). Quine is referring to the Naïve Comprehension principle mentioned earlier. In symbols, the principle states that

$$(NC) \exists A \forall x (x \in A \equiv \phi),$$

where A is not free in the formula ϕ . This says, "There is a set A such that for any object x , x is an element of A if and only if the condition expressed by ϕ holds." Russell's paradox arises by taking ϕ to be the formula: $x \notin x$.

Despite Quine's comment, it is possible to see Russell's paradox in a more positive light. For one thing, although the matter remains controversial, later research has revealed that the paradox does not necessarily short circuit Frege's derivation of arithmetic from logic alone. Frege's version of NC (his Axiom V) can simply be abandoned. (For details, see the entry on Frege's Theorem.) For another, Church gives an elegant formulation of the simple theory of types that has proven fruitful even in areas removed from the foundations of mathematics. (For details, see the entry on Type Theory.) Finally, the development of axiomatic (as opposed to naïve) set theories which exhibit various ingenious and mathematically and

philosophically significant ways of dealing with Russell's paradox paved the way for stunning results in the metamathematics of set theory. These results have included Gödel's and Cohen's theorems on the independence of the axiom of choice and Cantor's continuum hypothesis. So let us see, roughly, how some of these methods – specifically, the so-called “untyped” methods – deal with Russell's paradox.

Zermelo replaces NC with the following axiom schema of Separation (or Aussonderungsaxiom):

$$(ZA) \forall A \exists B \forall x (x \in B \equiv (x \in A \wedge \phi)).$$

Again, to avoid circularity, B cannot be free in ϕ . This demands that in order to gain entry into B , x must be a member of an existing set A . As one might imagine, this requires a host of additional set-existence axioms, none of which would be required if NC had held up.

How does ZA avoid Russell's paradox? One might think at first that it doesn't. After all, if we let A be V – the whole universe of sets – and ϕ be $x \notin x$, a contradiction again appears to arise. But in this case, all the contradiction shows is that V is not a set. All the contradiction shows is that “ V ” is an empty name (i.e., that it has no reference, that V does not exist), since the ontology of Zermelo's system consists solely of sets.

This same point can be made in yet another way, involving a relativized form of Russell's argument. Let B be any set. By ZA, the set $R_B = \{x \in B : x \notin x\}$ exists, but it cannot be an element of B . For if it is an element of B , then we can ask whether or not it is an element of R_B ; and it is if and only if it is not. Thus something, namely R_B , is “missing” from each set B . So again, V is not a set, since nothing can be missing from V . But notice the following subtlety: unlike the previous argument involving the direct application of Aussonderungs to V , the present argument hints at the idea that, while V is not a set, “ V ” is not an empty name. The next strategy for dealing with Russell's paradox capitalizes on this hint.

John von Neumann's (1925) untyped method for dealing with paradoxes, and with Russell's paradox in particular, is simple and ingenious. Von Neumann introduces a distinction between membership and non-membership and, on this basis, draws a distinction between sets and classes. An object is a member (simpliciter) if it is a member of some class; and it is a non-member if it is not a member of any class. (Actually, von Neumann develops a theory of functions, taken as primitive, rather than classes, wherein corresponding to the member/non-member distinction one has a distinction between an object that can be an argument of some function and one that cannot. In its modern form, due to Bernays and Gödel, it is a single-sorted theory of classes.)

Sets are then defined as members, and non-members are labeled "proper classes." So for example, the Russell class, R , cannot be a member of any class, and hence it must be a proper class. If R is assumed to be an element of a class A , then it follows from one of von Neumann's axioms that R is not equivalent to V . But R is equivalent to V , and hence not an element of A . Thus, von Neumann's method is closely related to the result stated above about the set R_B , for arbitrary B . Von Neumann's method, while admired by the likes of Gödel and Bernays, has been undervalued in recent years.

Quine (1937) and (1967) similarly provide another untyped method (in letter if not in spirit) of blocking Russell's paradox, and one that is rife with interesting anomalies. Quine's basic idea is to introduce a stratified comprehension axiom. In effect, the axiom blocks circularity by introducing a hierarchy (or stratification) that is similar to type theory in some ways, and dissimilar in others. (Details can be found in the entry on Quine's New Foundations.)

In contrast to Zermelo's, von Neumann's and Quine's strategies, which are in a sense purely set theoretic, there have also been attempts to avoid Russell's paradox by altering the underlying logic. There have been many such attempts and we shall not review them all, but one stands out as being, at the moment, both radical and somewhat popular (although not with set theorists per se): this is the paraconsistent approach, which limits the overall effect of an isolated

contradiction on an entire theory. Classical logic mandates that any contradiction trivializes a theory by making every sentence of the theory provable. This is because, in classical logic, the following is a theorem:

$$(\text{Ex Falso Quadlibet}) A \supset (\sim A \supset B).$$

Now, virtually the only way to avoid EFQ is to give up disjunctive syllogism, that is, given the usual definitions of the connectives, modus ponens! So altering basic sentential logic in this way is radical indeed – but possible. Unfortunately, even giving up EFQ is not enough to retain a semblance of NC. One also has to give up the following additional theorem of basic sentential logic:

$$(\text{Contraction}) (A \supset (A \supset B)) \supset (A \supset B).$$

It can then be argued that NC leads directly, not merely to an isolated contradiction, but to triviality. (For the argument that this is so, see the entry on Curry's paradox, section 2.2. Note too that it is not enough merely to retain the name "modus ponens"; it is the rule itself that becomes modified within non-traditional logics.) Thus it seems that the woes of NC are not confined to Russell's paradox but also include a negation-free paradox due to Curry.

Another suggestion might be to conclude that the paradox depends upon an instance of the principle of Excluded Middle, that either R is a member of R or it is not. This is a principle that is rejected by some non-classical approaches to logic, including intuitionism. However it is possible to formulate the paradox without appealing to Excluded Middle by relying instead upon the Law of Non-contradiction. We do so as follows: Given the definition of R it follows that $R \in R \equiv \sim(R \in R)$. So $R \in R \supset \sim(R \in R)$. But we also know that $R \in R \supset R \in R$. So $R \in R \supset (R \in R \wedge \sim(R \in R))$. But by the Law of Non-contradiction we know that $\sim(R \in R \wedge \sim(R \in R))$. So by modus tollens we conclude that $\sim(R \in R)$. At the same time we also know that since $R \in R \equiv \sim(R \in R)$, it follows that $\sim(R \in R) \supset R \in R$, and thus that $R \in R$. So we can deduce both $R \in R$ and its negation using only intuitionistically acceptable methods.

It seems, therefore, that proponents of non-classical logics cannot claim to have preserved NC in any significant sense, other than preserving the purely syntactical form of the principle, and neither intuitionism nor paraconsistency plus the abandonment of Contraction will offer an advantage over the untyped solutions of Zermelo, von Neumann, or Quine. (Further discussion can be found in Meyer, Routley and Dunn (1979), Irvine (1992), Priest (2006, ch. 18), Weber (2010), Weber (2012), and in the entries on Curry's paradox. and paraconsistent logic.

It is also worth noting that Russell's paradox was not the only paradox that troubled Russell and, hence, not the only motivation for the type restrictions one finds in Principia Mathematica. In his earlier work, The Principles of Mathematics, Russell devotes a chapter to "the Contradiction" (Russell's paradox), presenting it in several forms and dismissing several non-starter responses. He then signals that he will "shortly" discuss the doctrine of types. This doesn't happen for several hundred pages, until we reach the very end of the book, in Appendix B! There Russell presents an incipient, simple theory of types, not the theory of types we find in Principia Mathematica. Why was the later theory needed? The reason is that in Appendix B Russell also presents another paradox which he thinks cannot be resolved by means of the simple theory of types. This new paradox concerns propositions, not classes, and it, together with the semantic paradoxes, led Russell to formulate his ramified version of the theory of types.

The new, propositional version of the paradox has not figured prominently in the subsequent development of logic and set theory, but it sorely puzzled Russell. For one thing, it seems to contradict Cantor's theorem. Russell writes: "We cannot admit that there are more ranges [classes of propositions] than propositions" (1903, 527). The reason is that there seem to be easy, one to one correlations between classes of propositions and propositions. For example, the class m of propositions can be correlated with the proposition that every proposition in m is true. This, together with a fine-grained principle of individuation for propositions (asserting, for one thing, that if the classes m and n of propositions differ, then any proposition about m will differ from any proposition about n) leads to contradiction.

There has been relatively little discussion of this paradox, although it played a key role in the development of Church's logic of sense and denotation. While we have several set theories to choose from, we do not have anything like a well-developed theory of Russellian propositions, although such propositions are central to the views of Millians and direct-reference theorists. One would think that such a theory would be required for the foundations of semantics, if not for the foundations of mathematics. Thus, while one of Russell's paradoxes has led to the fruitful development of the foundations of mathematics, his "other" paradox has yet to lead to anything remotely similar in the foundations of semantics. To be sure, Church (1974a) and Anderson (1989) have attempted to develop a Russellian intensional logic based on the ramified theory of types, but an argument can be made that the ramified theory is too restrictive to serve as a foundation for the semantics of natural language. There have also been some recent attempts to obtain the beginnings of a Russellian intensional logic based on untyped set theories (Cantini 2004; Deutsch 2014). It is rather ironic that although fine-grained Russellian propositions are favored in the philosophy of language, the formal development of intensional logic is dominated by Montague grammar, with its course-grained theory of propositions.

It is also worth noting that a number of seemingly purely set-theoretical principles are actually (applied) instances of theorems of pure logic (i.e., of first-order quantification theory with identity)! There is a (partial) list of these in Kalish, Montague, and Mar (2000). Russell's paradox is an instance of T269 in this list:

$$(T269) \sim \exists y \forall x (Fxy \equiv \sim Fxx).$$

Reading the dyadic predicate letter "F" as "is a member of", this says that it is not the case that there is a y such that for any x, x is a member of y if and only if x is not a member of x. Does this mean that Russell's paradox reduces to T269?

Certainly the proof of T269 distills the essence of Russell's argument, its pattern of reasoning. But that pattern also underwrites an endless list of seemingly frivolous "paradoxes" such as the famous paradox of the barber who shaves all and only those who do not shave themselves or, similarly, the paradox of the

benevolent but efficient God who helps all and only those who do not help themselves.

How do these “pseudo paradoxes,” as they are sometimes called, differ, if at all, from Russell’s paradox? The pattern of reasoning is the same and the conclusion – that there is no such Barber, no such efficient God, no such set of non-self-membered sets – is the same: such things simply don’t exist. (However, as von Neumann showed, it is not necessary to go quite this far. Von Neumann’s method instructs us not that such things as R do not exist, but just that we cannot say much about them, inasmuch as R and the like cannot fall into the extension of any predicate that qualifies as a class.)

The standard answer to this question is that the difference lies in the subject matter. Quine asks, “why does it [Russell’s paradox] count as an antinomy and the barber paradox not?”; and he answers, “The reason is that there has been in our habits of thought an overwhelming presumption of there being such a class but no presumption of there being such a barber” (1966, 14). Even so, psychological talk of “habits of thought” is not particularly illuminating. More to the point, Russell’s paradox sensibly gives rise to the question of what sets there are; but it is nonsense to wonder, on such grounds as T269, what barbers or Gods there are!

This verdict, however, is not quite fair to fans of the Barber or of T269 generally. They will insist that the question raised by T269 is not what barbers or Gods there are, but rather what non-paradoxical objects there are. This question is virtually the same as that raised by Russell’s paradox itself. Thus, from this perspective, the relation between the Barber and Russell’s paradox is much closer than many (following Quine) have been willing to allow (Salmon 2013).

We note that there is a first-order logical formula that bears the same relation to the principle about the R_B ’s that T269 bears to Russell’s paradox. It is the following:

$$(T273) \forall z \forall y (\forall x [Fxy \equiv (Fxz \wedge \sim Fxx)] \supset \sim Fyz).$$

(We have taken the liberty of extending the numbering used in Kalish, Montague and Mar (2000) to T273.) But not all set-theoretic paradoxes are similarly related to first-order logical theorems. The Burali-Forti paradox is an example, since the notion of a well-ordering is not elementary; that is, it is not first-order definable.

Russell's paradox has never been passé, but recently there has been an explosion of interest in it by scholars involved in research in mathematical logic and in philosophical and historical studies of modern logic. A glance at the contents of the 2004 volume *One Hundred Years of Russell's Paradox* shows prominent mathematical and philosophical logicians and historians of logic pouring over the paradox, proposing new ways back into Cantor's paradise, or other ways of resolving the issue. Their investigations include radically new ways out of the dilemma posed by the paradox, new studies of the theories of types (simple and ramified, and extensions thereof), new interpretations of Russell's paradox and constructive theories, of Russell's paradox of propositions and of his own attempt at an untyped theory (the substitution theory), and so forth.

Russell's Paradox

Russell's paradox is a counterexample to naive set theory, which defines a set as any definable collection. The paradox defines the set RR of all sets that are **not** members of themselves, and notes that

- if RR contains itself, then RR must be a set that is not a member of itself by the definition of RR , which is contradictory;
- if RR does not contain itself, then RR is one of the sets that is not a member of itself, and is thus contained in RR by definition--also a contradiction.

This contradiction is Russell's paradox. It was significant due to reshaping the definitions of set theory, which was of particular interest at the time as the fundamental axioms of mathematics (e.g. the Peano axioms that define arithmetic) were being redefined in the language of sets.

Theory of Computation and Compilers Unit – 8

Russell's paradox (and similar issues) was eventually resolved by an axiomatic set theory called ZFC, after Zermelo, Franekel, and Skolem, which gained widespread acceptance after the axiom of choice was no longer controversial. In short, ZFC's resolved the paradox by defining a set of axioms in which it is not necessarily the case that there is a set of objects satisfying some given property, unlike naive set theory in which any property defines a set of objects satisfying it.

Informal Formulation

Consider a barber who shaves exactly those men who do not shave themselves (i.e. the barber shaves everyone who doesn't shave themselves and shaves nobody else). Then

- if the barber shaves himself, then the barber is an example of "those men who do not shave themselves," a contradiction;
- if the barber does not shave himself, then the barber is an example of "those men who do not shave themselves," and thus the barber shaves himself--also a contradiction.

Hence the barber does not shave himself, but he also does not not shave himself, hence the paradox.

In the above example, an easy resolution is "no such barber exists," but the point of Russell's paradox is that such a "barber" (i.e. a set) must exist if naive set theory were consistent. Since this barber leads to a paradox, naive set theory must be inconsistent.

Motivation and Importance

Set theory was of particular interest just prior to the 20th century, as its language is extremely useful in formalizing general mathematics. For instance, just a few applications are

Theory of Computation and Compilers Unit – 8

- Arithmetic can be formalized using sets as in the Peano axioms. As much of mathematics depends solely on the completeness of arithmetic, this allows for a large amount of mathematics to be implicitly formalized.
- Bijection can be understood as a statement about one-to-one correspondence of sets, which also leads to...
- Formalization of infinity and cardinality, especially in defining different types of infinity such as Cantor's proof that there are more real numbers than there are integers.

As a result of this incredibly useful formalization, much of mathematics was repurposed to be defined in terms of Cantorian set theory, to the point that it (literally) formed the foundation of mathematics.

Russell's paradox served to show that Cantorian set theory led to contradictions, meaning not only that set theory had to be rethought, but most of mathematics (due to resting on set theory) was technically in doubt. Fortunately, the field was repaired a short time later by new axioms (ZFC), and set theory remains the main foundational system of mathematics today.

Formal Definitions and Formulation

Naive set theory is the theory of predicate logic with binary predicate $\in$, that satisfies

$$\exists y \forall x (x \in y \Leftrightarrow \varphi(x))$$

for any predicate φ . This is called **unrestricted comprehension**, and means

There exists a set y whose members are exactly the objects satisfying the predicate φ .

Naive set theory also contains two other axioms (which ZFC also contains):

Existential instantiation:

Theory of Computation and Compilers Unit – 8

Given a formula of the form $(\exists x)\varphi(x)$, one can infer $\varphi(c)$ for some new symbol c .

All this is saying is that if there exists some object satisfying a given property, that element can be given a name c (in such a way that c was not previously used).

For instance,

- There exists a number satisfying the equation $x^2 + 1 = 0$.
- Define i to be a number satisfying the equation $i^2 + 1 = 0$.

is valid logic.

Universal instantiation:

Given a formula of the form $\forall x\varphi(x)$, one can infer $\varphi(c)$ for any c in the universe.

Intuitively speaking, this axiom states that if everything satisfies some property, any one of those things also satisfies that property. For instance,

- All people living in California live in the U.S.A. (hypothesis)
- John lives in California. (implying that John is part of the universe)
- John lives in the U.S.A. (invocation of universal instantiation) is valid logic.

These axioms are sufficient to illustrate Russell's paradox:

- Consider the predicate $\varphi: x \ni x$.
- By unrestricted comprehension, there exists a set y consisting of elements satisfying the predicate φ .
- By existential instantiation, there exists a z such $\forall x(x \in z \Leftrightarrow \varphi(x))$.
- By universal instantiation, z (which is in the universe) satisfies the predicate $x \in z \Leftrightarrow \varphi(x)$, so $z \in z \Leftrightarrow z \notin z$,

which is a contradiction, implying that naive set theory is inconsistent.

Resolving the Paradox

Theory of Computation and Compilers Unit – 8

Roughly speaking, there are two ways to resolve Russell's paradox: either to

- alter the logical language, i.e. first order logic, that the axioms of set theory are expressed in, or
- alter the axioms of set theory, while retaining the logical language they are expressed in.

Russell took the first approach in his attempt at redefining set theory with Whitehead in Principia Mathematica, developing type theory in the process. However, though they eventually succeeded in defining arithmetic in such a fashion, they were unable to do so using pure logic, and so other problems arose.

In fact, Godel showed that Peano arithmetic is incomplete (assuming Peano arithmetic is consistent), essentially showing that Russell's approach was impossible to formalize. In doing so, Godel demonstrated his acclaimed incompleteness theorems.

The second approach, in which the axioms of set theory are altered, was favored by Zermelo (later joined by Fraenkel and Skolem) in his derivation of ZFC. This resolves the paradox by replacing unrestricted comprehension with **restricted comprehension** (also called **specification**):

Given a predicate ϕ with free variables in $x, z, w_1, w_2, \dots, w_n$,

$$\forall z \forall w_1 \forall w_2 \dots \forall w_n \exists y \forall x (x \in y \iff (x \in z \wedge \phi)).$$

Essentially, this means that given a set z and a predicate ϕ , the subset

$$\{x \in z : \phi(x)\}$$

(i.e. all elements of z satisfying the predicate ϕ) exists.

This resolves Russell's paradox as only subsets can be constructed, rather than any set expressible in the form $\{x:\phi(x)\}$. In this sense, Russell's paradox serves to show that

There does not exist a set containing all sets,
which is also a useful result in its own right.

Regular Language

A regular language is a language that can be expressed with a regular expression or a deterministic or non-deterministic finite automata or state machine. A language is a set of strings which are made up of characters from a specified alphabet, or set of symbols. Regular languages are a subset of the set of all strings. Regular languages are used in parsing and designing programming languages and are one of the first concepts taught in computability courses. These are useful for helping computer scientists to recognize patterns in data and group certain computational problems together — once they do that, they can take similar approaches to solve the problems grouped together. Regular languages are a key topic in computability theory.

A language is regular if it can be expressed in terms of regular expression.

Regular Expressions

Regular Expressions are used to denote regular languages. An expression is regular if:

- ϕ is a regular expression for regular language ϕ .
- ϵ is a regular expression for regular language $\{\epsilon\}$.
- If $a \in \Sigma$ (Σ represents the input alphabet), a is regular expression with language $\{a\}$.

Theory of Computation and Compilers Unit – 8

- If a and b are regular expression, $a + b$ is also a regular expression with language $\{a,b\}$.
- If a and b are regular expression, ab (concatenation of a and b) is also regular.
- If a is regular expression, a^* (0 or more times a) is also regular.

	REGULAR EXPRESSION	REGULAR LANGUAGES
set of vowels	$(a \cup e \cup i \cup o \cup u)$	$\{a, e, i, o, u\}$
a followed by 0 or more b	$(a.b^*)$	$\{a, ab, abb, abbb, abbbb, \dots\}$
any no. of vowels followed by any no. of consonants	$v^*.c^*$ (where v – vowels and c – consonants)	$\{\epsilon, a, aou, aiou, b, abcd, \dots\}$ where ϵ represent empty string (in case 0 vowels and 0 consonants)

Regular Grammar : A grammar is regular if it has rules of form $A \rightarrow a$ or $A \rightarrow aB$ or $A \rightarrow \epsilon$ where ϵ is a special symbol called NULL.

Closure Properties of Regular Languages

Union : If L_1 and L_2 are two regular languages, their union $L_1 \cup L_2$ will also be regular. For example, $L_1 = \{a^n \mid n \geq 0\}$ and $L_2 = \{b^n \mid n \geq 0\}$

$L_3 = L_1 \cup L_2 = \{a^n \cup b^n \mid n \geq 0\}$ is also regular.

Intersection : If L_1 and L_2 are two regular languages, their intersection $L_1 \cap L_2$ will also be regular. For example,

Theory of Computation and Compilers Unit – 8

$L_1 = \{a^m b^n \mid n \geq 0 \text{ and } m \geq 0\}$ and $L_2 = \{a^m b^n \cup b^n a^m \mid n \geq 0 \text{ and } m \geq 0\}$

$L_3 = L_1 \cap L_2 = \{a^m b^n \mid n \geq 0 \text{ and } m \geq 0\}$ is also regular.

Concatenation : If L_1 and L_2 are two regular languages, their concatenation $L_1.L_2$ will also be regular. For example,

$L_1 = \{a^n \mid n \geq 0\}$ and $L_2 = \{b^n \mid n \geq 0\}$

$L_3 = L_1.L_2 = \{a^m . b^n \mid m \geq 0 \text{ and } n \geq 0\}$ is also regular.

Kleene Closure : If L_1 is a regular language, its Kleene closure L_1^* will also be regular. For example,

$L_1 = (a \cup b)$

$L_1^* = (a \cup b)^*$

Complement : If $L(G)$ is regular language, its complement $L'(G)$ will also be regular. Complement of a language can be found by subtracting strings which are in $L(G)$ from all possible strings. For example,

$L(G) = \{a^n \mid n > 3\}$

$L'(G) = \{a^n \mid n \leq 3\}$

Note : Two regular expressions are equivalent if languages generated by them are same. For example, $(a+b^*)^*$ and $(a+b)^*$ generate same language. Every string which is generated by $(a+b^*)^*$ is also generated by $(a+b)^*$ and vice versa.

How to solve problems on regular expression and regular languages?

Question 1 : Which one of the following languages over the alphabet $\{0,1\}$ is described by the regular expression?

$(0+1)^*0(0+1)^*0(0+1)^*$

- (A) The set of all strings containing the substring 00.
- (B) The set of all strings containing at most two 0's.
- (C) The set of all strings containing at least two 0's.
- (D) The set of all strings that begin and end with either 0 or 1.

Solution : Option A says that it must have substring 00. But 10101 is also a part of language but it does not contain 00 as substring. So it is not correct option.

Option B says that it can have maximum two 0's but 00000 is also a part of

Theory of Computation and Compilers Unit – 8

language. So it is not correct option.

Option C says that it must contain atleast two 0. In regular expression, two 0 are present. So this is correct option.

Option D says that it contains all strings that begin and end with either 0 or 1. But it can generate strings which start with 0 and end with 1 or vice versa as well. So it is not correct.

Question 2 : Which of the following languages is generated by given grammar?

$S \rightarrow aS \mid bS \mid \epsilon$

(A) $\{a^n b^m \mid n, m \geq 0\}$

(B) $\{w \in \{a, b\}^* \mid w \text{ has equal number of } a\text{'s and } b\text{'s}\}$

(C) $\{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\}$

(D) $\{a, b\}^*$

Solution : Option (A) says that it will have 0 or more a followed by 0 or more b.

But $S \rightarrow bS \Rightarrow baS \Rightarrow ba$ is also a part of language. So (A) is not correct.

Option (B) says that it will have equal no. of a's and b's. But $S \rightarrow bS \Rightarrow b$ is also a part of language. So (B) is not correct.

Option (C) says either it will have 0 or more a's or 0 or more b's or a's followed by b's. But as shown in option (A), ba is also part of language. So (C) is not correct.

Option (D) says it can have any number of a's and any numbers of b's in any order. So (D) is correct.

Question 3 : The regular expression $0^*(10^*)^*$ denotes the same set as

(A) $(1^*0)^*1^*$

(B) $0 + (0 + 10)^*$

(C) $(0 + 1)^* 10(0 + 1)^*$

(D) none of these

Solution : Two regular expressions are equivalent if languages generated by them are same.

Option (A) can generate all strings generated by $0^*(10^*)^*$. So they are equivalent.

Theory of Computation and Compilers Unit – 8

Option (B) string null can not generated by given languages but $0^*(10^*)^*$ can. So they are not equivalent.

Option (C) will have 10 as substring but $0^*(10^*)^*$ may or may not. So they are not equivalent.

Question 4 : The regular expression for the language having input alphabets a and b, in which two a's do not come together:

- (A) $(b + ab)^* + (b + ab)^*a$
- (B) $a(b + ba)^* + (b + ba)^*$
- (C) both options (A) and (B)
- (D) none of the above

Solution:

Option (C) stating both both options (A) and (B) is the correct regular expression for the stated question.

The language in the question can be expressed as
 $L = \{\epsilon, a, b, bb, ab, aba, ba, bab, baba, abab, \dots\}$.

In option (A) 'ab' is considered the building block for finding out the required regular expression. $(b + ab)^*$ covers all cases of strings generated ending with 'b'. $(b + ab)^*a$ covers all cases of strings generated ending with a.

Applying similar logic for option (B) we can see that the regular expression is derived considering 'ba' as the building block and it covers all cases of strings starting with a and starting with b.

Deterministic Finite Automaton

Finite Automaton can be classified into two types –

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NDFA / NFA)

Deterministic Finite Automaton (DFA)

Theory of Computation and Compilers Unit – 8

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton**.

Formal Definition of a DFA

A DFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where –

- Q is a finite set of states.
- Σ is a finite set of symbols called the alphabet.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow Q$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of a DFA

A DFA is represented by digraphs called **state diagram**.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

Let a deterministic finite automaton be $\rightarrow$

- $Q = \{a, b, c\}$,
- $\Sigma = \{0, 1\}$,
- $q_0 = \{a\}$,

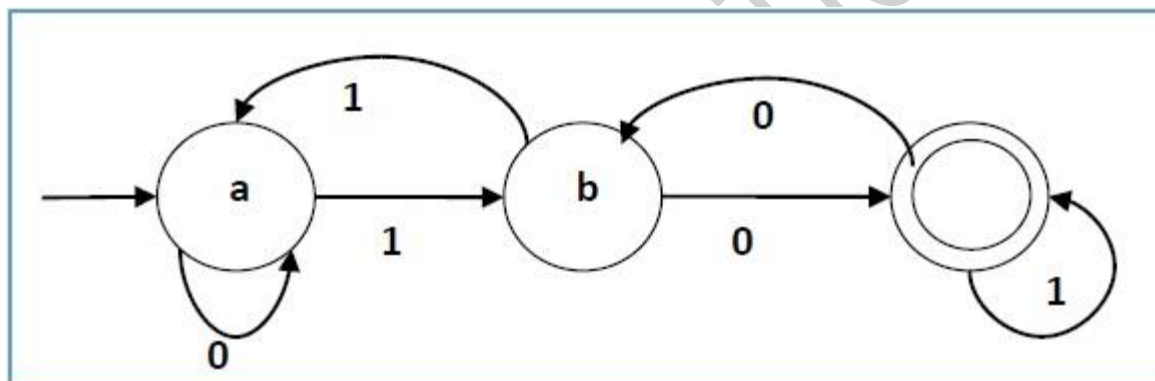
Theory of Computation and Compilers Unit – 8

- $F = \{c\}$, and

Transition function δ as shown by the following table –

Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c

Its graphical representation would be as follows –



Non-deterministic Finite Automaton (NDFA)

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where –

- Q is a finite set of states.

Theory of Computation and Compilers Unit – 8

- Σ is a finite set of symbols called the alphabets.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow 2^Q$
(Here the power set of Q (2^Q) has been taken because in case of N DFA, from a state, transition can occur to any combination of Q states)
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of an N DFA: (same as DFA)

An N DFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

Let a non-deterministic finite automaton be $\rightarrow$

- $Q = \{a, b, c\}$
- $\Sigma = \{0, 1\}$
- $q_0 = \{a\}$
- $F = \{c\}$

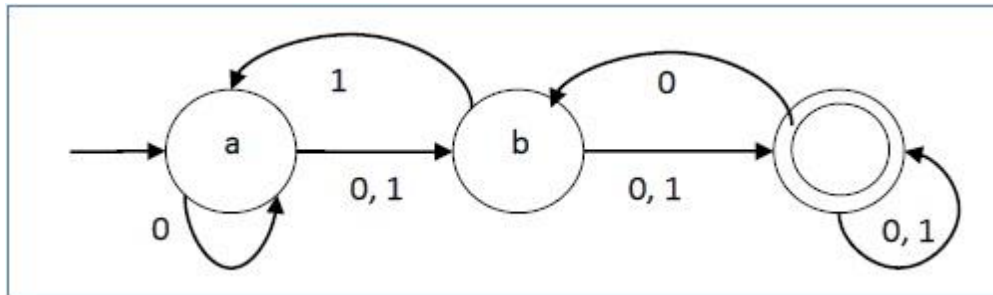
The transition function δ as shown below –

Present State	Next State for Input 0	Next State for Input 1
a	a, b	b
b	c	a, c

Theory of Computation and Compilers Unit – 8

c	b, c	c
---	------	---

Its graphical representation would be as follows –



DFA vs NDFA

The following table lists the differences between DFA and NDFA.

DFA	NDFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called deterministic.	The transition from a state can be to multiple next states for each input symbol. Hence it is called non-deterministic.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

Acceptors, Classifiers, and Transducers

Accepter (Recognizer)

An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

Classifier

A **classifier** has more than two final states and it gives a single output when it terminates.

Transducer

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types –

- **Mealy Machine** – The output depends both on the current state and the current input.
- **Moore Machine** – The output depends only on the current state.

Acceptability by DFA and NDFA

A string is accepted by a DFA/NDFA iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

A string S is accepted by a DFA/NDFA $(Q, \Sigma, \delta, q_0, F)$, iff

$$\delta^*(q_0, S) \in F$$

The language L accepted by DFA/NDFA is

$$\{S \mid S \in \Sigma^* \text{ and } \delta^*(q_0, S) \in F\}$$

A string S' is not accepted by a DFA/NDFA $(Q, \Sigma, \delta, q_0, F)$, iff

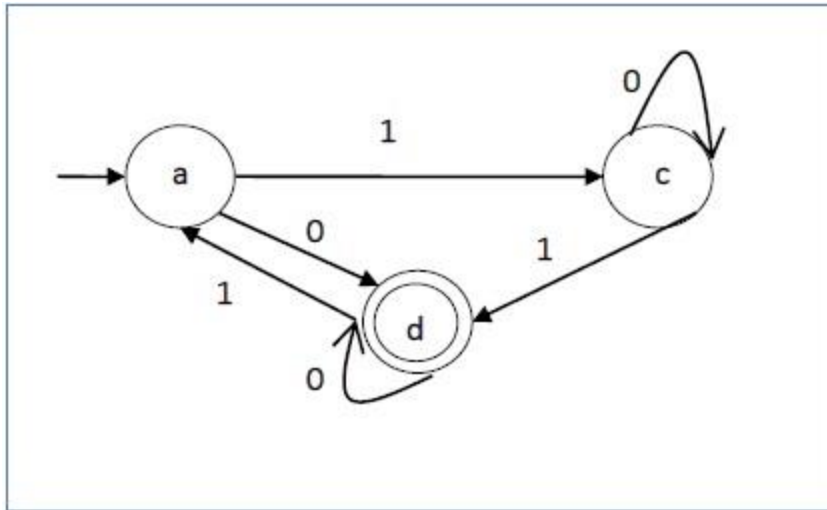
$$\delta^*(q_0, S') \notin F$$

The language L' not accepted by DFA/NDFA (Complement of accepted language L) is

$$\{S \mid S \in \Sigma^* \text{ and } \delta^*(q_0, S) \notin F\}$$

Example

Let us consider the DFA shown in Figure 1.3. From the DFA, the acceptable strings can be derived.



Strings accepted by the above DFA: {0, 00, 11, 010, 101, ...}

Strings not accepted by the above DFA: {1, 011, 111, ...}

NDFA to DFA Conversion

Problem Statement

Let $X = (Q_x, \Sigma, \delta_x, q_0, F_x)$ be an NDFA which accepts the language $L(X)$. We have to design an equivalent DFA $Y = (Q_y, \Sigma, \delta_y, q_0, F_y)$ such that $L(Y) = L(X)$. The following procedure converts the NDFA to its equivalent DFA –

Algorithm

Input – An NDFA

Output – An equivalent DFA

Step 1 – Create state table from the given NDFA.

Step 2 – Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 – Mark the start state of the DFA by q_0 (Same as the NDFA).

Step 4 – Find out the combination of States $\{Q_0, Q_1, \dots, Q_n\}$ for each possible input alphabet.

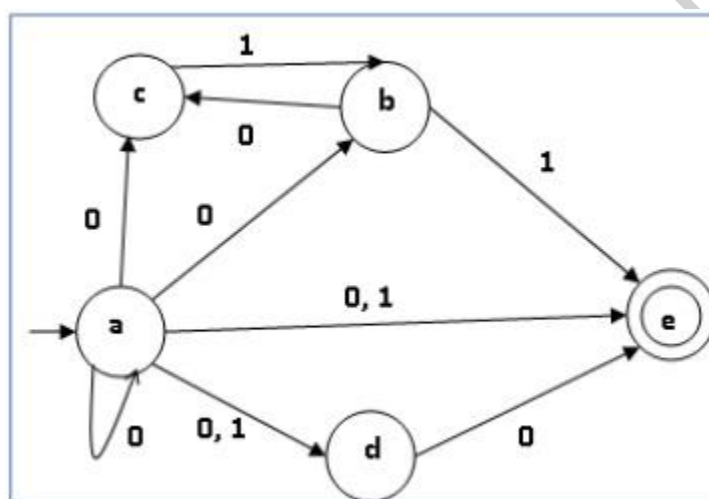
Theory of Computation and Compilers Unit – 8

Step 5 – Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 – The states which contain any of the final states of the NFA are the final states of the equivalent DFA.

Example

consider the NFA shown in the figure below.



q	$\delta(q,0)$	$\delta(q,1)$
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	$\emptyset$	{b}
d	{e}	$\emptyset$

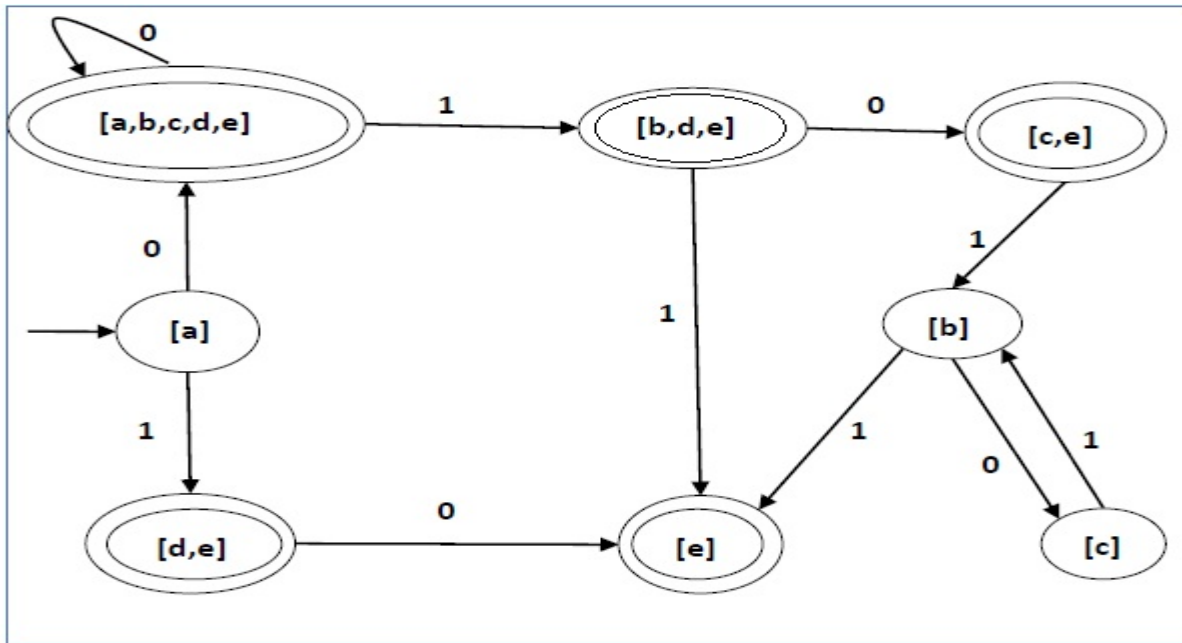
Theory of Computation and Compilers Unit – 8

e	$\emptyset$	$\emptyset$
---	-------------	-------------

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

q	$\delta(q,0)$	$\delta(q,1)$
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	$\emptyset$
[b,d,e]	[c,e]	[e]
[e]	$\emptyset$	$\emptyset$
[c, e]	$\emptyset$	[b]
[b]	[c]	[e]
[c]	$\emptyset$	[b]

The state diagram of the DFA is as follows –



Formal grammar

- Formal grammar is a set of rules. It is used to identify correct or incorrect strings of tokens in a language. The formal grammar is represented as G.
- Formal grammar is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- Formal grammar is used mostly in the syntactic analysis phase (parsing) particularly during the compilation.

Formal grammar G is written as follows:

- $G = \langle V, N, P, S \rangle$

Where:

N describes a finite set of non-terminal symbols.

V describes a finite set of terminal symbols.

P describes a set of production rules

S is the start symbol.

Example:

Theory of Computation and Compilers Unit – 8

1. $L = \{a, b\}$, $N = \{S, R, B\}$

Production rules:

1. $S = bR$
2. $R = aR$
3. $R = aB$
4. $B = b$

Through this production we can produce some strings like: bab, baab, baaab etc.

This production describes the string of shape ba^nab .

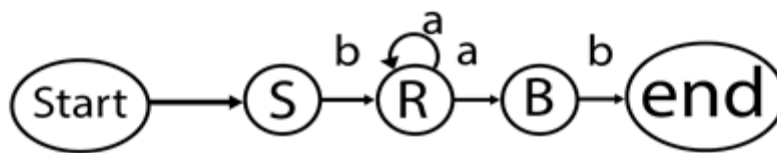


Fig: Formal grammar

Regular expression

- Regular expression is a sequence of pattern that defines a string. It is used to denote regular languages.
- It is also used to match character combinations in strings. String searching algorithm used this pattern to find the operations on string.
- In regular expression, x^* means zero or more occurrence of x . It can generate $\{e, x, xx, xxx, xxxx, \dots\}$
- In regular expression, x^+ means one or more occurrence of x . It can generate $\{x, xx, xxx, xxxx, \dots\}$

Operations on Regular Language

Theory of Computation and Compilers Unit – 8

The various operations on regular language are:

Union: If L and M are two regular languages then their union $L \cup M$ is also a union.

$$1. L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$$

Intersection: If L and M are two regular languages then their intersection is also an intersection.

$$1. L \cap M = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$$

Kleene closure: If L is a regular language then its kleene closure L^* will also be a regular language.

$$1. L^* = \text{Zero or more occurrence of language } L.$$

Example

Write the regular expression for the language:

$$L = \{ab^n w : n \geq 3, w \in (a,b)^+\}$$

Solution:

The string of language L starts with "a" followed by atleast three b's. It contains atleast one "a" or one "b" that is string are like abbba, abbbbbbba, abbbbbbbb, abbbb.....a

So regular expression is:

$$r = ab^3b^*(a+b)^+$$

Here + is a positive closure i.e. $(a+b)^+ = (a+b)^* - \epsilon$

Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Theory of Computation and Compilers Unit – 8

Context free grammar G can be defined by four tuples as:

1. $G = (V, T, P, S)$

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

$$L = \{wcw^R \mid w \in (a, b)^*\}$$

Production rules:

1. $S \rightarrow aSa$
2. $S \rightarrow bSb$
3. $S \rightarrow c$

Now check that abbcbbba string can be derived from the given CFG.

1. $S \Rightarrow aSa$
2. $S \Rightarrow abSba$
3. $S \Rightarrow abbSbba$
4. $S \Rightarrow abbcbbba$

By applying the production $S \rightarrow aSa$, $S \rightarrow bSb$ recursively and finally applying the production $S \rightarrow c$, we get the string $abbcbba$.

Pumping Lemma For Regular Grammars

Theorem

Let L be a regular language. Then there exists a constant ' c ' such that for every string w in L –

$$|w| \geq c$$

We can break w into three strings, $w = xyz$, such that –

- $|y| > 0$
- $|xy| \leq c$
- For all $k \geq 0$, the string xy^kz is also in L .

Applications of Pumping Lemma

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

- If L is regular, it satisfies Pumping Lemma.
- If L does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

- At first, we have to assume that L is regular.
- So, the pumping lemma should hold for L .
- Use the pumping lemma to obtain a contradiction –
 - Select w such that $|w| \geq c$
 - Select y such that $|y| \geq 1$

Theory of Computation and Compilers Unit – 8

- Select x such that $|xy| \leq c$
- Assign the remaining string to z .
- Select k such that the resulting string is not in L .

Hence L is not regular.

Problem

Prove that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

Solution –

- At first, we assume that L is regular and n is the number of states.
- Let $w = a^n b^n$. Thus $|w| = 2n \geq n$.
- By pumping lemma, let $w = xyz$, where $|xy| \leq n$.
- Let $x = a^p$, $y = a^q$, and $z = a^r b^n$, where $p + q + r = n$, $p \neq 0$, $q \neq 0$, $r \neq 0$. Thus $|y| \neq 0$.
- Let $k = 2$. Then $xy^2z = a^p a^{2q} a^r b^n$.
- Number of a s = $(p + 2q + r) = (p + q + r) + q = n + q$
- Hence, $xy^2z = a^{n+q} b^n$. Since $q \neq 0$, xy^2z is not of the form $a^n b^n$.
- Thus, xy^2z is not in L . Hence L is not regular.

Pumping Lemma for CFG

Lemma

If L is a context-free language, there is a pumping length p such that any string $w \in L$ of length $\geq p$ can be written as $w = uvxyz$, where $vy \neq \epsilon$, $|vxy| \leq p$, and for all $i \geq 0$, $uv^i xy^i z \in L$.

Applications of Pumping Lemma

Theory of Computation and Compilers Unit – 8

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem

Find out whether the language $L = \{x^n y^n z^n \mid n \geq 1\}$ is context free or not.

Solution

Let L is context free. Then, L must satisfy pumping lemma.

At first, choose a number n of the pumping lemma. Then, take z as $0^n 1^n 2^n$.

Break z into $uvwxy$, where

$$|vwx| \leq n \text{ and } vx \neq \epsilon.$$

Hence vwx cannot involve both 0s and 2s, since the last 0 and the first 2 are at least $(n+1)$ positions apart. There are two cases –

Case 1 – vwx has no 2s. Then vx has only 0s and 1s. Then uwv , which would have to be in L , has n 2s, but fewer than n 0s or 1s.

Case 2 – vwx has no 0s.

Here contradiction occurs.

Hence, L is not a context-free language.

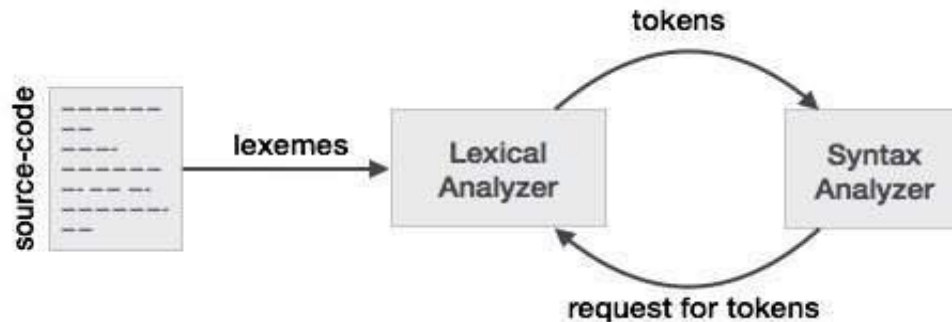
Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from

Theory of Computation and Compilers Unit – 8

the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Specifications of Tokens

how the language theory undertakes the following terms:

Alphabets

Theory of Computation and Compilers Unit – 8

Any finite set of symbols $\{0,1\}$ is a set of binary alphabets, $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ is a set of Hexadecimal alphabets, $\{a-z, A-Z\}$ is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by $|\text{tutorialspoint}| = 14$. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ϵ (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, , , !

Theory of Computation and Compilers Unit – 8

Shift Operator	>>, >>>, <<, <<<
----------------	------------------

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

int intValue;

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword int or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Context-free languages are **closed** under –

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n, n > 0 \}$. Corresponding grammar G_1 will have P: $S_1 \rightarrow aAb | ab$

Let $L_2 = \{ c^m d^m, m \geq 0 \}$. Corresponding grammar G_2 will have P: $S_2 \rightarrow cBb | \epsilon$

Union of L_1 and L_2 , $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S_1 | S_2$

Concatenation

If L_1 and L_2 are context free languages, then $L_1 L_2$ is also context free.

Example

Union of the languages L_1 and L_2 , $L = L_1 L_2 = \{ a^n b^n c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S_1 S_2$

Kleene Star

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

Example

Let $L = \{ a^n b^n, n \geq 0 \}$. Corresponding grammar G will have P: $S \rightarrow aAb | \epsilon$

Kleene Star $L_1 = \{ a^n b^n \}^*$

The corresponding grammar G_1 will have additional productions $S_1 \rightarrow SS_1 | \epsilon$

Context-free languages are **not closed** under –

- **Intersection** – If L_1 and L_2 are context free languages, then $L_1 \cap L_2$ is not necessarily context free.

Theory of Computation and Compilers Unit – 8

- **Intersection with Regular Language** – If L_1 is a regular language and L_2 is a context free language, then $L_1 \cap L_2$ is a context free language.
- **Complement** – If L_1 is a context free language, then L_1' may not be context free.

Pushdown Automata

Basic Structure of PDA

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is –

"Finite state machine" + "a stack"

A pushdown automaton has three components –

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

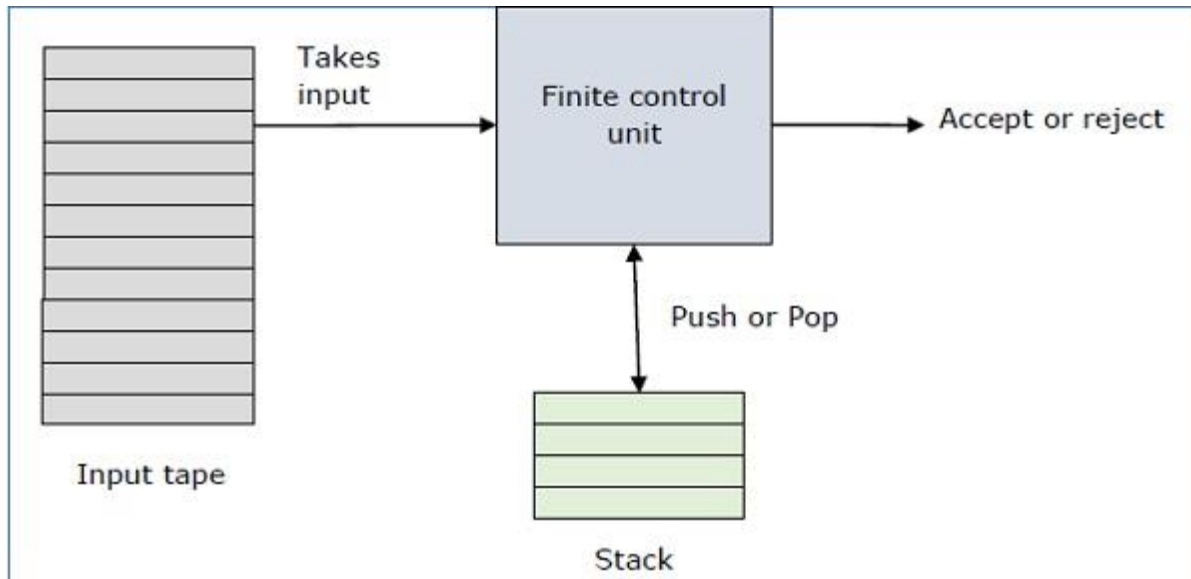
The stack head scans the top symbol of the stack.

A stack does two operations –

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

Theory of Computation and Compilers Unit – 8

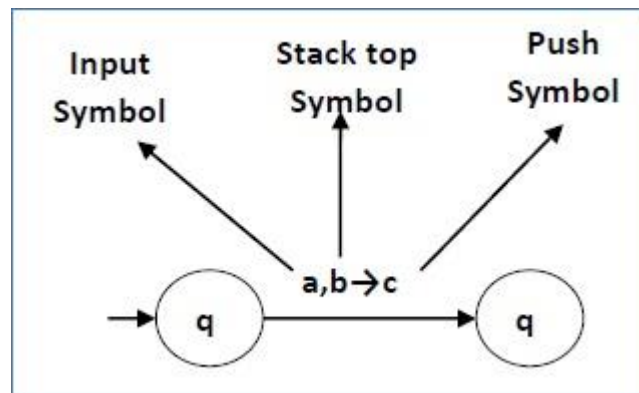
A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.



A PDA can be formally described as a 7-tuple $(Q, \Sigma, S, \delta, q_0, I, F)$ –

- **Q** is the finite number of states
- **Σ** is input alphabet
- **S** is stack symbols
- **δ** is the transition function: $Q \times (\Sigma \cup \{\epsilon\}) \times S \times Q \times S^*$
- **q_0** is the initial state ($q_0 \in Q$)
- **I** is the initial stack top symbol ($I \in S$)
- **F** is a set of accepting states ($F \subseteq Q$)

The following diagram shows a transition in a PDA from a state q_1 to state q_2 , labeled as $a, b \rightarrow c$ –



This means at state 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 q_2 .

Terminologies Related to PDA

Instantaneous Description

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- q is the state
- w is unconsumed input
- s is the stack contents

Turnstile Notation

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol " $\vdash$ ".

Consider a PDA $(Q, \Sigma, S, \delta, q_0, I, F)$. A transition can be mathematically represented by the following turnstile notation –

$$(p, aw, T\beta) \vdash (q, w, \alpha\beta)$$

This implies that while taking a transition from state p to state q , the input symbol 'a' is consumed, and the top of the stack 'T' is replaced by a new string ' α '.

Note – If we want zero or more moves of a PDA, we have to use the symbol ($\vdash^*$) for it.

Pushdown Automata Acceptance

There are two different ways to define PDA acceptability.

Final State Acceptability

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA $(Q, \Sigma, S, \delta, q_0, I, F)$, the language accepted by the set of final states F is –

$$L(\text{PDA}) = \{w \mid (q_0, w, I) \vdash^* (q, \epsilon, x), q \in F\}$$

for any input stack string x .

Empty Stack Acceptability

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

For a PDA $(Q, \Sigma, S, \delta, q_0, I, F)$, the language accepted by the empty stack is –

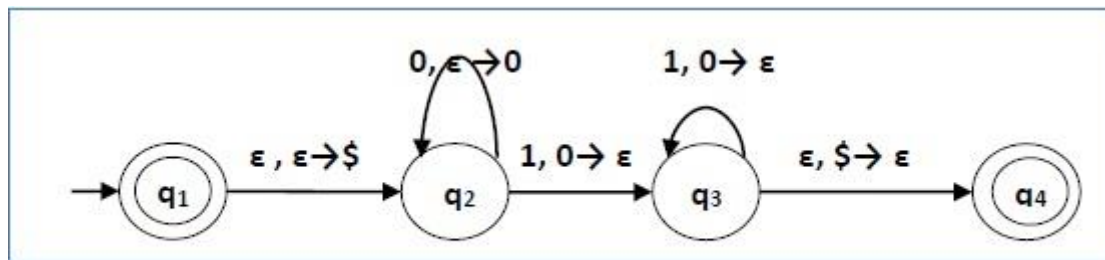
$$L(\text{PDA}) = \{w \mid (q_0, w, I) \vdash^* (q, \epsilon, \epsilon), q \in Q\}$$

Example

Construct a PDA that accepts $L = \{0^n 1^n \mid n \geq 0\}$

Solution

Theory of Computation and Compilers Unit – 8



PDA for $L = \{0^n 1^n \mid n \geq 0\}$

This language accepts $L = \{\epsilon, 01, 0011, 000111, \dots\}$

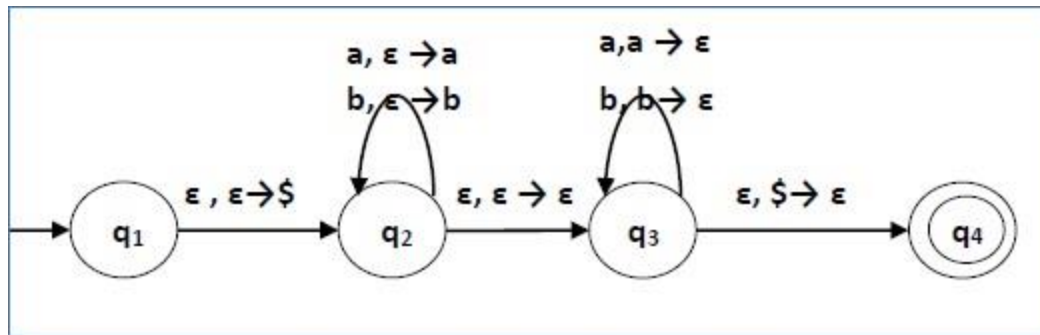
Here, in this example, the number of 'a' and 'b' have to be same.

- Initially we put a special symbol '\$' into the empty stack.
- Then at state q_2 , if we encounter input 0 and top is Null, we push 0 into stack. This may iterate. And if we encounter input 1 and top is 0, we pop this 0.
- Then at state q_3 , if we encounter input 1 and top is 0, we pop this 0. This may also iterate. And if we encounter input 1 and top is 0, we pop the top element.
- If the special symbol '\$' is encountered at top of the stack, it is popped out and it finally goes to the accepting state q_4 .

Example

Construct a PDA that accepts $L = \{ww^R \mid w = (a+b)^*\}$

Solution



PDA for $L = \{ww^R \mid w = (a+b)^*\}$

Initially we put a special symbol '\$' into the empty stack. At state q_2 , the w is being read. In state q_3 , each 0 or 1 is popped when it matches the input. If any other input is given, the PDA will go to a dead state. When we reach that special symbol '\$', we go to the accepting state q_4 .

Simplification of CFGs

In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called **simplification of CFGs**. Simplification essentially comprises of the following steps –

- Reduction of CFG
- Removal of Unit Productions
- Removal of Null Productions

Reduction of CFG

CFGs are reduced in two phases –

Phase 1 – Derivation of an equivalent grammar, G' , from the CFG, G , such that each variable derives some terminal string.

Derivation Procedure –

Step 1 – Include all symbols, W_1 , that derive some terminal and initialize $i=1$.

Step 2 – Include all symbols, W_{i+1} , that derive W_i .

Step 3 – Increment i and repeat Step 2, until $W_{i+1} = W_i$.

Step 4 – Include all production rules that have W_i in it.

Phase 2 – Derivation of an equivalent grammar, G'' , from the CFG, G' , such that each symbol appears in a sentential form.

Derivation Procedure –

Step 1 – Include the start symbol in Y_1 and initialize $i = 1$.

Step 2 – Include all symbols, Y_{i+1} , that can be derived from Y_i and include all production rules that have been applied.

Step 3 – Increment i and repeat Step 2, until $Y_{i+1} = Y_i$.

Problem

Find a reduced grammar equivalent to the grammar G , having production rules, P :
 $S \rightarrow AC \mid B, A \rightarrow a, C \rightarrow c \mid BC, E \rightarrow aA \mid e$

Solution

Phase 1 –

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

$W_1 = \{ A, C, E \}$ from rules $A \rightarrow a, C \rightarrow c$ and $E \rightarrow aA$

$W_2 = \{ A, C, E \} \cup \{ S \}$ from rule $S \rightarrow AC$

$W_3 = \{ A, C, E, S \} \cup \emptyset$

Since $W_2 = W_3$, we can derive G' as –

Theory of Computation and Compilers Unit – 8

$$G' = \{ \{ A, C, E, S \}, \{ a, c, e \}, P, \{ S \} \}$$

where $P: S \rightarrow AC, A \rightarrow a, C \rightarrow c, E \rightarrow aA \mid e$

Phase 2 –

$$Y_1 = \{ S \}$$

$$Y_2 = \{ S, A, C \} \text{ from rule } S \rightarrow AC$$

$$Y_3 = \{ S, A, C, a, c \} \text{ from rules } A \rightarrow a \text{ and } C \rightarrow c$$

$$Y_4 = \{ S, A, C, a, c \}$$

Since $Y_3 = Y_4$, we can derive G'' as –

$$G'' = \{ \{ A, C, S \}, \{ a, c \}, P, \{ S \} \}$$

where $P: S \rightarrow AC, A \rightarrow a, C \rightarrow c$

Removal of Unit Productions

Any production rule in the form $A \rightarrow B$ where $A, B \in \text{Non-terminal}$ is called **unit production**..

Removal Procedure –

Step 1 – To remove $A \rightarrow B$, add production $A \rightarrow x$ to the grammar rule whenever $B \rightarrow x$ occurs in the grammar. [$x \in \text{Terminal}$, x can be Null]

Step 2 – Delete $A \rightarrow B$ from the grammar.

Step 3 – Repeat from step 1 until all unit productions are removed.

Problem

Remove unit production from the following –

$$S \rightarrow XY, X \rightarrow a, Y \rightarrow Z \mid b, Z \rightarrow M, M \rightarrow N, N \rightarrow a$$

Solution –

Theory of Computation and Compilers Unit – 8

There are 3 unit productions in the grammar –

$Y \rightarrow Z$, $Z \rightarrow M$, and $M \rightarrow N$

At first, we will remove $M \rightarrow N$.

As $N \rightarrow a$, we add $M \rightarrow a$, and $M \rightarrow N$ is removed.

The production set becomes

$S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow Z \mid b$, $Z \rightarrow M$, $M \rightarrow a$, $N \rightarrow a$

Now we will remove $Z \rightarrow M$.

As $M \rightarrow a$, we add $Z \rightarrow a$, and $Z \rightarrow M$ is removed.

The production set becomes

$S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow Z \mid b$, $Z \rightarrow a$, $M \rightarrow a$, $N \rightarrow a$

Now we will remove $Y \rightarrow Z$.

As $Z \rightarrow a$, we add $Y \rightarrow a$, and $Y \rightarrow Z$ is removed.

The production set becomes

$S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow a \mid b$, $Z \rightarrow a$, $M \rightarrow a$, $N \rightarrow a$

Now Z , M , and N are unreachable, hence we can remove those.

The final CFG is unit production free –

$S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow a \mid b$

Removal of Null Productions

In a CFG, a non-terminal symbol '**A**' is a nullable variable if there is a production **A** $\rightarrow \epsilon$ or there is a derivation that starts at **A** and finally ends up with

ϵ : $A \rightarrow \dots \rightarrow \epsilon$

Removal Procedure

Step 1 – Find out nullable non-terminal variables which derive ϵ .

Step 2 – For each production $A \rightarrow a$, construct all productions $A \rightarrow x$ where x is obtained from 'a' by removing one or multiple non-terminals from Step 1.

Step 3 – Combine the original productions with the result of step 2 and remove ϵ - productions.

Problem

Remove null production from the following –

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

Solution –

There are two nullable variables – **A** and **B**

At first, we will remove $B \rightarrow \epsilon$.

After removing $B \rightarrow \epsilon$, the production set becomes –

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

Now we will remove $A \rightarrow \epsilon$.

After removing $A \rightarrow \epsilon$, the production set becomes –

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

This is the final production set without null transition.

PDA & Context-Free Grammar

If a grammar **G** is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar **G**. A parser can be built for the grammar **G**.

Theory of Computation and Compilers Unit – 8

Also, if **P** is a pushdown automaton, an equivalent context-free grammar **G** can be constructed where

$$L(G) = L(P)$$

In the next two topics, we will discuss how to convert from PDA to CFG and vice versa.

Algorithm to find PDA corresponding to a given CFG

Input – A CFG, $G = (V, T, P, S)$

Output – Equivalent PDA, $P = (Q, \Sigma, S, \delta, q_0, I, F)$

Step 1 – Convert the productions of the CFG into GNF.

Step 2 – The PDA will have only one state $\{q\}$.

Step 3 – The start symbol of CFG will be the start symbol in the PDA.

Step 4 – All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 – For each production in the form $A \rightarrow aX$ where a is terminal and A, X are combination of terminal and non-terminals, make a transition $\delta(q, a, A)$.

Problem

Construct a PDA from the following CFG.

$$G = (\{S, X\}, \{a, b\}, P, S)$$

where the productions are –

$$S \rightarrow XS \mid \epsilon, A \rightarrow aXb \mid Ab \mid ab$$

Solution

Let the equivalent PDA,

$$P = (\{q\}, \{a, b\}, \{a, b, X, S\}, \delta, q, S)$$

where δ –

$$\delta(q, \varepsilon, S) = \{(q, XS), (q, \varepsilon)\}$$

$$\delta(q, \varepsilon, X) = \{(q, aXb), (q, Xb), (q, ab)\}$$

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

$$\delta(q, 1, 1) = \{(q, \varepsilon)\}$$

Algorithm to find CFG corresponding to a given PDA

Input – A CFG, $G = (V, T, P, S)$

Output – Equivalent PDA, $P = (Q, \Sigma, S, \delta, q_0, I, F)$ such that the non-terminals of the grammar G will be $\{X_{wx} \mid w, x \in Q\}$ and the start state will be $A_{q_0, F}$.

Step 1 – For every $w, x, y, z \in Q, m \in S$ and $a, b \in \Sigma$, if $\delta(w, a, \varepsilon)$ contains (y, m) and $\delta(z, b, m)$ contains (x, ε) , add the production rule $X_{wx} \rightarrow a X_{yz} b$ in grammar G .

Step 2 – For every $w, x, y, z \in Q$, add the production rule $X_{wx} \rightarrow X_{wy} X_{yx}$ in grammar G .

Step 3 – For $w \in Q$, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G .

Non-Deterministic Pushdown Automaton (NPDA)

A non-deterministic pushdown automaton (NPDA), or just pushdown automaton (PDA) is a variation on the idea of a non-deterministic finite automaton (NFA). Unlike an NFA, a PDA is associated with a stack (hence the name pushdown). The transition function must also take into account the “state” of the stack.

Formally defined, a pushdown automaton MM is a 7-tuple $M = (Q, \Sigma, \Gamma, T, q_0, \perp, F)$, where $Q, \Sigma, q_0, \Gamma, \perp$, and F , like those in an NFA, are the set of states, the input alphabet, the start state, and the

Theory of Computation and Compilers Unit – 8

set of final states respectively. Γ is the stack alphabet, specifying the set of symbols that can be pushed onto the stack. Γ is not necessarily disjoint from Σ . $\perp$ is an element of Γ called the start stack symbol. The transition function is

$$T: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow P(Q \times \Gamma^*). T: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*).$$

How It Works

To see how the computing machine MM works, first imagine MM with the following features:

1. a finite set Q of internal states,
2. a horizontal tape of cells each containing an input symbol of Σ ,
3. a tape reader that reads at most one tape cell in any given internal state, and
4. a vertical stack of cells storing symbols of Γ .

Now, given that MM is in state p , with symbol A on top of the stack, and tape reader pointing at a tape cell containing symbol a , it may do one of the following:

- if $T(p, a, A) \neq \emptyset$, then it
 - (a) “pops” A off the stack,
 - (b) “pushes” word $A_1 \cdots A_n$ onto the stack, by starting with symbol A_n , and ending with symbol A_1 ,
 - (c) “consumes” a by moving the tape reader to the right of the cell containing a , and
 - (d) enters state q ,

provided that $(q, A_1 \dots A_n) \in T(p, a, A)$ $(q, A_1 \dots A_n) \in T(p, a, A)$;
if $T(p, a, A) = \emptyset$ $T(p, a, A) = \emptyset$, then MM does nothing.

- if $T(p, \lambda, A) \neq \emptyset$ $T(p, \lambda, A) \neq \emptyset$, then, without reading aa , it

5. (a) “pops” AA off the stack,
6. (b) “pushes” word $A_1 \dots A_n A_1 \dots A_n$ onto the stack, and
7. (c) enters state qq ,

as long as $(q, A_1 \dots A_n) \in T(p, \lambda, A)$ $(q, A_1 \dots A_n) \in T(p, \lambda, A)$;
if $T(p, \lambda, A) = \emptyset$ $T(p, \lambda, A) = \emptyset$, then MM does nothing.

If $(q, \lambda) \in T(p, a, A)$ $(q, \lambda) \in T(p, a, A)$, then AA gets popped off, and nothing gets pushed onto the stack.

Modes of Acceptance

A PDA is a language acceptor. We describe how words are accepted by a PDA MM. First, we start with configurations.

A configuration of MM is an element of $Q \times \Sigma^* \times \Gamma^* \times Q \times \Sigma^* \times \Gamma^*$. For any word uu , the configuration $(q_0, u, \perp)(q_0, u, \perp)$ is called the start configuration of uu . A binary relation $\vdash$ on the set of configurations is defined as follows:

if $(p, u, \alpha)(p, u, \alpha)$ and $(q, v, \beta)(q, v, \beta)$ are configurations of MM, then

$(p, u, \alpha) \vdash (q, v, \beta)(p, u, \alpha) \vdash (q, v, \beta)$

provided that $\alpha = A\gamma$ $\alpha = A\gamma$ and $\beta = B_1 \dots B_n \gamma$ $\beta = B_1 \dots B_n \gamma$, for some $A, B_1, \dots, B_n \in \Gamma$, $B_1, \dots, B_n \in \Gamma$, and

- either $u = avu = a\gamma v$, and $(q, B_1 \dots B_n) \in T(p, a, A)$ $(q, B_1 \dots B_n) \in T(p, a, A)$,
- or $u = vu = v$, and $(q, B_1 \dots B_n) \in T(p, \lambda, A)$ $(q, B_1 \dots B_n) \in T(p, \lambda, A)$.

Theory of Computation and Compilers Unit – 8

Now, take the reflexive transitive closure $\vdash^* \vdash^*$ of $\vdash$.

When $(p, u, \alpha) \vdash^* (q, v, \beta)$, we say that vv is derivable from uu . A word $u \in \Sigma^*$ is said to be

- accepted on final state by MM if $(q_0, u, \perp) \vdash^* (q, \lambda, \alpha)$ for some final state $q \in F$,
- accepted on empty stack by MM if $(q_0, u, \perp) \vdash^* (q, \lambda, \lambda)$,
- accepted on final state and empty stack by MM if $(q_0, u, \perp) \vdash^* (q, \lambda, \lambda)$ for some $q \in F$.

Languages Accepted by a PDA

Given a mode of acceptance, the set of words accepted by MM is called the language accepted by MM based on that mode of acceptance. Given a PDA MM, there are three languages accepted by MM, corresponding to the three acceptance modes above.

It turns out that three modes of acceptance are equivalent, in the following sense: if a language L is accepted by MM on one acceptance mode, there are PDA M_1 and M_2 that accept L in the other two acceptance modes.

In general, unless otherwise stated, the language $L(M)$ accepted by a PDA MM stands for the language accepted by MM on final state.

Remarks.

1. Two PDAs are said to be equivalent if they accept the same language. It can be shown that any PDA is equivalent to a PDA where $T(p, \lambda, A) = \emptyset$ for all $p \in F$ and $A \in \Gamma$ (called a $\lambda\lambda$ -free PDA).
2. One of the main reasons for studying PDA is: the notion of a PDA is equivalent to the notion of a context-free grammar. This means that, every

Theory of Computation and Compilers Unit – 8

language accepted by a PDA is context-free, and every context-free language is accepted by some PDA.

Representation by State Diagrams

Like an NDFA, a PDA can be presented visually as a directed graph, called a state diagram. Instead of simply labelling edges representing transitions with the leading symbol, two additional symbols are added, representing what symbol must be matched and removed from the top of the stack (or $\lambda\lambda$ if none) and what symbol should be pushed onto the stack (or $\lambda\lambda$ if none). For instance, the notation aA/B for an edge label indicates that a must be the first symbol in the remaining input string and A must be the symbol at the top of the stack for this transition to occur, and after the transition, A is replaced by B at the top of the stack. If the label had been $a\lambda/Ba\lambda\lambda/B$, then the symbol at the top of the stack would not matter (the stack could even be empty), and B would be pushed on top of the stack during the transition. If the label had been $aA/\lambda a\lambda\lambda A/\lambda$, A would be popped from the stack and nothing would replace it during the transition.

For example, consider the alphabet $\Sigma := \{ (,) \}$. Let us define a context-free language LL that consists of strings where the parentheses are fully balanced. If we define $\Gamma := \{ A \}$, then a PDA for accepting such strings is:

Non-deterministic Pushdown Automata

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more powerful than DPDA.

Example:

Design PDA for Palindrome strips.

Solution:

Theory of Computation and Compilers Unit – 8

Suppose the language consists of string $L = \{aba, aa, bb, bab, bbabb, aabaa, \dots\}$. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

1. $\delta(q_1, a, Z) = (q_1, aZ)$

2. $\delta(q_0, b, Z) = (q_1, bZ)$

3. $\delta(q_0, a, a) = (q_1, aa)$

4. $\delta(q_1, a, b) = (q_1, ab)$

5. $\delta(q_1, a, b) = (q_1, ba)$

6. $\delta(q_1, b, b) = (q_1, bb)$

7. $\delta(q_1, a, a) = (q_2, \epsilon)$

8. $\delta(q_1, b, b) = (q_2, \epsilon)$

9. $\delta(q_2, a, a) = (q_2, \epsilon)$

10. $\delta(q_2, b, b) = (q_2, \epsilon)$

11. $\delta(q_2, \epsilon, Z) = (q_2, \epsilon)$

Pushing the symbols onto the stack

Popping the symbols on reading the same kind of symbol

Simulation of abaaba

1. $\delta(q_1, abaaba, Z)$ Apply rule 1

2. $\vdash \delta(q_1, baaba, aZ)$ Apply rule 5

Theory of Computation and Compilers Unit – 8

3. $\vdash \delta(q1, aaba, baZ)$ Apply rule 4
4. $\vdash \delta(q1, aba, abaZ)$ Apply rule 7
5. $\vdash \delta(q2, ba, baZ)$ Apply rule 8
6. $\vdash \delta(q2, a, aZ)$ Apply rule 7
7. $\vdash \delta(q2, \epsilon, Z)$ Apply rule 11
8. $\vdash \delta(q2, \epsilon)$ Accept

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ϵ . For example, $A \rightarrow \epsilon$.
- A non-terminal generating two non-terminals. For example, $S \rightarrow AB$.
- A non-terminal generating a terminal. For example, $S \rightarrow a$.

For example:

1. $G1 = \{S \rightarrow AB, S \rightarrow c, A \rightarrow a, B \rightarrow b\}$
2. $G2 = \{S \rightarrow aA, A \rightarrow a, B \rightarrow c\}$

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as $S \rightarrow aZ$ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

Theory of Computation and Compilers Unit – 8

1. $S_1 \rightarrow S$

Where S_1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the [Simplification of CFG](#).

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production $S \rightarrow aA$ can be decomposed as:

1. $S \rightarrow RA$
2. $R \rightarrow a$

Step 4: Eliminate RHS with more than two non-terminals. For example, $S \rightarrow ASB$ can be decomposed as:

1. $S \rightarrow RS$
2. $R \rightarrow AS$

Example:

Convert the given CFG to CNF. Consider the given grammar G_1 :

1. $S \rightarrow a \mid aA \mid B$
2. $A \rightarrow aBB \mid \epsilon$
3. $B \rightarrow Aa \mid b$

Solution:

Step 1: We will create a new production $S_1 \rightarrow S$, as the start symbol S appears on the RHS. The grammar will be:

1. $S_1 \rightarrow S$
2. $S \rightarrow a \mid aA \mid B$

Theory of Computation and Compilers Unit – 8

3. $A \rightarrow aBB \mid \epsilon$

4. $B \rightarrow Aa \mid b$

Step 2: As grammar G1 contains $A \rightarrow \epsilon$ null production, its removal from the grammar yields:

1. $S1 \rightarrow S$

2. $S \rightarrow a \mid aA \mid B$

3. $A \rightarrow aBB$

4. $B \rightarrow Aa \mid b \mid a$

Now, as grammar G1 contains Unit production $S \rightarrow B$, its removal yield:

1. $S1 \rightarrow S$

2. $S \rightarrow a \mid aA \mid Aa \mid b$

3. $A \rightarrow aBB$

4. $B \rightarrow Aa \mid b \mid a$

Also remove the unit production $S1 \rightarrow S$, its removal from the grammar yields:

1. $S0 \rightarrow a \mid aA \mid Aa \mid b$

2. $S \rightarrow a \mid aA \mid Aa \mid b$

3. $A \rightarrow aBB$

4. $B \rightarrow Aa \mid b \mid a$

Step 3: In the production rule $S0 \rightarrow aA \mid Aa$, $S \rightarrow aA \mid Aa$, $A \rightarrow aBB$ and $B \rightarrow Aa$, terminal a exists on RHS with non-terminals. So we will replace terminal a with X :

1. $S0 \rightarrow a \mid XA \mid AX \mid b$

2. $S \rightarrow a \mid XA \mid AX \mid b$

3. $A \rightarrow XBB$

4. $B \rightarrow AX \mid b \mid a$

5. $X \rightarrow a$

Step 4: In the production rule $A \rightarrow XBB$, RHS has more than two symbols, removing it from grammar yield:

1. $S_0 \rightarrow a \mid XA \mid AX \mid b$

2. $S \rightarrow a \mid XA \mid AX \mid b$

3. $A \rightarrow RB$

4. $B \rightarrow AX \mid b \mid a$

5. $X \rightarrow a$

6. $R \rightarrow XB$

Hence, for the given grammar, this is the required CNF.

Greibach Normal Form (GNF)

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:

- A start symbol generating ϵ . For example, $S \rightarrow \epsilon$.
- A non-terminal generating a terminal. For example, $A \rightarrow a$.
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, $S \rightarrow aASB$.

For example:

1. $G_1 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid a, B \rightarrow bB \mid b\}$

$$2. G_2 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid \epsilon, B \rightarrow bB \mid \epsilon\}$$

The production rules of Grammar G_1 satisfy the rules specified for GNF, so the grammar G_1 is in GNF. However, the production rule of Grammar G_2 does not satisfy the rules specified for GNF as $A \rightarrow \epsilon$ and $B \rightarrow \epsilon$ contains ϵ (only start symbol can generate ϵ). So the grammar G_2 is not in GNF.

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

If any production rule in the grammar is not in GNF form, convert it.

Example:

$$1. S \rightarrow XB \mid AA$$

$$2. A \rightarrow a \mid SA$$

$$3. B \rightarrow b$$

$$4. X \rightarrow a$$

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

Theory of Computation and Compilers Unit – 8

The production rule $A \rightarrow SA$ is not in GNF, so we substitute $S \rightarrow XB \mid AA$ in the production rule $A \rightarrow SA$ as:

1. $S \rightarrow XB \mid AA$
2. $A \rightarrow a \mid XBA \mid AAA$
3. $B \rightarrow b$
4. $X \rightarrow a$

The production rule $S \rightarrow XB$ and $B \rightarrow XBA$ is not in GNF, so we substitute $X \rightarrow a$ in the production rule $S \rightarrow XB$ and $B \rightarrow XBA$ as:

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow a \mid aBA \mid AAA$
3. $B \rightarrow b$
4. $X \rightarrow a$

Now we will remove left recursion ($A \rightarrow AAA$), we get:

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow aC \mid aBAC$
3. $C \rightarrow AAC \mid \epsilon$
4. $B \rightarrow b$
5. $X \rightarrow a$

Now we will remove null production $C \rightarrow \epsilon$, we get:

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$
3. $C \rightarrow AAC \mid AA$

Theory of Computation and Compilers Unit – 8

4. $B \rightarrow b$

5. $X \rightarrow a$

The production rule $S \rightarrow AA$ is not in GNF, so we substitute $A \rightarrow aC \mid aBAC \mid a \mid aBA$ in production rule $S \rightarrow AA$ as:

1. $S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$

2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$

3. $C \rightarrow AAC$

4. $C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$

5. $B \rightarrow b$

6. $X \rightarrow a$

The production rule $C \rightarrow AAC$ is not in GNF, so we substitute $A \rightarrow aC \mid aBAC \mid a \mid aBA$ in production rule $C \rightarrow AAC$ as:

1. $S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$

2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$

3. $C \rightarrow aCAC \mid aBACAC \mid aAC \mid aBAAC$

4. $C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$

5. $B \rightarrow b$

6. $X \rightarrow a$

Hence, this is the GNF form for the grammar G.

Ambiguity

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for

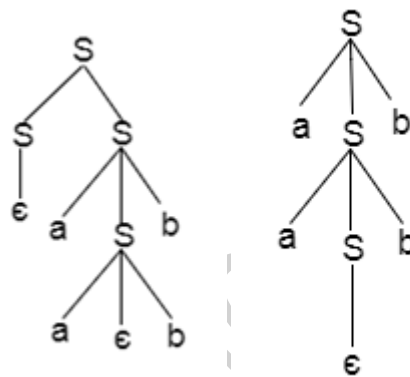
Theory of Computation and Compilers Unit – 8

the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

1. $S = aSb \mid SS$
2. $S = \epsilon$

For the string aabb, the above grammar generates two parse trees:



If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

Ambiguity in Context-Free Grammars

If a context free grammar **G** has more than one derivation tree for some string **w** $\in L(G)$, it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem

Check whether the grammar **G** with production rules –

$$X \rightarrow X+X \mid X*X \mid X \mid a$$

is ambiguous or not.

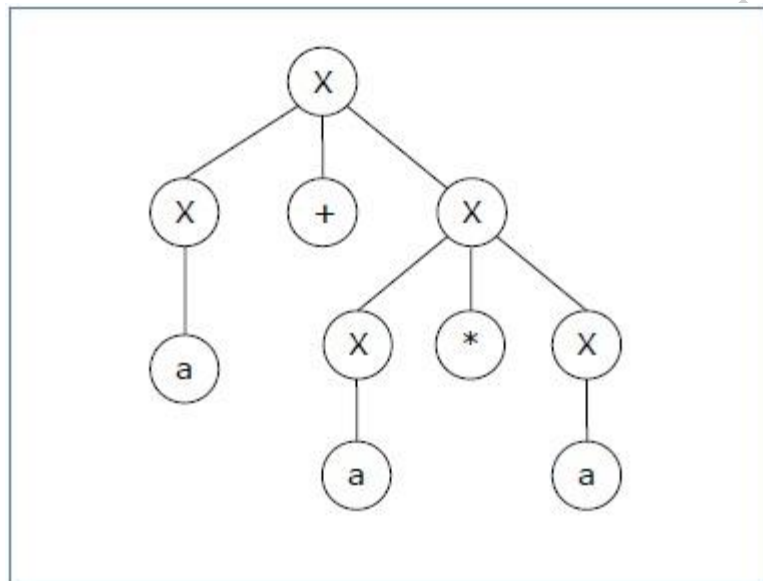
Theory of Computation and Compilers Unit – 8

Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

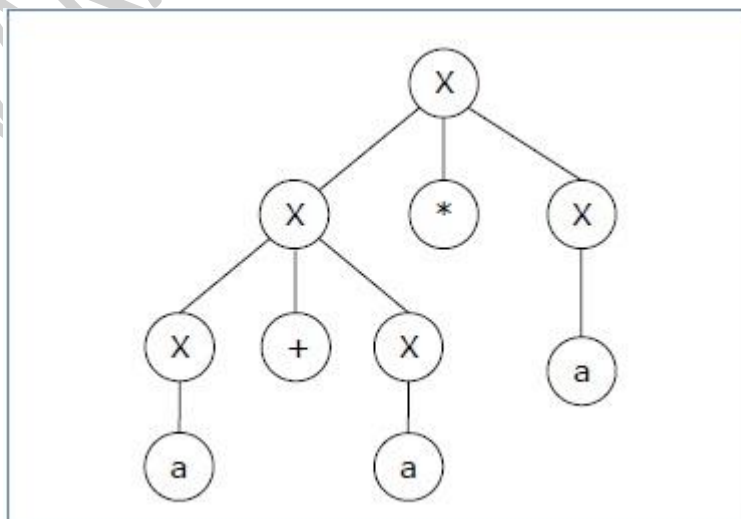
Derivation 1 – $X \rightarrow X+X \rightarrow a+X \rightarrow a+X*X \rightarrow a+a*X \rightarrow a+a*a$

Parse tree 1 –



Derivation 2 – $X \rightarrow X*X \rightarrow X+X*X \rightarrow a+X*X \rightarrow a+a*X \rightarrow a+a*a$

Parse tree 2 –



Theory of Computation and Compilers Unit – 8

Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

Production rules:

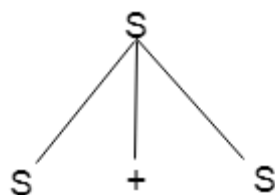
1. $T = T + T \mid T * T$
2. $T = a \mid b \mid c$

Input:

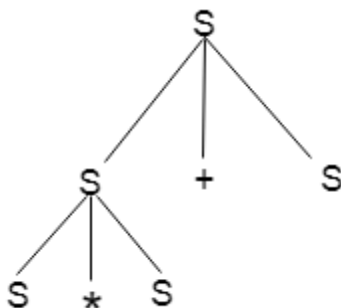
$a * b + c$

Step 1:

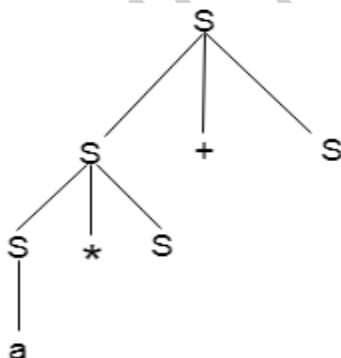
Theory of Computation and Compilers Unit – 8



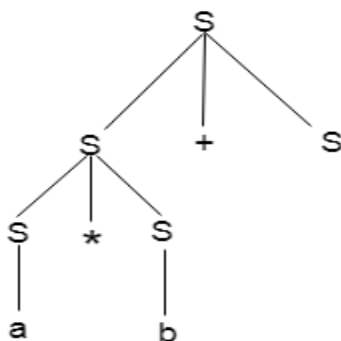
Step 2:



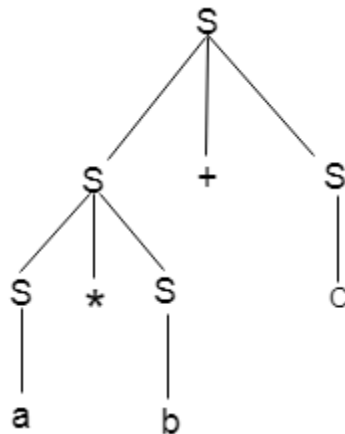
Step 3:



Step 4:



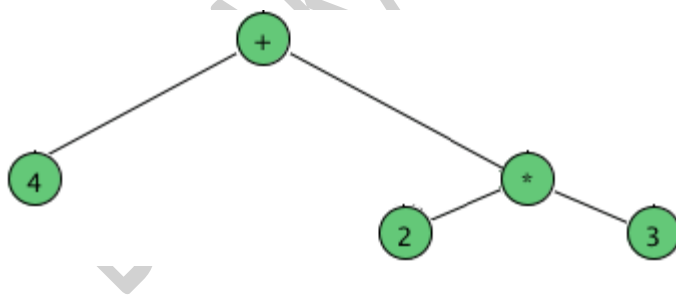
Step 5:



Abstract Syntax Tree

A computer language is basically a context-free language. The symbols are tokens without any particular semantic meaning, namely, all numbers are the same, perhaps even all literal things (strings, numbers, etc) are regarded equally. All variables are regarded equally, etc. So the point is that we have a finite symbol set.

The first step of a compiler is to create a parse tree of the program, and the second phase is to assign meaning, or semantics to the entities in the tree. In reality, you create an abstract syntax tree of the the program. For example, considering the parse tree for $4 + 2 * 3$ above, an abstract syntax tree for this expression would look like this:



Parse Trees and Derivations

First of all, what is a derivation? It's a sequence of strings in V^* which starts with a non-terminal (in $V - \Sigma$ and ends with a string in Σ^* . Let's consider the sample grammar

Theory of Computation and Compilers Unit – 8

$$E \rightarrow E+E \mid a$$

We write:

$$E \Rightarrow E+E \Rightarrow E+E+E \Rightarrow a+E+E \Rightarrow a+a+E \Rightarrow a+a+a$$

but this is incomplete, because it doesn't tell us where the replacement rules are applied. We actually need "marked" strings which indicate which non-terminal is replaced in all but the first and last step:

$$E \Rightarrow \check{E}+E \Rightarrow \check{E}+E+E \Rightarrow a+\check{E}+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a$$

In this case, the marking is only necessary in the second step; however it is crucial, because we want to distinguish between this derivation and the following one:

$$E \Rightarrow E+\check{E} \Rightarrow \check{E}+E+E \Rightarrow a+\check{E}+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a$$

We want to characterize two derivations as "coming from the same parse tree." Technically speaking, we want to characterize a parse tree as an equivalence class of derivations.

The first step is to define the relation among derivations as being "more left-oriented at one step". Assume we have two equal length derivations of length $n > 2$:

$$D: x_1 \Rightarrow x_2 \Rightarrow \dots \Rightarrow x_n$$

$$D': x_1' \Rightarrow x_2' \Rightarrow \dots \Rightarrow x_n'$$

where

$$x_1 = x_1' \text{ is a non-terminal}$$

and

$$x_n = x_n' \in \Sigma^*$$

Namely they start with the same non-terminal and end at the same terminal string and have at least two intermediate steps.

Theory of Computation and Compilers Unit – 8

We want to say $D < D'$ if the two derivations differ in only one step in which there are 2 non-terminals, A and B, such that D replaces the left one before the right one and D' does the opposite. Formally:

$D < D'$ if there exists k , $1 < k < n$ such that

$x_i = x'_i$ for all $i \neq k$ (equal strings, same marked position)

$x_{k-1} = u\check{A}vBw$, for $u, v, w \in V^*$

$x_{k-1}' = uAv\check{B}w$, for $u, v, w \in V^*$

$x_k = uyv\check{B}w$, for production $A \rightarrow y$

$x_k' = u\check{A}vzw$, for production $B \rightarrow z$

$x_{k+1} = x_{k+1}' = uyvzw$ (marking not shown)

Two derivations are said to be similar if they belong to the reflexive, symmetric, transitive closure of $<$. Therefore, two derivations are similar if one can be transformed to the other by a sequence of switching of the order in which rules are applied. Similarity is an equivalence relation and we think of the parse tree of a string as the **equivalence class** of all similar derivations.

Textbook Issue

The textbook never explicitly says that derivations consist of marked strings, just strings. Unfortunately this would make it so that the initial portion of the derivation

$$E \Rightarrow E+E \Rightarrow E+E+E$$

could be interpreted ambiguously as either of these two:

$$E \Rightarrow \check{E}+E \Rightarrow E+E+E$$

$$E \Rightarrow E+\check{E} \Rightarrow E+E+E$$

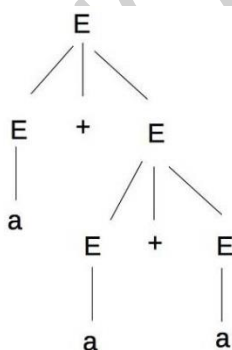
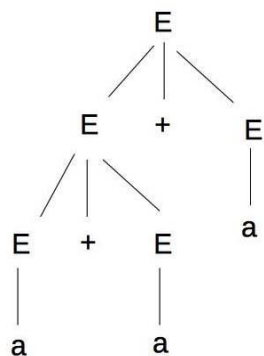
Basic Expression Grammar

Theory of Computation and Compilers Unit – 8

Consider again, the grammar specifying only addition in expression:

$$E \rightarrow E+E \mid a$$

This we have described as being ambiguous. Here are two parse trees for the string "a+a+a":



For the left parse tree, we have these derivations:

L1: $E \Rightarrow \check{E}+E \Rightarrow \check{E}+E+E \Rightarrow a+\check{E}+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a$ (left-most)

L2: $E \Rightarrow \check{E}+E \Rightarrow E+\check{E}+E \Rightarrow \check{E}+a+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a$

L3: $E \Rightarrow \check{E}+E \Rightarrow E+\check{E}+E \Rightarrow E+a+\check{E} \Rightarrow \check{E}+a+a \Rightarrow a+a+a$

L4: $E \Rightarrow \check{E}+E \Rightarrow E+E+\check{E} \Rightarrow E+\check{E}+a \Rightarrow \check{E}+a+a \Rightarrow a+a+a$

L5: $E \Rightarrow E+\check{E} \Rightarrow \check{E}+a \Rightarrow E+\check{E}+a \Rightarrow \check{E}+a+a \Rightarrow a+a+a$ (right-most)

Theory of Computation and Compilers Unit – 8

These are structured so that $L1 < L2 < L3 < L4 < L5$ and so are all equivalent.

For the right parse tree, we have these derivations:

$$R1: E \Rightarrow \check{E}+E \Rightarrow a+\check{E} \Rightarrow a+\check{E}+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a \text{ (left-most)}$$

$$R2: E \Rightarrow E+\check{E} \Rightarrow \check{E}+E+E \Rightarrow a+\check{E}+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a$$

$$R3: E \Rightarrow E+\check{E} \Rightarrow E+\check{E}+E \Rightarrow \check{E}+a+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a$$

$$R4: E \Rightarrow E+\check{E} \Rightarrow E+\check{E}+E \Rightarrow E+a+\check{E} \Rightarrow \check{E}+a+a \Rightarrow a+a+a$$

$$R5: E \Rightarrow E+\check{E} \Rightarrow E+E+\check{E} \Rightarrow E+\check{E}+a \Rightarrow \check{E}+a+a \Rightarrow a+a+a \text{ (right-most)}$$

Again, $R1 < R2 < R3 < R4 < R5$ and so are all equivalent. The point is that none of the derivations for the left parse tree are similar to any of the derivations of the right parse tree. For example, these two would be related if it were not for the crucial second step in the derivation:

$$R3: E \Rightarrow E+\check{E} \Rightarrow E+\check{E}+E \Rightarrow \check{E}+a+E \Rightarrow a+a+\check{E} \Rightarrow a+a+a$$

$$L3: E \Rightarrow \check{E}+E \Rightarrow E+\check{E}+E \Rightarrow E+a+\check{E} \Rightarrow \check{E}+a+a \Rightarrow a+a+a$$

Similarity and parse trees

We claim that if $D < D'$, then both derivations construct the same parse tree in reverse. This is pretty obvious since the tree constructions differ only in two steps, one where A is created before B and the other vice-versa. From this, we can deduce that any two similar derivations create the same parse tree.

Going the other way, we want to say that if two derivations construct the same parse tree, they must be similar. This is an inductive argument. Consider the root of the parse tree and its children:

$$(A \ t_1 \ t_2 \ \dots \ t_n)$$

Theory of Computation and Compilers Unit – 8

Inductively, we assume the two derivations are similar for all subtrees: $t_1, t_2, \dots, t_n$. It follows immediately by the transitivity and symmetry of the similarity relation that the two derivations are similar for the whole tree.

Leftmost and rightmost derivations

The similarity relation, being an equivalence relation, partitions the set of all derivations of a given string. These partitions are finite, and we can argue that, within a partition, there is a unique one D_L such that

$$D_L < D, \text{ for all } D \neq D_L, D \text{ is similar to } D_L$$

D_L is said to be a **leftmost** derivation. It is easy to convince yourself that any derivation of one or two steps must be leftmost. Similarly we can define and argue the existence of a **rightmost** derivation.

Balanced Parentheses

The grammar is:

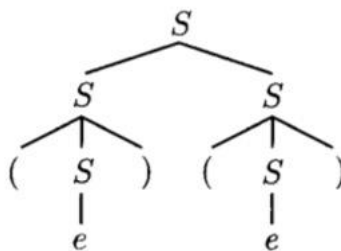
$$S \rightarrow SS \mid (S) \mid \epsilon$$

Consider the string " $()()$ ". It has at several distinct derivations, such as

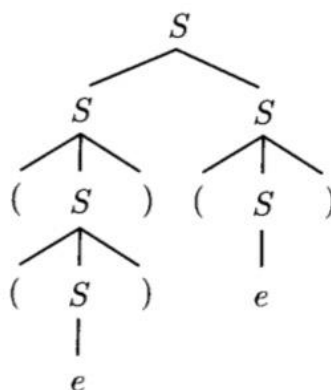
$$S \Rightarrow \check{S}S \Rightarrow (\check{S})S \Rightarrow ()\check{S} \Rightarrow ()(S) \Rightarrow ()()$$

$$S \Rightarrow S\check{S} \Rightarrow \check{S}(S) \Rightarrow (S)(\check{S}) \Rightarrow (\check{S})() \Rightarrow ()()$$

They are similar because they come from the same parse tree:



Then consider this parse tree for the string " $((()))()$ "



Here are several similar derivations:

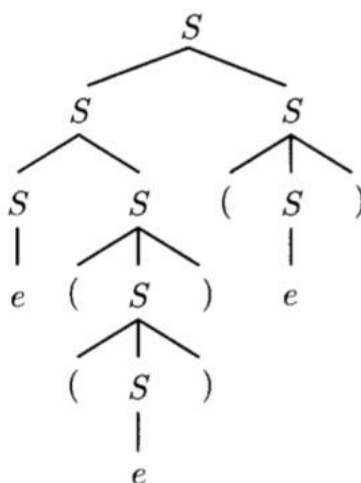
$$D1: S \Rightarrow \check{S}S \Rightarrow (\check{S})S \Rightarrow ((\check{S}))S \Rightarrow (())\check{S} \Rightarrow (())(\check{S}) \Rightarrow (())()$$

$$D2: S \Rightarrow \check{S}S \Rightarrow (\check{S})S \Rightarrow ((S))\check{S} \Rightarrow ((\check{S}))(S) \Rightarrow (())(\check{S}) \Rightarrow (())()$$

$$D3: S \Rightarrow \check{S}S \Rightarrow (\check{S})S \Rightarrow ((S))\check{S} \Rightarrow ((S))(\check{S}) \Rightarrow ((\check{S}))() \Rightarrow (())()$$

In this case $D1 < D2 < D3$

This language is ambiguous, and so we also have the parse tree for the string " $()()$ "



which has a derivation unrelated to the others:

$$S \Rightarrow \check{S}S \Rightarrow S\check{S}S \Rightarrow S(\check{S})S \Rightarrow S((\check{S}))S \Rightarrow S(())\check{S} \Rightarrow S(())(\check{S}) \Rightarrow \check{S}()() \Rightarrow (())()$$

CFG to PDA Conversion

The first symbol on R.H.S. production must be a terminal symbol. The following steps are used to obtain PDA from CFG is:

Step 1: Convert the given productions of CFG into GNF.

Step 2: The PDA will only have one state $\{q\}$.

Step 3: The initial symbol of CFG will be the initial symbol in the PDA.

Step 4: For non-terminal symbol, add the following rule:

$$1. \delta(q, \epsilon, A) = (q, \alpha)$$

Where the production rule is $A \rightarrow \alpha$

Step 5: For each terminal symbols, add the following rule:

$$1. \delta(q, a, a) = (q, \epsilon) \text{ for every terminal symbol}$$

Example 1:

Convert the following grammar to a PDA that accepts the same language.

$$1. S \rightarrow 0S1 \mid A$$

$$2. A \rightarrow 1A0 \mid S \mid \epsilon$$

Solution:

The CFG can be first simplified by eliminating unit productions:

$$1. S \rightarrow 0S1 \mid 1S0 \mid \epsilon$$

Now we will convert this CFG to GNF:

$$1. S \rightarrow 0SX \mid 1SY \mid \epsilon$$

$$2. X \rightarrow 1$$

3. $Y \rightarrow 0$

The PDA can be:

R1: $\delta(q, \epsilon, S) = \{(q, 0SX) \mid (q, 1SY) \mid (q, \epsilon)\}$

R2: $\delta(q, \epsilon, X) = \{(q, 1)\}$

R3: $\delta(q, \epsilon, Y) = \{(q, 0)\}$

R4: $\delta(q, 0, 0) = \{(q, \epsilon)\}$

R5: $\delta(q, 1, 1) = \{(q, \epsilon)\}$

Example 2:

Construct PDA for the given CFG, and test whether 0104 is acceptable by this PDA.

1. $S \rightarrow 0BB$

2. $B \rightarrow 0S \mid 1S \mid 0$

Solution:

The PDA can be given as:

1. $A = \{(q), (0, 1), (S, B, 0, 1), \delta, q, S, ?\}$

The production rule δ can be:

R1: $\delta(q, \epsilon, S) = \{(q, 0BB)\}$

R2: $\delta(q, \epsilon, B) = \{(q, 0S) \mid (q, 1S) \mid (q, 0)\}$

R3: $\delta(q, 0, 0) = \{(q, \epsilon)\}$

R4: $\delta(q, 1, 1) = \{(q, \epsilon)\}$

Testing 010^4 i.e. 010000 against PDA:

1. $\delta(q, 010000, S) \vdash \delta(q, 010000, 0BB)$

Theory of Computation and Compilers Unit – 8

- | | | |
|-----|--------------------------------|----|
| 2. | $\vdash \delta(q, 10000, BB)$ | R1 |
| 3. | $\vdash \delta(q, 10000, 1SB)$ | R3 |
| 4. | $\vdash \delta(q, 0000, SB)$ | R2 |
| 5. | $\vdash \delta(q, 0000, 0BBB)$ | R1 |
| 6. | $\vdash \delta(q, 000, BBB)$ | R3 |
| 7. | $\vdash \delta(q, 000, 0BB)$ | R2 |
| 8. | $\vdash \delta(q, 00, BB)$ | R3 |
| 9. | $\vdash \delta(q, 00, 0B)$ | R2 |
| 10. | $\vdash \delta(q, 0, B)$ | R3 |
| 11. | $\vdash \delta(q, 0, 0)$ | R2 |
| 12. | $\vdash \delta(q, \epsilon)$ | R3 |
| 13. | ACCEPT | |

Thus 010^4 is accepted by the PDA.

Example 3:

Draw a PDA for the CFG given below:

1. $S \rightarrow aSb$
2. $S \rightarrow a \mid b \mid \epsilon$

Solution:

The PDA can be given as:

1. $P = \{(q), (a, b), (S, a, b, z_0), \delta, q, z_0, q\}$

The mapping function δ will be:

R1: $\delta(q, \epsilon, S) = \{(q, aSb)\}$

R2: $\delta(q, \epsilon, S) = \{(q, a) \mid (q, b) \mid (q, \epsilon)\}$

R3: $\delta(q, a, a) = \{(q, \epsilon)\}$

R4: $\delta(q, b, b) = \{(q, \epsilon)\}$

R5: $\delta(q, \epsilon, z0) = \{(q, \epsilon)\}$

Simulation: Consider the string aaabb

1. $\delta(q, \epsilon aaabb, S) \vdash \delta(q, aaabb, aSb)$ R3
2. $\vdash \delta(q, \epsilon aabb, Sb)$ R1
3. $\vdash \delta(q, aabb, aSbb)$ R3
4. $\vdash \delta(q, \epsilon abb, Sbb)$ R2
5. $\vdash \delta(q, abb, abb)$ R3
6. $\vdash \delta(q, bb, bb)$ R4
7. $\vdash \delta(q, b, b)$ R4
8. $\vdash \delta(q, \epsilon, z0)$ R5
9. $\vdash \delta(q, \epsilon)$
10. ACCEPT

Turing Machine

Turing machine was invented in 1936 by **Alan Turing**. It is an accepting device which accepts Recursive Enumerable Language generated by type 0 grammar.

There are various features of the Turing machine:

1. It has an external memory which remembers arbitrary long sequence of input.

Theory of Computation and Compilers Unit – 8

2. It has unlimited memory capability.
3. The model has a facility by which the input at left or right on the tape can be read easily.
4. The machine can produce a certain output based on its input. Sometimes it may be required that the same input has to be used to generate the output. So in this machine, the distinction between input and output has been removed. Thus a common set of alphabets can be used for the Turing machine.

Formal definition of Turing machine

A Turing machine can be defined as a collection of 7 components:

0	0	1	1	Δ	-----
---	---	---	---	----------	-------

Now, we will see how this turing machine will works for 0011. Initially, state is q_0 and head points to 0 as:

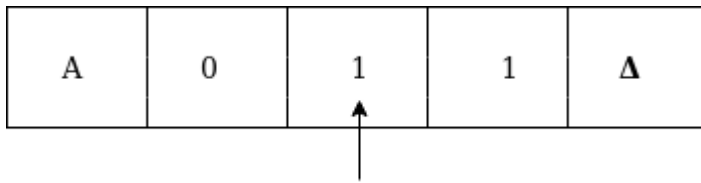
0	0	1	1	Δ
↑				

The move will be $\delta(q_0, 0) = \delta(q_1, A, R)$ which means it will go to state q_1 , replaced 0 by A and head will move to the right as:

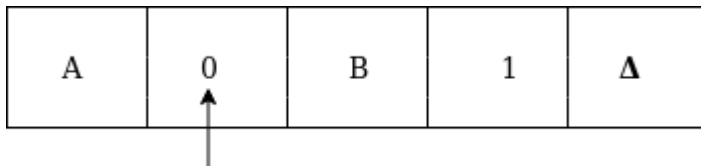
A	0	1	1	Δ
	↑			

The move will be $\delta(q_1, 0) = \delta(q_1, 0, R)$ which means it will not change any symbol, remain in the same state and move to the right as:

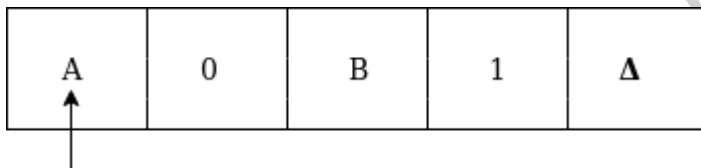
Theory of Computation and Compilers Unit – 8



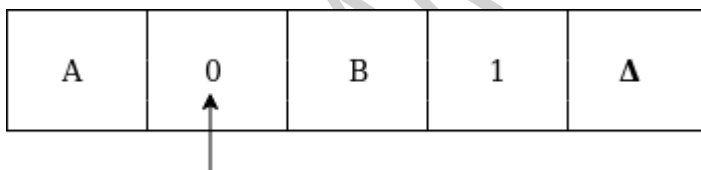
The move will be $\delta(q_1, 1) = \delta(q_2, B, L)$ which means it will go to state q_2 , replaced 1 by B and head will move to left as:



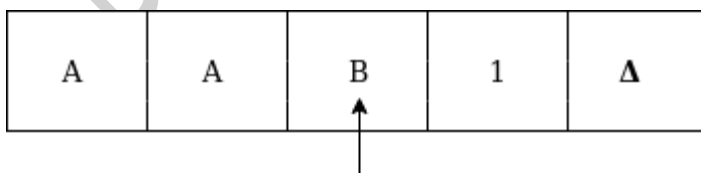
Now move will be $\delta(q_2, 0) = \delta(q_2, 0, L)$ which means it will not change any symbol, remain in the same state and move to left as:



The move will be $\delta(q_2, A) = \delta(q_0, A, R)$, it means will go to state q_0 , replaced A by A and head will move to the right as:

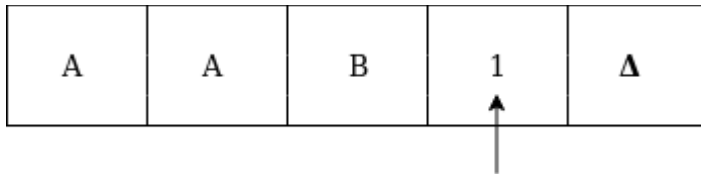


The move will be $\delta(q_0, 0) = \delta(q_1, A, R)$ which means it will go to state q_1 , replaced 0 by A, and head will move to right as:

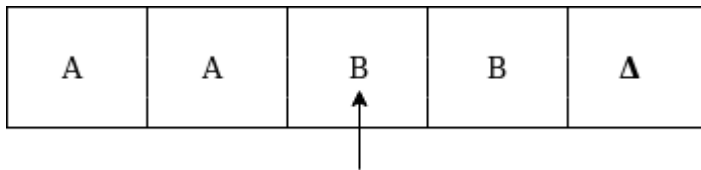


Theory of Computation and Compilers Unit – 8

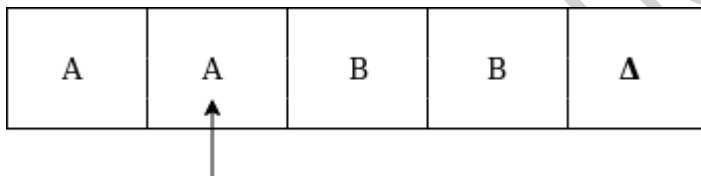
The move will be $\delta(q_1, B) = \delta(q_1, B, R)$ which means it will not change any symbol, remain in the same state and move to right as:



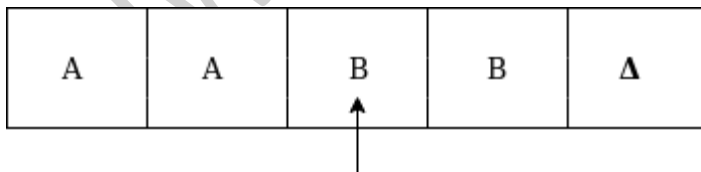
The move will be $\delta(q_1, 1) = \delta(q_2, B, L)$ which means it will go to state q_2 , replaced 1 by B and head will move to left as:



The move $\delta(q_2, B) = (q_2, B, L)$ which means it will not change any symbol, remain in the same state and move to left as:

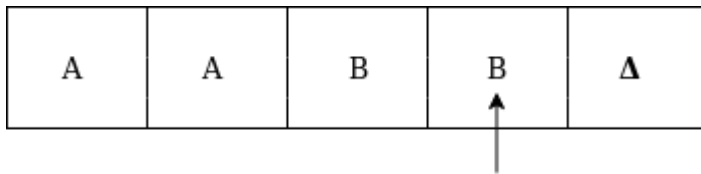


Now immediately before B is A that means all the 0's are marked by A. So we will move right to ensure that no 1 is present. The move will be $\delta(q_2, A) = (q_0, A, R)$ which means it will go to state q_0 , will not change any symbol, and move to right as:

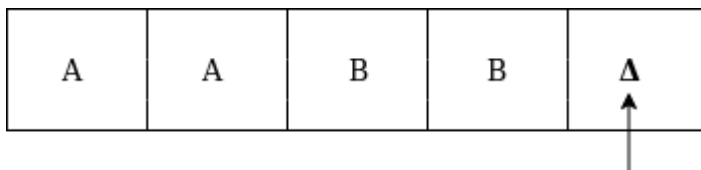


The move $\delta(q_0, B) = (q_3, B, R)$ which means it will go to state q_3 , will not change any symbol, and move to right as:

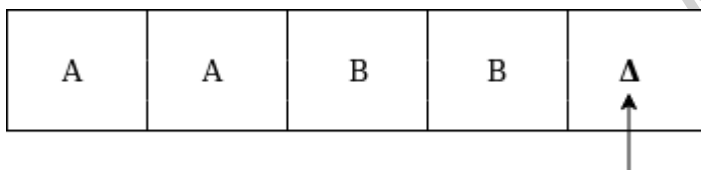
Theory of Computation and Compilers Unit – 8



The move $\delta(q_3, B) = (q_3, B, R)$ which means it will not change any symbol, remain in the same state and move to right as:



The move $\delta(q_3, \Delta) = (q_4, \Delta, R)$ which means it will go to state q_4 which is the HALT state and HALT state is always an accept state for any TM.

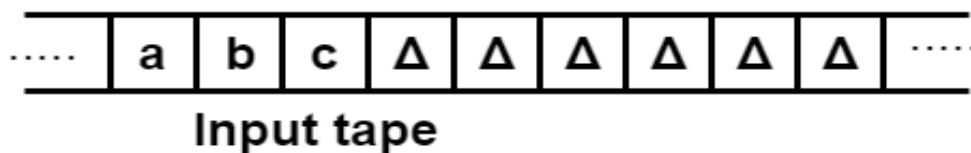


The same TM can be represented by Transition Diagram:

Basic Model of Turing machine

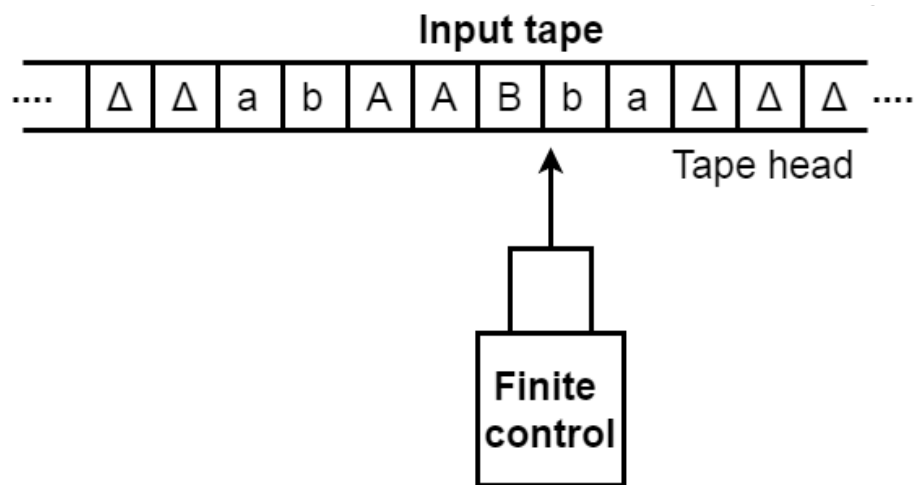
The turning machine can be modelled with the help of the following representation.

1. The input tape is having an infinite number of cells, each cell containing one input symbol and thus the input string can be placed on tape. The empty tape is filled by blank characters.



Theory of Computation and Compilers Unit – 8

2. The finite control and the tape head which is responsible for reading the current input symbol. The tape head can move to left to right.
3. A finite set of states through which machine has to undergo.
4. Finite set of symbols called external symbols which are used in building the logic of turing machine.



Accepted Language & Decided Language

A TM accepts a language if it enters into a final state for any input string w. A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.

A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine.

There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

Designing a Turing Machine

The basic guidelines of designing a Turing machine have been explained below with the help of a couple of examples.

Theory of Computation and Compilers Unit – 8

Example 1

Design a TM to recognize all strings consisting of an odd number of α 's.

Solution

The Turing machine **M** can be constructed by the following moves –

- Let q_1 be the initial state.
- If **M** is in q_1 ; on scanning α , it enters the state q_2 and writes **B** (blank).
- If **M** is in q_2 ; on scanning α , it enters the state q_1 and writes **B** (blank).
- From the above moves, we can see that **M** enters the state q_1 if it scans an even number of α 's, and it enters the state q_2 if it scans an odd number of α 's. Hence q_2 is the only accepting state.

Hence,

$$M = \{\{q_1, q_2\}, \{1\}, \{1, B\}, \delta, q_1, B, \{q_2\}\}$$

where δ is given by –

Tape alphabet symbol	Present State ' q_1 '	Present State ' q_2 '
α	BR q_2	BR q_1

Example 2

Design a Turing Machine that reads a string representing a binary number and erases all leading 0's in the string. However, if the string comprises of only 0's, it keeps one 0.

Solution

Let us assume that the input string is terminated by a blank symbol, B, at each end of the string.

Theory of Computation and Compilers Unit – 8

The Turing Machine, **M**, can be constructed by the following moves –

- Let **q₀** be the initial state.
- If **M** is in **q₀**, on reading 0, it moves right, enters the state **q₁** and erases 0. On reading 1, it enters the state **q₂** and moves right.
- If **M** is in **q₁**, on reading 0, it moves right and erases 0, i.e., it replaces 0's by B's. On reaching the leftmost 1, it enters **q₂** and moves right. If it reaches B, i.e., the string comprises of only 0's, it moves left and enters the state **q₃**.
- If **M** is in **q₂**, on reading either 0 or 1, it moves right. On reaching B, it moves left and enters the state **q₄**. This validates that the string comprises only of 0's and 1's.
- If **M** is in **q₃**, it replaces B by 0, moves left and reaches the final state **q_f**.
- If **M** is in **q₄**, on reading either 0 or 1, it moves left. On reaching the beginning of the string, i.e., when it reads B, it reaches the final state **q_f**.

Hence,

$$M = \{\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{0, 1, B\}, \{1, B\}, \delta, q_0, B, \{q_f\}\}$$

where δ is given by –

Tape alphabet symbol	Present State 'q ₀ '	Present State 'q ₁ '	Present State 'q ₂ '	Present State 'q ₃ '	Present State 'q ₄ '
0	BRq ₁	BRq ₁	ORq ₂	-	OLq ₄
1	1Rq ₂	1Rq ₂	1Rq ₂	-	1Lq ₄
B	BRq ₁	BLq ₃	BLq ₄	OLq _f	BRq _f

Standard Turing Machine

A standard Turing Machine is a machine which on providing an input moves either to the left or right and it may overwrite the existing symbol.

A standard TM can be described as a 7-tuple:

$(Q, X, *, f, q_0, B, F)$

where

Q is a finite set of states

X is the tape alphabet

* is the input alphabet

f is a transition function;

$f: Q \times X \rightarrow Q \times X \times \{\text{Left_shift}, \text{Right_shift}\}.$

q_0 is the initial state

B is the blank symbol

F is the set of final states

A standard Turing machine is capable of accepting some of the languages, called recursively enumerable language. Let's see if by doing some kind of modification we can increase the number of languages accepted by Turing Machine.

1. Turing Machine with Stay option:

If instead of moving left or right on seeing an input, the head could also stay at one position without moving anywhere i.e.

$f: Q \times X \rightarrow Q \times X \times \{\text{Left_shift}, \text{Right_shift}, \text{Stay}\}.$

Still the number of languages accepted by turing machine remains same .

1.

Turing Machine with Semi-infinite tape:

We know that Turing machine has an infinite input tape with extends in both the directions (left and right) infinitely. So now if we restrict it to extend only in one direction and not in both the directions, i.e., we make the tape to be semi infinite, then also the number of languages accepted by the Turing machine remains same.

2. **Offline Turing machine:**

In standard Turing machine both the input and output are present on the tape, the head has the authority to move across the input and can change or modify the input, if we don't want to modify the input we can provide the input in separate file, which is read only then the head cannot make changes to it.

If the Turing machine wants to modify the input, the input need to be copied on the tape and the changes can be made by the head but still the input file remains unchanged as changes are made in the tape and not in the file. By doing such modification to Turing machine still the number of languages accepted by the Turing machine remains the same.

3. **Jumping Turing Machine:**

The standard Turing machine's head can move only one step to the right or left, but in case of jumping Turing machine the head can move not only just one step to the left or right but it can move more the one, i.e., 2, 3, 4, 5, 6, ... so on, or it can jump to any cell to the right or left of the input tape.

$f: Q \times X \rightarrow Q \times X \times \{\text{Left_shift}, \text{Right_shift}\} \times \{n\}$

n , is the number of steps that we wish to move to the right or left. But still the languages accepted by Turing machine remains the same.

4. **Non erasing Turing Machine:**

In standard Turing machine the input symbol can be changed to blank, but

if we remove this facility of changing the input symbol to blank then such type of Turing machine is called as non erasing Turing machine . We can replace the input with any other symbol except blank. By doing this modification still the number of languages accepted by the Turing machine remains the same.

5. **Always writing Turing Machine:**

Standard Turing machine gives us the freedom that on seeing an input we can leave it as it is without doing any changes but in always writing Turing machine it is compulsory to modify the input whenever we see it we cannot leave it as it is. But this kind of modification didn't help in increasing the number of languages accepted by the Turing machine.

Hence, we see that in spite of doing so many modifications to the Turing machine still the number of languages accepted by it remains the same. Therefore, Turing Machine is most powerful machine.

Universal Turing Machine

A Turing Machine is the mathematical tool equivalent to a digital computer. It was suggested by the mathematician Turing in the 30s, and has been since then the most widely used model of computation in computability and complexity theory.

The model consists of an input output relation that the machine computes. The input is given in binary form on the machine's tape, and the output consists of the contents of the tape when the machine halts.

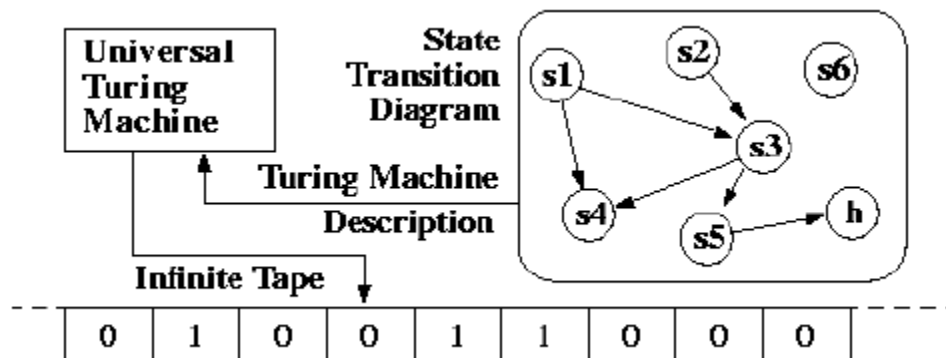
What determines how the contents of the tape change is a finite state machine (or FSM, also called a finite automaton) inside the Turing Machine. The FSM is determined by the number of states it has, and the transitions between them.

At every step, the current state and the character read on the tape determine the next state the FSM will be in, the character that the machine will output on the tape (possibly the one read, leaving the contents unchanged), and which direction the head moves in, left or right.

Theory of Computation and Compilers Unit – 8

The problem with Turing Machines is that a different one must be constructed for every new computation to be performed, for every input output relation.

This is why we introduce the notion of a universal turing machine (UTM), which along with the input on the tape, takes in the description of a machine M. The UTM can go on then to simulate M on the rest of the contents of the input tape. A universal turing machine can thus simulate any other machine.



Church Turing Thesis

In 1936, A method named as lambda-calculus was created by Alonzo Church in which the Church numerals are well defined, i.e. the encoding of natural numbers. Also in 1936, Turing machines (earlier called theoretical model for machines) was created by Alan Turing, that is used for manipulating the symbols of string with the help of tape.

Church Turing Thesis :

Turing machine is defined as an abstract representation of a computing device such as hardware in computers. Alan Turing proposed Logical Computing Machines (LCMs), i.e. Turing's expressions for Turing Machines. This was done to define algorithms properly. So, Church made a mechanical method named as 'M' for manipulation of strings by using logic and mathematics.

This method M must pass the following statements:

- Number of instructions in M must be finite.

- Output should be produced after performing finite number of steps.
- It should not be imaginary, i.e. can be made in real life.
- It should not require any complex understanding.

Using these statements Church proposed a hypothesis called **Church's Turing thesis** that can be stated as: "The assumption that the intuitive notion of computable functions can be identified with partial recursive functions."

In 1930, this statement was first formulated by Alonzo Church and is usually referred to as Church's thesis, or the Church-Turing thesis. However, this hypothesis cannot be proved.

The recursive functions can be computable after taking following assumptions:

1. Each and every function must be computable.
2. Let 'F' be the computable function and after performing some elementary operations to 'F', it will transform a new function 'G' then this function 'G' automatically becomes the computable function.
3. If any functions that follow above two assumptions must be states as computable function.

Recursive and Recursive Enumerable Languages in TOC

Recursive Enumerable (RE) or Type -0 Language

RE languages or type-0 languages are generated by type-0 grammars. An RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages.

Recursive Language (REC)

A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. e.g.; $L = \{a^n b^n c^n \mid n \geq 1\}$ is recursive because we can construct a turing machine which will move to final state if the string is of the form $a^n b^n c^n$ else move to non-final state. So the TM will always halt in this case. REC languages are also called as Turing decidable languages. The relationship between RE and REC languages can be shown in Figure 1.

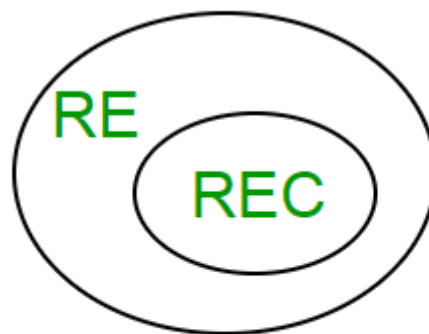


Figure 1

Closure Properties of Recursive Languages

- **Union:** If L_1 and L_2 are two recursive languages, their union $L_1 \cup L_2$ will also be recursive because if TM halts for L_1 and halts for L_2 , it will also halt for $L_1 \cup L_2$.
- **Concatenation:** If L_1 and L_2 are two recursive languages, their concatenation $L_1.L_2$ will also be recursive. For Example:
 - $L_1 = \{a^n b^n c^n \mid n \geq 0\}$
 - $L_2 = \{d^m e^m f^m \mid m \geq 0\}$

Theory of Computation and Compilers Unit – 8

- $L3 = L1.L2$
- $= \{a^n b^n c^n d^m e^m f^m \mid m \geq 0 \text{ and } n \geq 0\}$ is also recursive.

$L1$ says n no. of a 's followed by n no. of b 's followed by n no. of c 's. $L2$ says m no. of d 's followed by m no. of e 's followed by m no. of f 's. Their concatenation first matches no. of a 's, b 's and c 's and then matches no. of d 's, e 's and f 's. So it can be decided by TM.

- **Kleene Closure:** If $L1$ is recursive, its kleene closure $L1^*$ will also be recursive. For Example:

$$L1 = \{a^n b^n c^n \mid n \geq 0\}$$

$$L1^* = \{a^n b^n c^n \mid n \geq 0\}^* \text{ is also recursive.}$$

- **Intersection and complement:** If $L1$ and $L2$ are two recursive languages, their intersection $L1 \cap L2$ will also be recursive. For Example:
- $L1 = \{a^n b^n c^n d^m \mid n \geq 0 \text{ and } m \geq 0\}$
- $L2 = \{a^n b^n c^n d^n \mid n \geq 0 \text{ and } m \geq 0\}$
- $L3 = L1 \cap L2$
- $= \{a^n b^n c^n d^n \mid n \geq 0\}$ will be recursive.

$L1$ says n no. of a 's followed by n no. of b 's followed by n no. of c 's and then any no. of d 's. $L2$ says any no. of a 's followed by n no. of b 's followed by n no. of c 's followed by n no. of d 's. Their intersection says n no. of a 's followed by n no. of b 's followed by n no. of c 's followed by n no. of d 's. So it can be decided by turing machine, hence recursive.

Similarly, complement of recursive language $L1$ which is $\Sigma^* - L1$, will also be recursive.

Note: As opposed to REC languages, RE languages are not closed under complementon which means complement of RE language need not be RE.

Context-Sensitive Grammar –

A Context-sensitive grammar is an Unrestricted grammar in which all the productions are of form –

$$\alpha \rightarrow \beta$$

where $\alpha, \beta \in (V \cup T)^+$ and $|\alpha| \leq |\beta|$

Where α and β are strings of non-terminals and terminals.

Context-sensitive grammars are **more powerful** than context-free grammars because there are some languages that can be described by CSG but not by context-free grammars and CSL are less powerful than Unrestricted grammar. That's why context-sensitive grammars are positioned between context-free and unrestricted grammars in the Chomsky hierarchy.



Context-sensitive grammar has 4-tuples. $G = \{N, \Sigma, P, S\}$, Where

N = Set of non-terminal symbols

Σ = Set of terminal symbols

S = Start symbol of the production

P = Finite set of productions

All rules in P are of the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$

Context-sensitive Language

The language that can be defined by context-sensitive grammar is called CSL.

Properties of CSL are :

- Union, intersection and concatenation of two context-sensitive languages is context-sensitive.
- Complement of a context-sensitive language is context-sensitive.

Example –

Consider the following CSG.

$S \rightarrow abc/aAbc$

$Ab \rightarrow bA$

$Ac \rightarrow Bbcc$

$bB \rightarrow Bb$

$aB \rightarrow aa/aaA$

What is the language generated by this grammar?

Solution:

$S \rightarrow aAbc$

$\rightarrow abAc$

$\rightarrow abBbcc$

$\rightarrow aBbbcc$

$\rightarrow aaAbbcc$

$\rightarrow aabAbcc$

$\rightarrow aabbAcc$

$\rightarrow aabbBbcc$

$\rightarrow aabBbbcc$

$\rightarrow aaBbbbcc$

→ aaabbbccc

The language generated by this grammar is $\{a^n b^n c^n \mid n \geq 1\}$.

Chomsky Classification of Grammars

Undecidable Problem about Turing Machine

In this section, we will discuss all the undecidable problems regarding turing machine. The reduction is used to prove whether given language is desirable or not. In this section, we will understand the concept of reduction first and then we will see an important theorem in this regard.

Reduction

Reduction is a technique in which if a problem P1 is reduced to a problem P2 then any solution of P2 solves P1. In general, if we have an algorithm to convert an instance of a problem P1 to an instance of a problem P2 that have the same answer then it is called as P1 reduced P2. Hence if P1 is not recursive then P2 is also not recursive. Similarly, if P1 is not recursively enumerable then P2 also is not recursively enumerable.

Theorem: if P1 is reduced to P2 then

1. If P1 is undecidable, then P2 is also undecidable.
2. If P1 is non-RE, then P2 is also non-RE.

Proof:

1. Consider an instance w of P1. Then construct an algorithm such that the algorithm takes instance w as input and converts it into another instance x of P2. Then apply that algorithm to check whether x is in P2. If the algorithm answer 'yes' then that means x is in P2, similarly we can also say that w is in P1. Since we have obtained P2 after reduction of P1. Similarly if algorithm answer 'no' then x is not in P2, that also means w is not in P1. This proves that if P1 is undecidable, then P1 is also undecidable.

2. We assume that P1 is non-RE but P2 is RE. Now construct an algorithm to reduce P1 to P2, but by this algorithm, P2 will be recognized. That means there will be a Turing machine that says 'yes' if the input is P2 but may or may not halt for the input which is not in P2. As we know that one can convert an instance of w in P1 to an instance x in P2. Then apply a TM to check whether x is in P2. If x is accepted that also means w is accepted. This procedure describes a TM whose language is P1 if w is in P1 then x is also in P2 and if w is not in P1 then x is also not in P2. This proves that if P1 is non-RE then P2 is also non-RE.

Empty and non empty languages:

There are two types of languages empty and non empty language. Let L_e denotes an empty language, and L_{ne} denotes non empty language. Let w be a binary string, and M_i be a TM. If $L(M_i) = \Phi$ then M_i does not accept input then w is in L_e .

Similarly, if $L(M_i)$ is not the empty language, then w is in L_{ne} . Thus we can say that

$$L_e = \{M \mid L(M) = \Phi\}$$

$$L_{ne} = \{M \mid L(M) \neq \Phi\}$$

Both L_e and L_{ne} are the complement of one another.

Chomsky Hierarchy

Chomsky Hierarchy represents the class of languages that are accepted by the different machine. The category of language in Chomsky's Hierarchy is as given below:

1. Type 0 known as Unrestricted Grammar.
2. Type 1 known as Context Sensitive Grammar.
3. Type 2 known as Context Free Grammar.
4. Type 3 Regular Grammar.

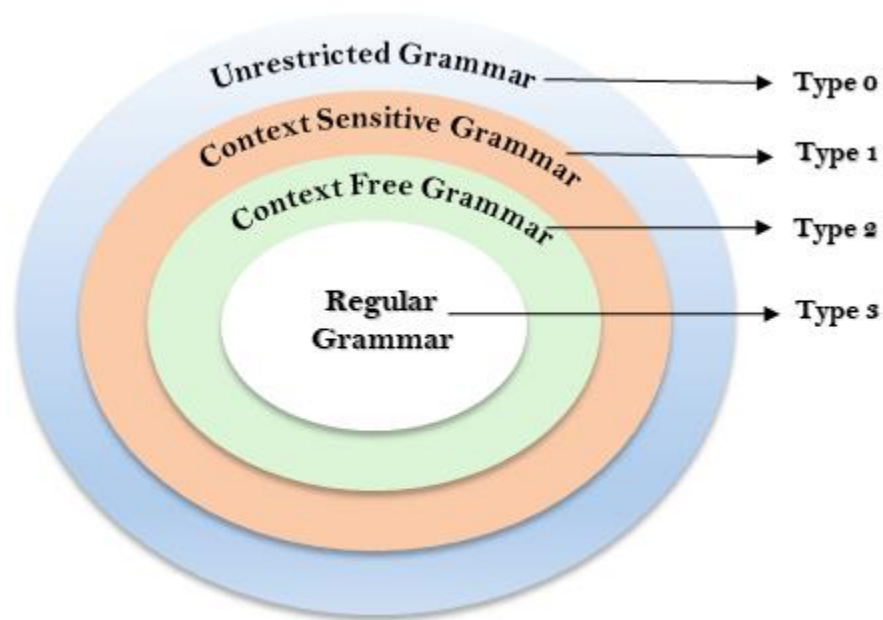


Fig: Chomsky Hierarchy

This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc.

Type 0 Grammar:

Type 0 grammar is known as Unrestricted grammar. There is no restriction on the grammar rules of these types of languages. These languages can be efficiently modeled by Turing machines.

For example:

1. $bAa \rightarrow aa$
2. $S \rightarrow s$

Type 1 Grammar:

Type 1 grammar is known as Context Sensitive Grammar. The context sensitive grammar is used to represent context sensitive language. The context sensitive grammar follows the following rules:

Theory of Computation and Compilers Unit – 8

- The context sensitive grammar may have more than one symbol on the left hand side of their production rules.
- The number of symbols on the left-hand side must not exceed the number of symbols on the right-hand side.
- The rule of the form $A \rightarrow \epsilon$ is not allowed unless A is a start symbol. It does not occur on the right-hand side of any rule.
- The Type 1 grammar should be Type 0. In type 1, Production is in the form of $V \rightarrow T$

Where the count of symbol in V is less than or equal to T.

For example:

1. $S \rightarrow AT$
2. $T \rightarrow xy$
3. $A \rightarrow a$

Type 2 Grammar:

Type 2 Grammar is known as Context Free Grammar. Context free languages are the languages which can be represented by the context free grammar (CFG). Type 2 should be type 1. The production rule is of the form

1. $A \rightarrow \alpha$

Where A is any single non-terminal and α is any combination of terminals and non-terminals.

For example:

1. $A \rightarrow aBb$
2. $A \rightarrow b$

3. $B \rightarrow a$

Type 3 Grammar:

Type 3 Grammar is known as Regular Grammar. Regular languages are those languages which can be described using regular expressions. These languages can be modeled by NFA or DFA.

Type 3 is most restricted form of grammar. The Type 3 grammar should be Type 2 and Type 1. Type 3 should be in the form of

1. $V \rightarrow T^*V / T^*$

For example:

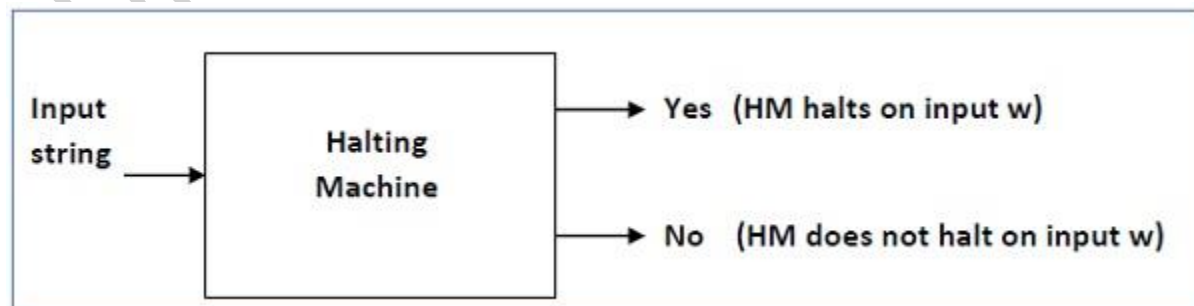
1. $A \rightarrow xy$

Turing Machine Halting Problem

Input – A Turing machine and an input string **w**.

Problem – Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.

Proof – At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine

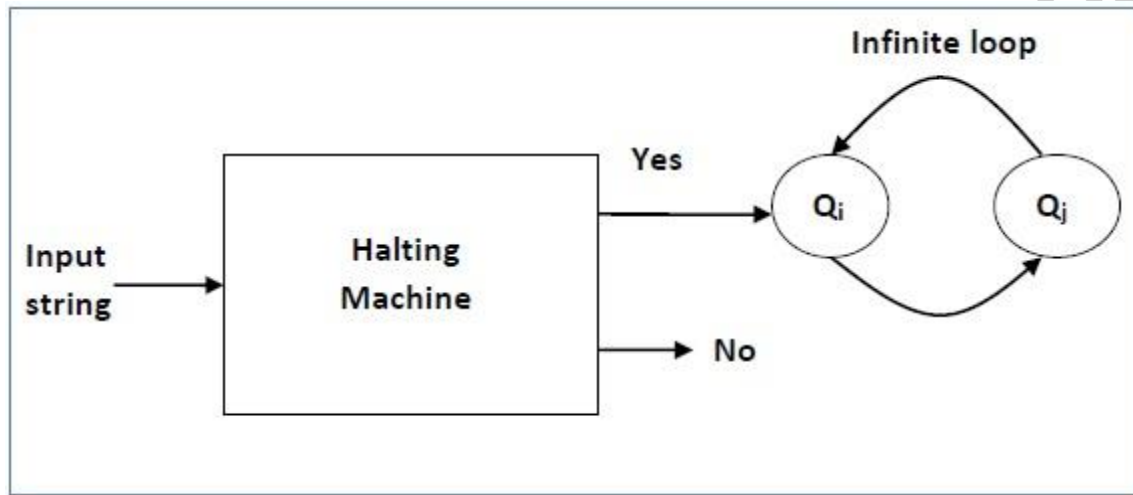


Theory of Computation and Compilers Unit – 8

Now we will design an **inverted halting machine (HM)'** as –

- If **H** returns YES, then loop forever.
- If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine' –



Further, a machine **(HM)₂** which input itself is constructed as follows –

- If **(HM)₂** halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable**.

Halting problem in theory of computation

Some important terms:

- **Computability theory –**
The branch of theory of computation that studies which problems are computationally solvable using different model. In computer science, the computational complexity, or simply complexity of an algorithm is the amount of resources required for running it.

- **Decision problems –**

A decision problem has only two possible outputs (yes or no) on any input. In computability theory and computational complexity theory, a decision problem is a problem that can be posed as a yes-no question of the input values. Like is there any solution to a particular problem? The answer would be either a yes or no. A decision problem is any arbitrary yes/no question on an infinite set of inputs.

- **Turing machine –**

A Turing machine is a mathematical model of computation. A Turing machine is a general example of a CPU that controls all data manipulation done by a computer. Turing machine can be halting as well as non halting and it depends on algorithm and input associated with the algorithm.

Now, Halting problem:

The Halting problem – Given a program/algorithm will ever halt or not?

Halting means that the program on certain input will accept it and halt or reject it and halt and it would never go into an infinite loop. Basically halting means terminating. So can we have an algorithm that will tell that the given program will halt or not. In terms of Turing machine, will it terminate when run on some machine with some particular given input string.

The answer is no we cannot design a generalized algorithm which can appropriately say that given a program will ever halt or not?

The only way is to run the program and check whether it halts or not.

We can reframe the halting problem question in such a way also: Given a program written in some programming language(c/c++/java) will it ever get into an infinite loop(loop never stops) or will it always terminate(halt)?

This is an undecidable problem because we cannot have an algorithm which will tell us whether a given program will halt or not in a generalized way i.e by having specific program/algorithm. In general we can't always know that's why we can't have a general algorithm. The best possible way is to run the program and see

whether it halts or not. In this way for many programs we can see that it will sometimes loop and always halt.

Proof by Contradiction –

Problem statement: Can we design a machine which if given a program can find out if that program will always halt or not halt on a particular input?

Solution: Let us assume that we can design that kind of machine called as $HM(P, I)$ where HM is the machine/program, P is the program and I is the input. On taking input the both arguments the machine HM will tell that the program P either halts or not.

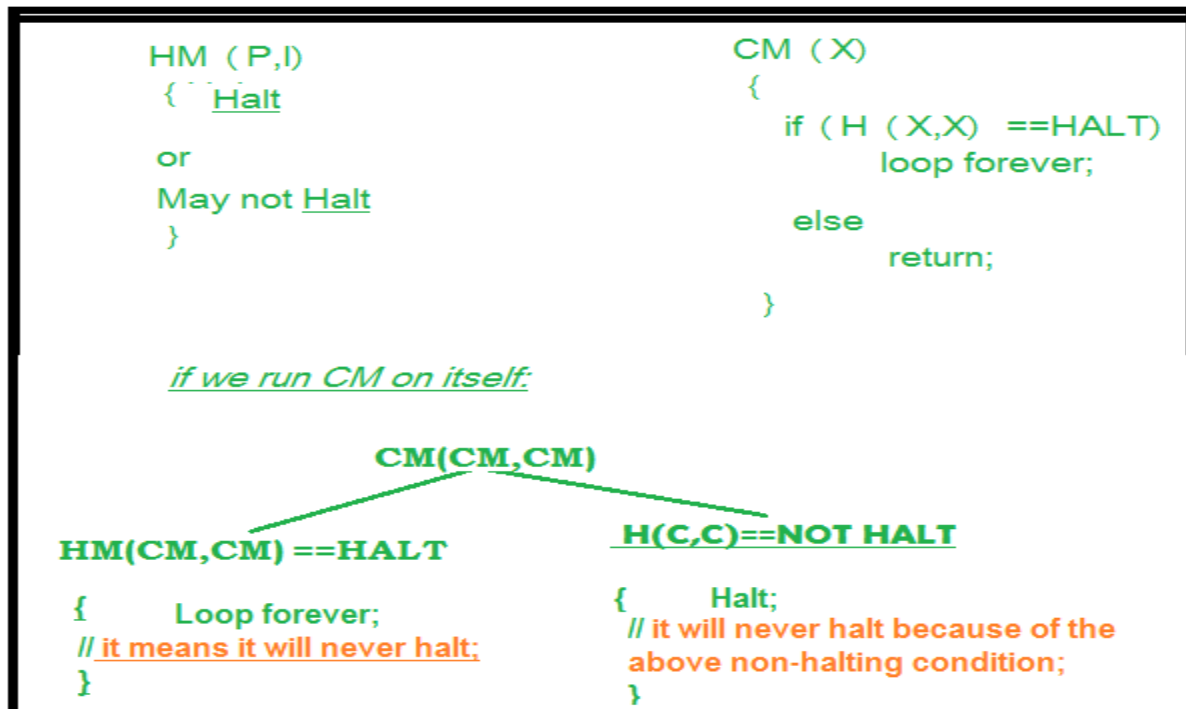
If we can design such a program this allows us to write another program we call this program $CM(X)$ where X is any program (taken as argument) and according to the definition of the program $CM(X)$ shown in the figure.

$HM(P, I)$	$CM(X)$
<pre>{ <u>Halt</u></pre>	<pre>{</pre>
or	<pre> if($HM(X, X) == \text{Halt}$)</pre>
May not <u>Halt</u>	<pre> loop forever;</pre>
}	<pre> else</pre>
	<pre> return;</pre>
	<pre>}</pre>

In the program $CM(X)$ we call the function $HM(X)$, which we have already defined and to $HM()$ we pass the arguments (X, X) , according to the definition of $HM()$ it can take two arguments i.e one is program and another is the input. Now in the second program we pass X as a program and X as input to the function $HM()$. We know that the program $HM()$ gives two output either “Halt” or “Not Halt”. But in case second program, when $HM(X, X)$ will halt loop body tells to go in loop and when it doesn’t halt that means loop, it is asked to return.

Theory of Computation and Compilers Unit – 8

Now we take one more situation where the program CM is passed to CM() function as an argument. Then there would be some impossibility, i.e., a condition arises which is not possible.



It is impossible for outer function to halt if its code (inner body) is in loop and also it is impossible for outer non halting function to halt even after its inner code is halting. So the both condition is non halting for CM machine/program even we had assumed in the beginning that it would halt. So this is the contradiction and we can say that our assumption was wrong and this problem, i.e., halting problem is undecidable.

Identifying languages (or problems*) as decidable, undecidable or partially decidable is a very common question in GATE. With correct knowledge and ample experience, this question becomes very easy to solve.

some definitions:-

Decidable language -A decision problem P is said to be decidable (i.e., have an algorithm)

Theory of Computation and Compilers Unit – 8

if the language L of all yes instances to P is decidable.

Example- (I) (Acceptance problem for DFA) Given a DFA does it accept a given word?

(II) (Emptiness problem for DFA) Given a DFA does it accept any word?

(III) (Equivalence problem for DFA) Given two DFAs, do they accept the same language?

Undecidable language -- A decision problem P is said to be undecidable if the language L of all

yes instances to P is not decidable or a language is undecidable if it is not decidable. An undecidable language maybe a partially decidable language or something else but not decidable. If a language is not even partially decidable , then there exists no Turing machine for that language.

Partially decidable or Semi-Decidable Language -- A decision problem P is said to be semi-decidable (i.e., have a semi-algorithm) if the language L of all yes instances to P is RE. A language 'L' is partially decidable if 'L' is a RE but not REC language.

Recursive language(REC) – A language 'L' is said to be recursive if there exists a Turing machine which will accept all the strings in 'L' and reject all the strings not in 'L'. The Turing machine will halt every time and give an answer(accepted or rejected) for each and every string input. A language 'L' is decidable if it is a recursive language. All decidable languages are recursive languages and vice-versa.

Recursively enumerable language(RE) – A language 'L' is said to be a recursively enumerable language if there exists a Turing machine which will accept (and therefore halt) for all the input strings which are in 'L' but may or may not halt for all input strings which are not in 'L'. By definition , all REC languages are also RE languages but not all RE languages are REC languages.

Theory of computation | Computable and non-computable problems

Computable Problems –

We may be familiar with many problems (or functions) that are computable (or decidable), meaning there exists some algorithm that computes an answer (or output) to any instance of the problem (or for any input to the function) in a finite number of simple steps. A simple example is the integer increment operation:

$$f(x) = x + 1$$

It should be intuitive that given any integer x , we can compute $x + 1$ in a finite number of steps. Since x is finite, it may be represented by a finite string of digits. Using the addition method (or algorithm) we all learned in school, we can clearly compute another string of digits representing the integer equivalent to $x + 1$.

Yet there are also problems and functions that are non-computable (or undecidable or uncomputable), meaning that there exists no algorithm that can compute an answer or output for all inputs in a finite number of simple steps. (Undecidable simply means non-computable in the context of a decision problem, whose answer (or output) is either “true” or “false”).

Non-Computable Problems –

A non-computable is a problem for which there is no algorithm that can be used to solve it. Most famous example of a non-computability (or undecidability) is the **Halting Problem**. Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes).

The alternative is that it runs forever without halting. The halting problem is about seeing if a machine will ever come to a halt when a certain input is given to it or if it will finish running. This input itself can be something that keeps calling itself forever which means that it will cause the program to run forever.

Other example of an uncomputable problem is: determining whether a computer program loops forever on some input. You can replace “computer program” by “Turing machine or algorithm” if you know about Turing machine.

Proving Computability or Non-Computability –

We can show that a problem is computable by describing a procedure and proving that the procedure always terminates and always produces the correct answer. It is enough to provide a convincing argument that such a procedure exists finding the actual procedure is not necessary (but often helps to make the argument more convincing).

To show that a problem is not computable, we need to show that no algorithm exists that solves the problem. Since there are an infinite number of possible procedures, we cannot just list all possible procedures and show why each one does not solve the problem. Instead, we need to construct an argument showing that if there were such an algorithm it would lead to a contradiction.

The core of our argument is based on knowing the Halting Problem is noncomputable. If a solution to some new problem P could be used to solve the Halting Problem, then we know that P is also noncomputable. That is, no algorithm exists that can solve P since if such an algorithm exists it could be used to also solve the Halting Problem which we already know is impossible. The proof technique where we show that a solution for some problem P can be used to solve a different problem Q is known as a **reduction**. A problem Q is **reducible** to a problem P if a solution to P could be used to solve Q. This means that problem Q is no harder than problem P, since a solution to problem Q leads to a solution to problem P.

Some Examples On Computable Problems –

These are four simple examples of computable problem:

1. Computing the greatest common divisor of a pair of integers.
2. Computing the least common multiple of a pair of integers.
3. Finding the shortest path between a pair of nodes in a finite graph.
4. Determining whether a propositional formula is a tautology.

Some Examples On Computable Problems – State Entry Problem.

Consider the problem of determining if a string 'w' is given to some Turing machine 'M' will it enter some state 'q' (where q belongs to set of all states in Turing machine M and string w is not equal to empty string). Is it computable or non-computable?

Explanation –

We show the State Entry Problem is non-computable by showing that it is as hard as the Halting Problem, which we already know is non-computable.

State Entry Problem is asking us on given string w if we start from initial state of Turing machine will it reach to a state q. Now this state entry problem can be converted to halting problem. Halting problem is whether our Turing machine ever halts and state entry problem is asking same thing whether this Turing machine halts at some state q if we give string w as input to the Turing machine M. So the state entry problem is non-computable as we converted it to halting problem which we already know is non-computable problem. So in this way we can prove non-computability.

Computational Complexity

Computational complexity is a computer science concept that focuses on the amount of computing resources needed for particular kinds of tasks. In computational complexity theory, researchers assess the kinds of resources that will be needed for a given type or class of task in order to classify different kinds of tasks into various levels of complexity.

Explanation

Although computational complexity is in some ways similar to the analysis of algorithms, it is essentially its own branch of mathematical theory. Some think of this approach as a measurement of how much work it would take to solve a particular problem or to achieve a particular task. Different kinds of analysts use computational complexity research to find which parts of a task may be most difficult for a computing system, or to figure out how to most efficiently complete

some project. Although some developers might consider computational complexity to be irrelevant to their work, others have pointed out that successfully changing tasks or algorithms from a higher complexity class to a lower complexity class can make them work much better. Programmers and developers who use computational complexity theory on items like nested loops, logic trees or other kinds of rhythms can build more efficient systems with a better understanding of how to create less resource-hungry processes.

Computation is the process of calculating something by mathematical or logical methods. Execution of an algorithm is a well-known example of computation.

In order to perform any sort of computation, we will need energy. This energy can be directly related to the resources that we spend to perform computation. In computing these resources are called System Resources. The most fundamental system resources that we can imagine are processor and memory as what a computer basically does is it performs calculations and remember results.

Since computation is a term which can be used outside the scope of a physical computer, we will relate the processor use to time and memory use to space.

In an ideal world we would like to perform computation without using any resources.

There is an inherent difficulty associated with any computational problem. But we can solve each problem with different solutions (or algorithms). So how can one decide the which is the best solution?

This is where **computational complexity** comes into picture. As per the computational complexity theory

A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used.

The amount of resources required for executing a particular (computation or) algorithm is the computational complexity of that algorithm. In general, when we

Theory of Computation and Compilers Unit – 8

talk about complexity we are talking about time complexity and space complexity. Of which time complexity is the one which is analysed the most.

With this in mind, we will be able to determine which algorithm is best for solving a problem, since we can estimate the complexity associated with each and every solution. Most importantly how the algorithm works as the number of inputs increases. The question is how the algorithm scales when the amount of input increases.

In most cases we will be interested in the **worst-case** scenario for analyzing complexity for any algorithm it is denoted by **Big O** notation. Some times represented as just '**O**'.

The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n .

The **best-case** complexity of an algorithm is represented by **Big- Ω (Big-Omega) notation**

The best-case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n .

Generally, when we talk about complexity we are talking about worst case complexity.

The computational complexity of a problem is the minimum of the complexities of all possible algorithms for this problem

In order to grasp more about this, we need to use some mathematical terms.

' $f(n)$ ' is the computational complexity for an algorithm where ' n ' is the number of data-set used by the algorithm. A dataset can be considered as a fundamental unit on top of which computation is done. So if I have an algorithm which sums ' n ' numbers, the number will be the data-set here.

Theory of Computation and Compilers Unit – 8

With a considerably large data set, we will not bother about the precise value of $f(n)$ we will take an approximation about what the value of $f(n)$ tends to.

$$\text{Say, } f(n) = n^3 + 3n$$

if the value of 'n' is considerably high (say $n = 1,000,000,000$) then

$$f(n) \approx n^3$$

So the computational complexity for this particular algorithm will be in polynomial time.

Following is a list of most commonly used complexities in increasing order.

$O(1)$	constant time
$O(\log n)$	logarithmic time
$O(n)$	linear time
$O(n \log n)$	linearithmic time
$O(n^2)$	quadratic time
$O(n^c)$	polynomial time
$O(c^n)$	exponential time
$O(n!)$	factorial time
$O(\infty)$	infinite time

Most widely used complexities for understanding the concept of computational complexities are

P -> Problems that are solvable in polynomial time .

EXP -> Problems that are solvable in exponential time.

R -> Problems that are solvable in finite time.

So the time complexities can be represented as follows

Unsolvable Problems

This means that every problem that can be solved in polynomial time can be solved in exponential time but the inverse is not true. Most of the interesting puzzles that we solve day-to-day are in exponential complexity space. For example Chess: given any state of a chess board and the next moving colour, we will be able to predict who will win in exponential complexity.

In Computational Complexity theory the most commonly used problems are decision problems, in fact most of the optimisation problems can be converted into decision problems.

So the question is how can we prove that most problems can not be solved in finite time.

A **decision problem** is a problem that can be posed as a yes-no question of the input values. And most problems can be represented as a decision problem.

In order to solve a problem we need an algorithm, a program is the best representation of an algorithm.

Any program can be represented in binary format. So a program is a binary string which in turn can be represented by a natural number. In other words a program is a big integer.

The solutions to all the problems can be represented as a set of natural numbers.

What about a decision problem? How can we express that?

A decision problem can be expressed as a function, $f(x)$, gives either 0 or 1. That is either true or false.

Here we can see that if we start putting values for x , then we will find result space as an infinite set of binary data. if

$$f(0) = 1$$

$$f(1) = 1$$

$$f(2) = 0$$

$$f(3) = 0$$

The solution space will be 1100.... which is an infinite set of binary. This can be expressed as a set of real numbers between 0 and 1 if we put a binary point right at the beginning.

List of all the problems can be represented as a set of real numbers.

By Cantor's Diagonal Argument, we can prove that the set of all real numbers is much larger than a set of all natural numbers.

Which, in turn, concludes that the number of solutions (or programs) out there in the world is much lesser than the number of problems out there.

That means most problems in computation are unsolvable.

This is by mathematical definition, but most of the problems the human brain can think of are solvable. Yes, lucky us.

Speaking about lucky, there are some problems in EXP (exponential complexity space) which can be solved in P (polynomial complexity space) with a "lucky" algorithm. This class of problems are defined as NP (Nondeterministic Polynomial) complexity problems..

"Without complexity, there's no intensity."— Alexandra Adornetto

Time complexity

Time complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input.

In other words, time complexity is essentially efficiency, or how long a program function takes to process a given input.

Explanation

Time complexity is simply a measure of the time it takes for a function or expression to complete its task, as well as the name of the process to measure that time. It can be applied to almost any algorithm or function but is more useful for recursive functions. There is little point in measuring time complexity for applications such as fetching the username and password from a database for comparison or simply saving data whether it is 20 ms or 5 ms; that would be more in the line of access time. It has nothing to do with caring about its execution time, but rather that the difference is negligible. However, if there is a recursive function that may be called multiple times, determining and understanding the source of its time complexity may help shorten the overall processing time from, say, 600 ms to 100 ms.

Time complexity is expressed typically in the "big O notation," but there are other notations. This is a mathematical representation of the upper limit of the scaling factor for an algorithm and is written as $O(Nn)$, with "N" being the number of inputs and "n" being the number of looping expressions. For example, we have the algorithm:

```
numbers[] = {5,6,10,11,2};  
foreach (number as number1)  
{  
    foreach(number as number2)  
    {  
        statements;  
    }  
}
```

Theory of Computation and Compilers Unit – 8

There are five inputs in the "numbers" array, and the "foreach" loop is repeated twice. Therefore, exponential growth in processing time occurs as the number of inputs and the number of loops grow.

Space complexity

Space complexity in algorithm development is a metric for how much storage space the algorithm needs in relation to its inputs. This measurement is extremely useful in some kinds of programming evaluations as engineers, coders and other scientists look at how a particular algorithm works.

Explanation

Professionals may use concrete equations to show an algorithm's space complexity where N equals inputs. Different forms of presentation such as "big O " notation may be used.

One reason that it is important to estimate the space complexity of an algorithm, the space it needs relative to inputs, is that some algorithms are designed with particular limitations. Some are designed with a cap on total storage space use, which can result in rough or imprecise results. Others are made to enforce precise results regardless of the space used.

Space complexity is a straightforward way for professionals to look at the footprint of an algorithm. For instance, engineers may add up memory to store program instructions, memory for variable values, and other types of central or auxiliary memory to get a total number of all of the memory that the program uses with a given number of inputs. This also helps professionals to plan for resource needs in operation.

Post Correspondence Problem

In this section, we will discuss the undecidability of string and not of Turing machines. The undecidability of the string is determined with the help of Post's Correspondence Problem (PCP). Let us define the PCP.

Theory of Computation and Compilers Unit – 8

"The Post's correspondence problem consists of two lists of string that are of equal length over the input. The two lists are $A = w_1, w_2, w_3, \dots, w_n$ and $B = x_1, x_2, x_3, \dots, x_n$ then there exists a non empty set of integers $i_1, i_2, i_3, \dots$, in such that, $w_{i_1} w_{i_2} w_{i_3} \dots w_{i_n} = x_{i_1} x_{i_2} x_{i_3} \dots x_{i_n}$ "

To solve the post correspondence problem we try all the combinations of $i_1, i_2, i_3, \dots$, in to find the $w_1 = x_1$ then we say that PCP has a solution.

Example 1:

Consider the correspondence system as given below

$A = (b, bab^3, ba)$ and $B = (b^3, ba, a)$. The input set is $\Sigma = \{0, 1\}$. Find the solution.

Solution:

A solution is 2, 1, 1, 3. That means $w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3$

The constructed string from both lists is bab^3b^3a .

w_2	w_1	w_1	w_3
bab^3	b	b	ba
x_2	x_1	x_1	x_3
ba	b^3	b^3	a

Example 2:

Does PCP with two lists $x = (b, a, aba, bb)$ and $y = (ba, ba, ab, b)$ have a solution?

Theory of Computation and Compilers Unit – 8

Solution: Now we have to find out such a sequence that strings formed by x and y are identical. Such a sequence is 1, 2, 1, 3, 3, 4. Hence from x and y list

1	2	1	3	3	4		1	2	1	3	3	4
b	a	b	aba	aba	bb	=	ba	ba	ba	ab	ab	b

Example 3:

Obtain the solution for the following system of posts correspondence problem. $A = \{100, 0, 1\}$, $B = \{1, 100, 00\}$

Solution: Consider the sequence 1, 3, 2. The string obtained from $A = babababb$. The string obtained from $B = bababbbb$. These two strings are not equal. Thus if we try various combination from both the sets to find the unique sequence, we could not get such a sequence. Hence there is no solution for this system.

Example 4:

Obtain the solution for the following system of posts correspondence problem, $X = \{100, 0, 1\}$, $Y = \{1, 100, 00\}$.

Solution: The solution is 1, 3, 1, 1, 3, 2, 2. The string is

$$X1X3X1X1X3X2X2 = 100 + 1 + 100 + 100 + 1 + 0 + 0 = 1001100100100$$

$$Y1Y3Y1Y1Y3Y2Y2 = 1 + 00 + 1 + 1 + 00 + 100 + 100 = 1001100100100$$

Tractable and Intractable Problems

In Computer Science, many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not. Hence, the problems can be categorized as follows –

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

- Finding the minimum number of colors needed to color a given graph.
- Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

- Whether a given graph can be colored by only 4-colors.
- Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer 'yes' for a specific input. A language is the totality of inputs for which the answer is Yes. Most of the algorithms discussed in the previous chapters are **polynomial time algorithms**.

For input size n , if worst-case time complexity of an algorithm is $O(n^k)$, where k is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows –

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where k is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size n in a time $O(p(n))$.

Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large n . Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of k .

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in $NP - P$ are intractable.

Theory of Computation and Compilers Unit – 8

It is not known whether $P = NP$. However, many problems are known in NP with the property that if they belong to P, then it can be proved that $P = NP$.

If $P \neq NP$, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it's easy to find a solution for the problem. The problem belongs to **NP**, if it's easy to check a solution that may have been very tedious to find.

Tractable Problem: a problem that is solvable by a polynomial-time algorithm.

The upper bound is polynomial.

Intractable Problem: a problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.

Here are examples of tractable problems (ones with known polynomial-time algorithms):

– Searching an unordered list

- Searching an ordered list
- Sorting a list
- Multiplication of integers (even though there's a gap)
- Finding a minimum spanning tree in a graph (even though there's a gap)

Here are examples of intractable problems (ones that have been proven to have no polynomial-time algorithm).

- Some of them require a non-polynomial amount of output, so they clearly will take a non-polynomial amount of time, e.g.:

* Towers of Hanoi: we can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2n - 1$.

* List all permutations (all possible orderings) of n numbers.

- Others have polynomial amounts of output, but still cannot be solved in polynomial time:

For an $n \times n$ draughts board with an arrangement of pieces, determine whether there is a winning strategy for White (i.e. a sequence of moves so that, no matter what Black does, White is guaranteed to win). We can prove that any algorithm that solves this problem must have a worst-case running time that is at least 2^n .

So we might think that problems can be neatly divided into these two classes. But this ignores 'gaps' between lower and upper bounds. Incredibly, there are problems for which the state of our knowledge is such that the gap spans this coarse division into tractable and intractable. So, in fact, there are three broad classes of problems:

- Problems with known polynomial-time algorithms.
- Problems that are provably intractable (proven to have no polynomial-time algorithm).
- Problems with no known polynomial-time algorithm but not yet proven to be intractable.

Natural Language Processing - Syntactic Analysis

Syntactic analysis or parsing or syntax analysis is the third phase of NLP. The purpose of this phase is to draw exact meaning, or you can say dictionary meaning from the text. Syntax analysis checks the text for meaningfulness comparing to the rules of formal grammar. For example, the sentence like "hot ice-cream" would be rejected by semantic analyzer.

In this sense, syntactic analysis or parsing may be defined as the process of analyzing the strings of symbols in natural language conforming to the rules of formal grammar. The origin of the word '**parsing**' is from Latin word '**pars**' which means '**part**'.

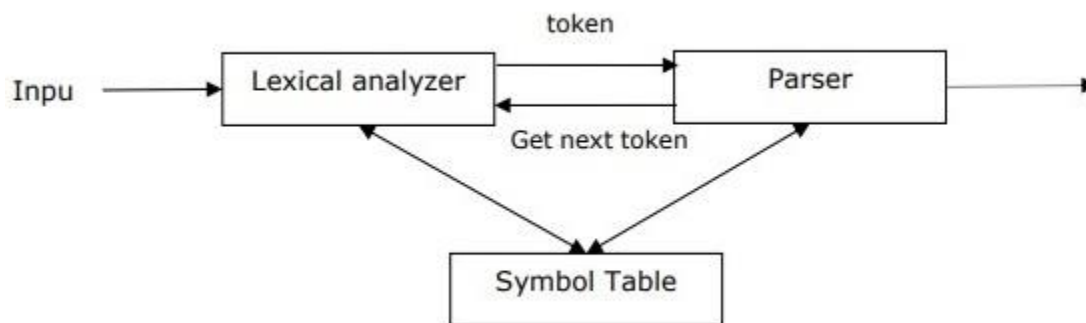
Parser

Theory of Computation and Compilers Unit – 8

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.

A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

It is used to implement the task of parsing. It may be defined as the software component designed for taking input data (text) and giving structural representation of the input after checking for correct syntax as per formal grammar. It also builds a data structure generally in the form of parse tree or abstract syntax tree or other hierarchical structure.



The main roles of the parser include –

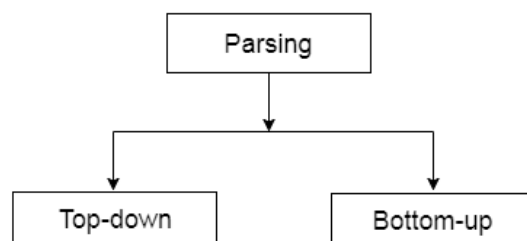
- To report any syntax error.
- To recover from commonly occurring error so that the processing of the remainder of program can be continued.
- To create parse tree.
- To create symbol table.
- To produce intermediate representations (IR).

Types of Parsing

Derivation divides parsing into the following two types –

Theory of Computation and Compilers Unit – 8

- Top-down Parsing
- Bottom-up Parsing

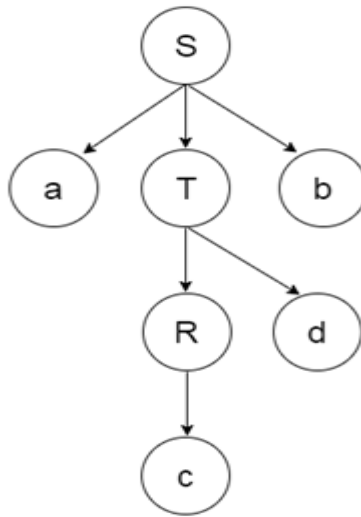


Top-down Parsing

In this kind of parsing, the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input. The most common form of topdown parsing uses recursive procedure to process the input. The main disadvantage of recursive descent parsing is backtracking.

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



Bottom-up Parsing

In this kind of parsing, the parser starts with the input symbol and tries to construct the parser tree up to the start symbol.

- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

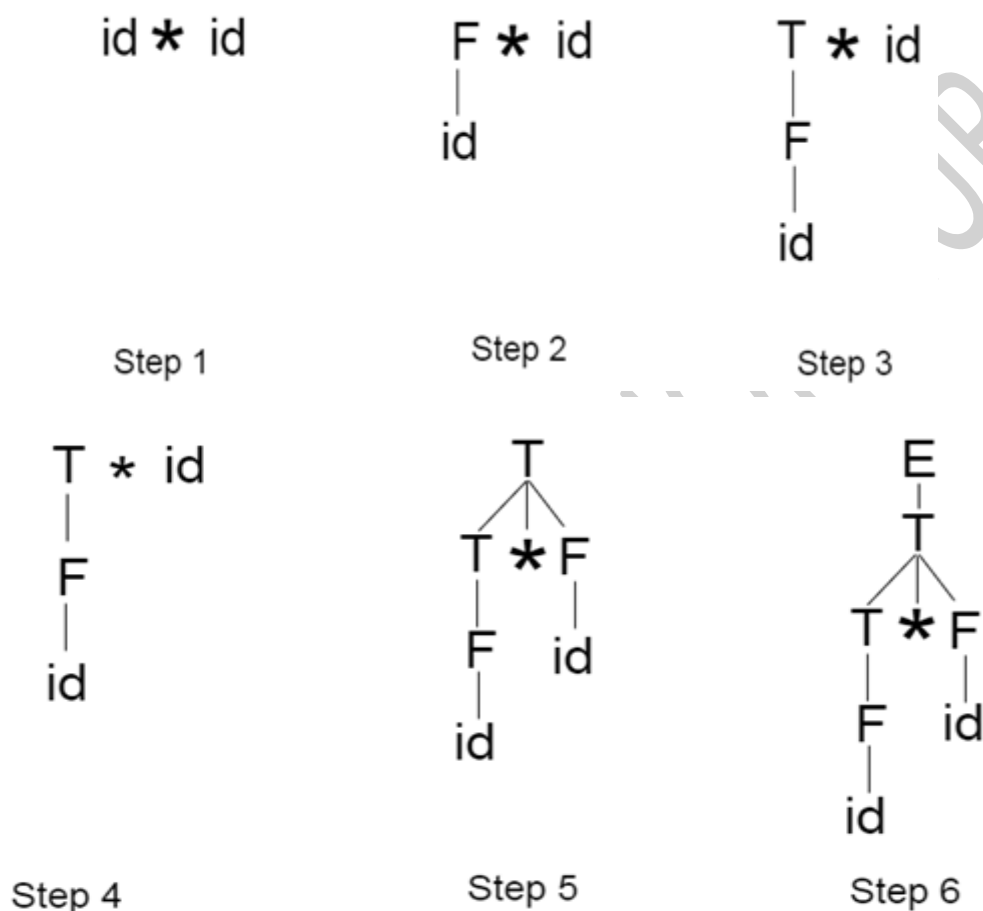
Example

Production

1. $E \rightarrow T$
2. $T \rightarrow T * F$
3. $T \rightarrow id$
4. $F \rightarrow T$

5. $F \rightarrow id$

Parse Tree representation of input string "id * id" is as follows:



Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing
 - a. LR(1)
 - b. SLR(1)
 - c. CLR (1)

d. LALR(1)

Concept of Derivation

In order to get the input string, we need a sequence of production rules. Derivation is a set of production rules. During parsing, we need to decide the non-terminal, which is to be replaced along with deciding the production rule with the help of which the non-terminal will be replaced.

Types of Derivation

In this section, we will learn about the two types of derivations, which can be used to decide which non-terminal to be replaced with production rule –

Left-most Derivation

In the left-most derivation, the sentential form of an input is scanned and replaced from the left to the right. The sentential form in this case is called the left-sentential form.

Right-most Derivation

In the right-most derivation, the sentential form of an input is scanned and replaced from right to left. The sentential form in this case is called the right-sentential form.

Concept of Parse Tree

It may be defined as the graphical depiction of a derivation. The start symbol of derivation serves as the root of the parse tree. In every parse tree, the leaf nodes are terminals and interior nodes are non-terminals. A property of parse tree is that in-order traversal will produce the original input string.

Concept of Grammar

Grammar is very essential and important to describe the syntactic structure of well-formed programs. In the literary sense, they denote syntactical rules for

conversation in natural languages. Linguistics have attempted to define grammars since the inception of natural languages like English, Hindi, etc.

The theory of formal languages is also applicable in the fields of Computer Science mainly in programming languages and data structure. For example, in 'C' language, the precise grammar rules state how functions are made from lists and statements.

A mathematical model of grammar was given by **Noam Chomsky** in 1956, which is effective for writing computer languages.

Mathematically, a grammar G can be formally written as a 4-tuple (N, T, S, P) where –

- N or V_N = set of non-terminal symbols, i.e., variables.
- T or Σ = set of terminal symbols.
- S = Start symbol where $S \in N$
- P denotes the Production rules for Terminals as well as Non-terminals. It has the form $\alpha \rightarrow \beta$, where α and β are strings on $V_N \cup \Sigma$ and least one symbol of α belongs to V_N

Phrase Structure or Constituency Grammar

Phrase structure grammar, introduced by Noam Chomsky, is based on the constituency relation. That is why it is also called constituency grammar. It is opposite to dependency grammar.

Example

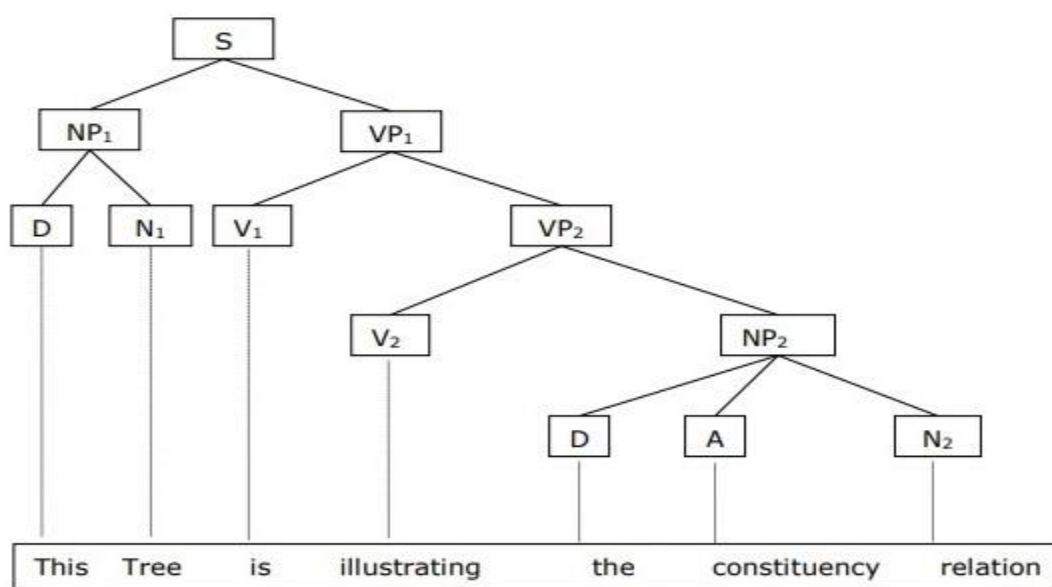
Before giving an example of constituency grammar, we need to know the fundamental points about constituency grammar and constituency relation.

- All the related frameworks view the sentence structure in terms of constituency relation.

Theory of Computation and Compilers Unit – 8

- The constituency relation is derived from the subject-predicate division of Latin as well as Greek grammar.
- The basic clause structure is understood in terms of **noun phrase NP** and **verb phrase VP**.

We can write the sentence “**This tree is illustrating the constituency relation**” as follows –



Dependency Grammar

It is opposite to the constituency grammar and based on dependency relation. It was introduced by Lucien Tesniere. Dependency grammar (DG) is opposite to the constituency grammar because it lacks phrasal nodes.

Example

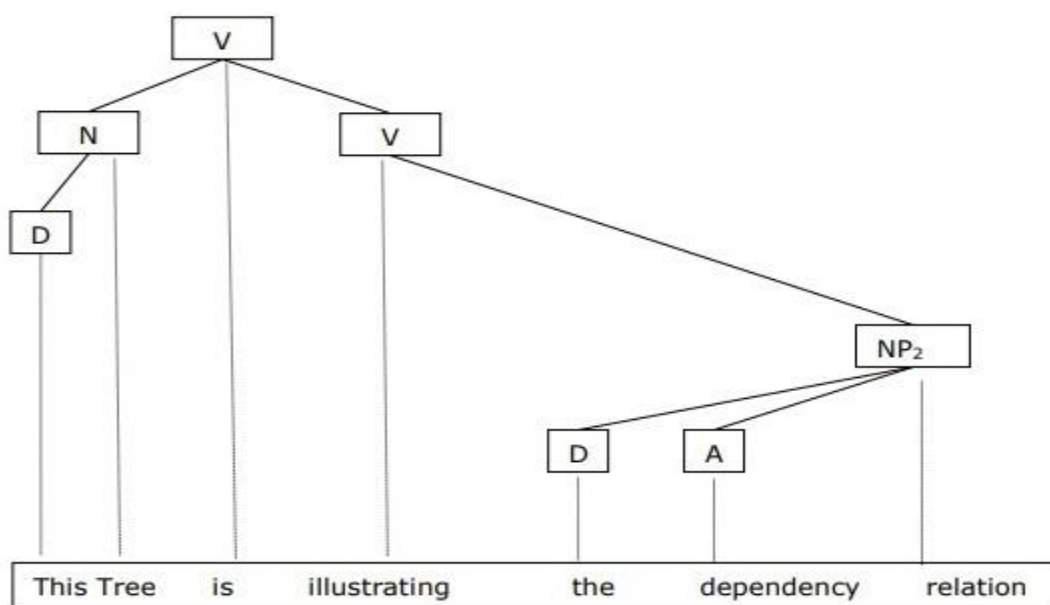
Before giving an example of Dependency grammar, we need to know the fundamental points about Dependency grammar and Dependency relation.

- In DG, the linguistic units, i.e., words are connected to each other by directed links.

Theory of Computation and Compilers Unit – 8

- The verb becomes the center of the clause structure.
- Every other syntactic units are connected to the verb in terms of directed link. These syntactic units are called **dependencies**.

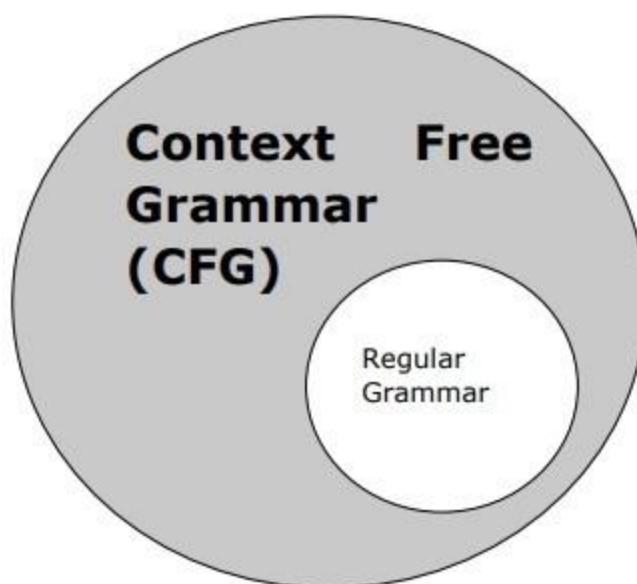
We can write the sentence “**This tree is illustrating the dependency relation**” as follows;



Parse tree that uses Constituency grammar is called constituency-based parse tree; and the parse trees that uses dependency grammar is called dependency-based parse tree.

Context Free Grammar

Context free grammar, also called CFG, is a notation for describing languages and a superset of Regular grammar. It can be seen in the following diagram –



Definition of CFG

CFG consists of finite set of grammar rules with the following four components –

Set of Non-terminals

It is denoted by V . The non-terminals are syntactic variables that denote the sets of strings, which further help defining the language, generated by the grammar.

Set of Terminals

It is also called tokens and defined by Σ . Strings are formed with the basic symbols of terminals.

Set of Productions

It is denoted by P . The set defines how the terminals and non-terminals can be combined. Every production(P) consists of non-terminals, an arrow, and terminals (the sequence of terminals). Non-terminals are called the left side of the production and terminals are called the right side of the production.

Start Symbol

Theory of Computation and Compilers Unit – 8

The production begins from the start symbol. It is denoted by symbol S. Non-terminal symbol is always designated as start symbol.

Operator precedence parsing

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has $a \in$.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

$a \succ b$ means that terminal "a" has the higher precedence than terminal "b".

$a \preceq b$ means that terminal "a" has the lower precedence than terminal "b".

$a \doteq b$ means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	(	)	id	\$
+	$\succ$	$\prec$	$\prec$	$\succ$	$\prec$	$\succ$
*	$\succ$	$\succ$	$\prec$	$\succ$	$\prec$	$\succ$
(	$\prec$	$\prec$	$\prec$	$\doteq$	$\prec$	X
)	$\succ$	$\succ$	X	$\succ$	X	$\succ$
id	$\succ$	$\succ$	X	$\succ$	X	$\succ$
\$	$\prec$	$\prec$	$\prec$	X	$\prec$	X

Parsing Action

- Both end of the given input string, add the \$ symbol.
- Now scan the input string from left right until the $\succ$ is encountered.

Theory of Computation and Compilers Unit – 8

- Scan towards left over all the equal precedence until the first left most $\leq$ is encountered.
- Everything between left most $\leq$ and right most $\geq$ is a handle.
- \$ on \$ means parsing is successful.

Example

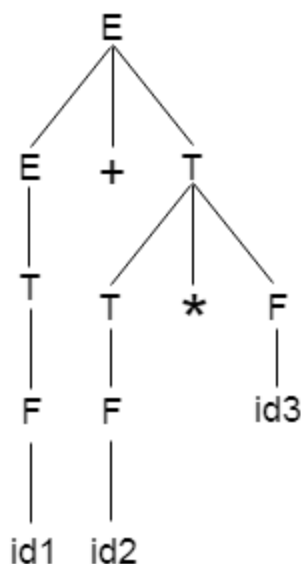
Grammar:

1. $E \rightarrow E+T/T$
2. $T \rightarrow T*F/F$
3. $F \rightarrow \text{id}$

Given string:

1. $w = \text{id} + \text{id} * \text{id}$

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table:

Theory of Computation and Compilers Unit – 8

	E	T	F	id	+	*	\$
E	X	X	X	X	$\doteq$	X	$\triangleright$
T	X	X	X	X	$\triangleright$	$\doteq$	$\triangleright$
F	X	X	X	X	$\triangleright$	$\triangleright$	$\triangleright$
id	X	X	X	X	$\triangleright$	$\triangleright$	$\triangleright$
+	X	$\doteq$	$\triangleleft$	$\triangleleft$	X	X	X
*	X	X	$\doteq$	$\triangleleft$	X	X	X
\$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	X	X	X

Now process the string with the help of the above precedence table:

$\$ \triangleleft \text{id1} \triangleright + \text{id2} * \text{id3} \$$

$\$ \triangleleft F \triangleright + \text{id2} * \text{id3} \$$

$\$ \triangleleft T \triangleright + \text{id2} * \text{id3} \$$

$\$ \triangleleft E \doteq + \triangleleft \text{id2} \triangleright * \text{id3} \$$

$\$ \triangleleft E \doteq + \triangleleft F \triangleright * \text{id3} \$$

$\$ \triangleleft E \doteq + \triangleleft T \doteq * \triangleleft \text{id3} \triangleright \$$

$\$ \triangleleft E \doteq + \triangleleft T \doteq * \doteq F \triangleright \$$

$\$ \triangleleft E \doteq + \doteq T \triangleright \$$

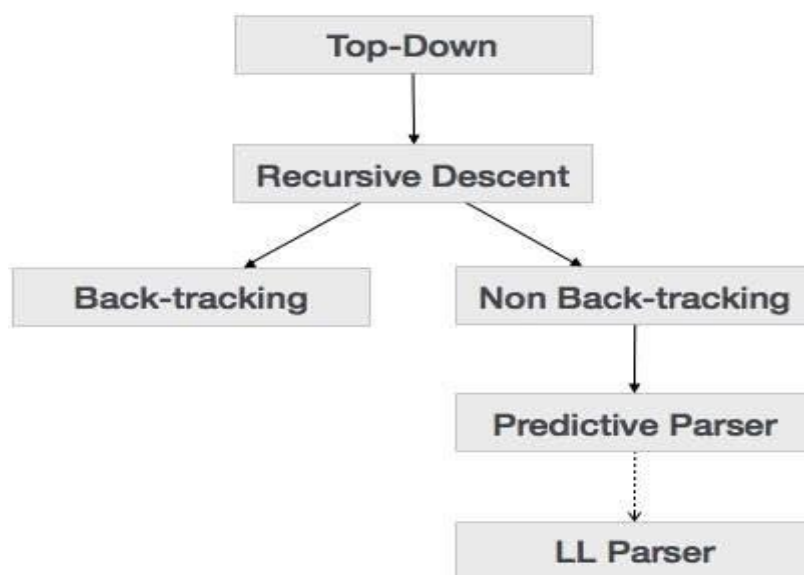
$\$ \triangleleft E \doteq + \doteq T \triangleright \$$

$\$ \triangleleft E \triangleright \$$

Accept.

Recursive Descent Parsing

The types of top-down parsing are depicted below:



Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

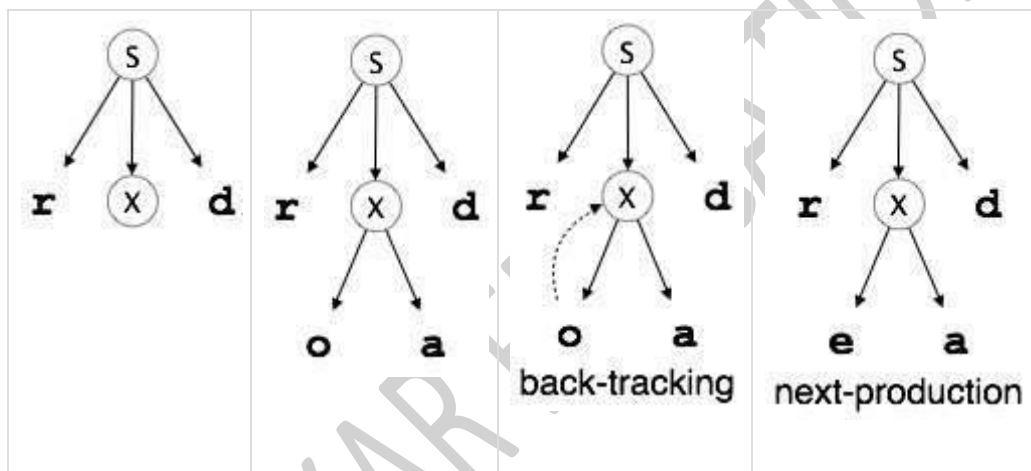
$$S \rightarrow rXd \mid rZd$$
$$X \rightarrow oa \mid ea$$

$Z \rightarrow ai$

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S ($S \rightarrow rXd$) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left ($X \rightarrow oa$). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, ($X \rightarrow ea$).

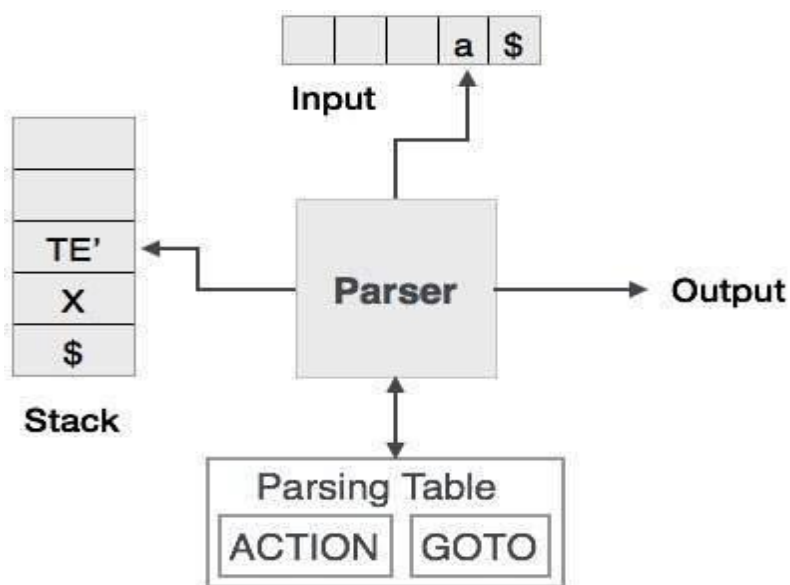
Now the parser matches all the input letters in an ordered manner. The string is accepted.



Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol \$ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

Top-Bottom Parser

*Remove Left Recursion
Left Factored Grammar*

Recursive Descent

Remove Back-tracking

Predictive Parser

*Use Table
Remove Recursion*

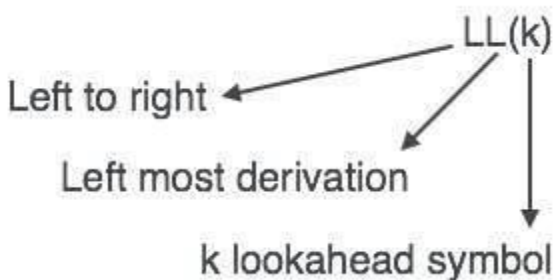
Non-recursive Predictive Parser

In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parser

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as $LL(k)$. The first L in $LL(k)$ is parsing the input from left to right, the second L in $LL(k)$ stands for left-most derivation and k itself represents the number of look aheads. Generally $k = 1$, so $LL(k)$ may also be written as $LL(1)$.



LL Parsing Algorithm

We may stick to deterministic $LL(1)$ for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not $LL(1)$, then usually, it is not $LL(k)$, for any given k.

Given below is an algorithm for $LL(1)$ Parsing:

Input:

string ω

parsing table M for grammar G

Output:

If ω is in $L(G)$ then left-most derivation of ω ,

error otherwise.

Initial State : $\$S$ on stack (with S being start symbol)

$\omega\$$ in the input buffer

SET ip to point the first symbol of $\omega\$$.

repeat

let X be the top stack symbol and a the symbol pointed by ip .

if $X \in V_t$ or $\$$

if $X = a$

POP X and advance ip .

else

error()

endif

else /* X is non-terminal */

if $M[X, a] = X \rightarrow Y_1, Y_2, \dots, Y_k$

POP X

PUSH $Y_k, Y_{k-1}, \dots, Y_1$ /* Y_1 on top */

Output the production $X \rightarrow Y_1, Y_2, \dots, Y_k$

else

error()

endif

endif

Theory of Computation and Compilers Unit – 8

until X = \$ /* empty stack */

A grammar G is LL(1) if $A \rightarrow \alpha \mid \beta$ are two distinct productions of G :

- for no terminal, both α and β derive strings beginning with a.
- at most one of α and β can derive empty string.
- if $\beta \rightarrow t$, then α does not derive any string beginning with a terminal in FOLLOW(A).

Shift reduce parsing

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.

A String $\xrightarrow{\text{reduce to}}$ the starting symbol

- Shift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

Example:

Grammar:

1. $S \rightarrow S+S$

Theory of Computation and Compilers Unit – 8

2. $S \rightarrow S-S$

3. $S \rightarrow (S)$

4. $S \rightarrow a$

Input string:

1. $a1-(a2+a3)$

Parsing table:

Stack contents	Input string	Actions
\$	$a1-(a2+a3)\$$	shift $a1$
$\$a1$	$-(a2+a3)\$$	reduce by $S \rightarrow a$
$\$S$	$-(a2+a3)\$$	shift $-$
$\$S-$	$(a2+a3)\$$	shift $($
$\$S-($	$a2+a3)\$$	shift $a2$
$\$S-(a2$	$+a3)\$$	reduce by $S \rightarrow a$
$\$S-(S$	$+a3)\$$	shift $+$
$\$S-(S+$	$a3)\$$	shift $a3$
$\$S-(S+a3$	$)\$$	reduce by $S \rightarrow a$
$\$S-(S+S$	$)\$$	shift $)$
$\$S-(S+S)$	$\$$	reduce by $S \rightarrow S+S$
$\$S-(S)$	$\$$	reduce by $S \rightarrow (S)$
$\$S-S$	$\$$	reduce by $S \rightarrow S-S$
$\$S$	$\$$	Accept

There are two main categories of shift reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

LR Parser

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input.

"R" stands for constructing a right most derivation in reverse.

"K" is the number of input symbols of the look ahead used to make number of parsing decision.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

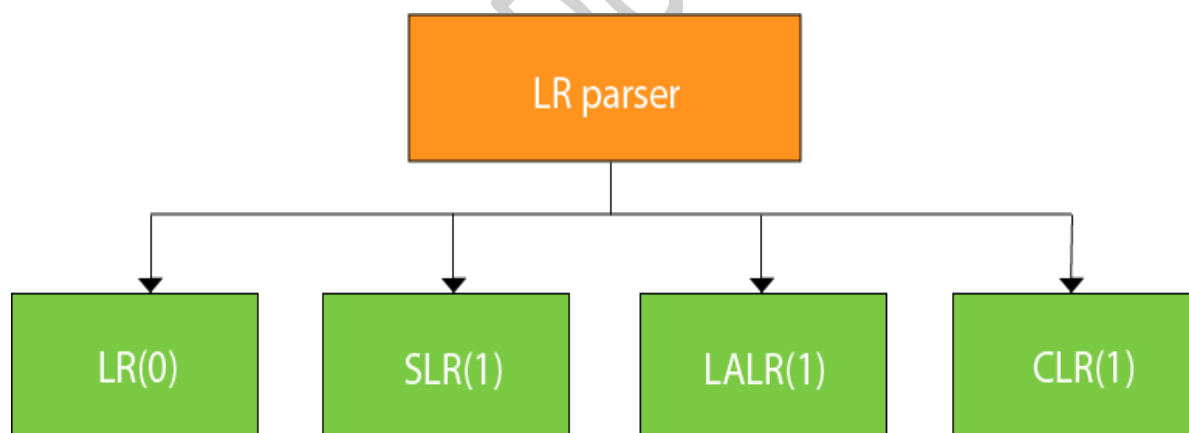


Fig: Types of LR parser

LR algorithm:

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

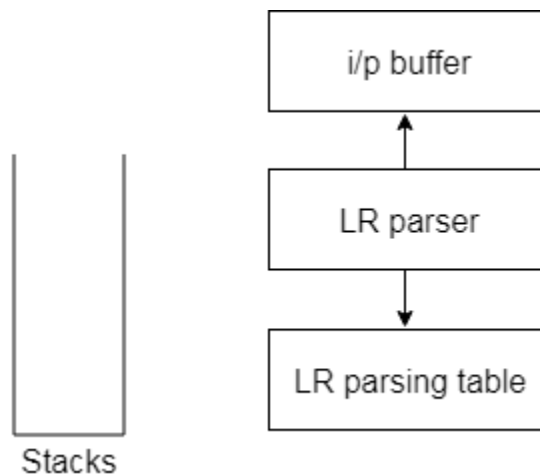


Fig: Block diagram of LR parser

Input buffer is used to indicate end of input and it contains the string to be parsed followed by a \$ Symbol.

A stack is used to contain a sequence of grammar symbols with a \$ at the bottom of the stack.

Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

LR (1) Parsing

Various steps involved in the LR (1) Parsing:

- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram (DFA).
- Construct a LR (1) parsing table.

Augment Grammar

Theory of Computation and Compilers Unit – 8

Augmented grammar G' will be generated if we add one more production in the given grammar G . It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example

Given grammar

1. $S \rightarrow AA$
2. $A \rightarrow aA \mid b$

The Augment grammar G' is represented by

1. $S' \rightarrow S$
2. $S \rightarrow AA$
3. $A \rightarrow aA \mid b$

LALR (1) Parsing:

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example

LALR (1) Grammar

1. $S \rightarrow AA$
2. $A \rightarrow aA$
3. $A \rightarrow b$

Theory of Computation and Compilers Unit – 8

Add Augment Production, insert ' $\bullet$ ' symbol at the first position for every production in G and also add the look ahead.

1. $S' \rightarrow \bullet S, \$$
2. $S \rightarrow \bullet AA, \$$
3. $A \rightarrow \bullet aA, a/b$
4. $A \rightarrow \bullet b, a/b$

I0 State:

Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure ($S' \rightarrow \bullet S$)

Add all productions starting with S in to I0 State because " $\bullet$ " is followed by the non-terminal. So, the I0 State becomes

I0 = $S' \rightarrow \bullet S, \$$
 $S \rightarrow \bullet AA, \$$

Add all productions starting with A in modified I0 State because " $\bullet$ " is followed by the non-terminal. So, the I0 State becomes.

I0 = $S' \rightarrow \bullet S, \$$
 $S \rightarrow \bullet AA, \$$
 $A \rightarrow \bullet aA, a/b$
 $A \rightarrow \bullet b, a/b$

I1 = Go to (I0, S) = closure ($S' \rightarrow S\bullet, \$$) = $S' \rightarrow S\bullet, \$$

I2 = Go to (I0, A) = closure ($S \rightarrow A\bullet A, \$$)

Add all productions starting with A in I2 State because " $\bullet$ " is followed by the non-terminal. So, the I2 State becomes

I2= $S \rightarrow A \bullet A, \$$

$A \rightarrow \bullet aA, \$$

$A \rightarrow \bullet b, \$$

I3= Go to (I0, a) = Closure ($A \rightarrow a \bullet A, a/b$)

Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes

I3= $A \rightarrow a \bullet A, a/b$

$A \rightarrow \bullet aA, a/b$

$A \rightarrow \bullet b, a/b$

Go to (I3, a) = Closure ($A \rightarrow a \bullet A, a/b$) = (same as I3)

Go to (I3, b) = Closure ($A \rightarrow b \bullet, a/b$) = (same as I4)

I4= Go to (I0, b) = closure ($A \rightarrow b \bullet, a/b$) = $A \rightarrow b \bullet, a/b$

I5= Go to (I2, A) = Closure ($S \rightarrow AA \bullet, \$$) = $S \rightarrow AA \bullet, \$$

I6= Go to (I2, a) = Closure ($A \rightarrow a \bullet A, \$$)

Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes

I6 = $A \rightarrow a \bullet A, \$$

$A \rightarrow \bullet aA, \$$

$A \rightarrow \bullet b, \$$

Go to (I6, a) = Closure ($A \rightarrow a \bullet A, \$$) = (same as I6)

Go to (I6, b) = Closure ($A \rightarrow b \bullet, \$$) = (same as I7)

I7= Go to (I2, b) = Closure ($A \rightarrow b \bullet, \$$) = $A \rightarrow b \bullet, \$$

I8= Go to (I3, A) = Closure ($A \rightarrow aA \bullet, a/b$) = $A \rightarrow aA \bullet, a/b$

I9= Go to (I6, A) = Closure ($A \rightarrow aA \bullet, \$$) = $A \rightarrow aA \bullet, \$$

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { $A \rightarrow a \bullet A, a/b$
 $A \rightarrow \bullet aA, a/b$
 $A \rightarrow \bullet b, a/b$
}

I6 = { $A \rightarrow a \bullet A, \$$
 $A \rightarrow \bullet aA, \$$
 $A \rightarrow \bullet b, \$$
}

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { $A \rightarrow a \bullet A, a/b/\$$
 $A \rightarrow \bullet aA, a/b/\$$
 $A \rightarrow \bullet b, a/b/\$$
}

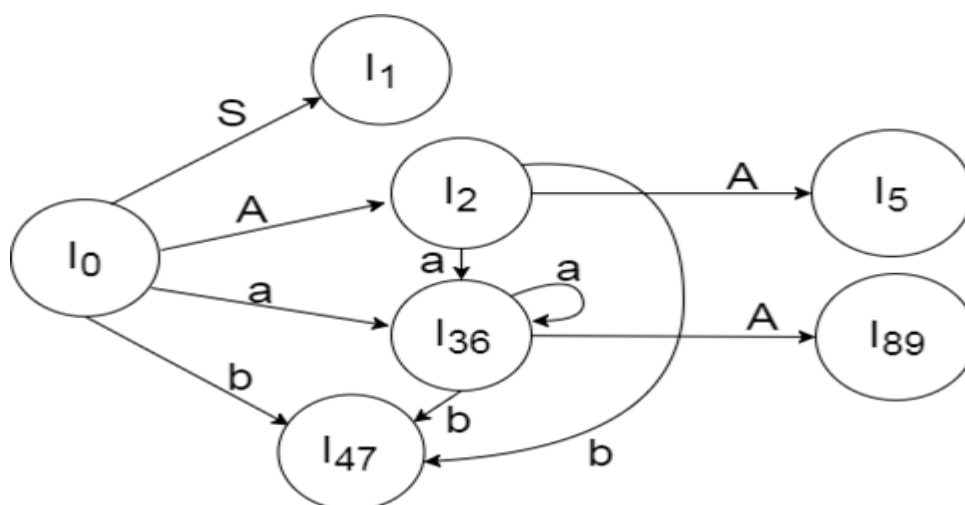
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = { $A \rightarrow b \bullet, a/b/\$$ }

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = { $A \rightarrow aA \bullet, a/b/\$$ }

Drawing DFA:



LALR (1) Parsing table:

States	a	b	S	S	A
I ₀	S ₃₆	S ₄₇		12	
I ₁		accept			
I ₂	S ₃₆	S ₄₇			5
I ₃₆	S ₃₆ S ₄₇				89
I ₄₇	R ₃ R ₃	R ₃			
I ₅			R ₁		
I ₈₉	R ₂	<u>R₂</u>	<u>R₂</u>		

Semantic Analysis

The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

$$E \rightarrow E + T$$

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Theory of Computation and Compilers Unit – 8

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

`int a = "value";`

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

$$E \rightarrow E + T \{ E.value = E.value + T.value \}$$

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$$S \rightarrow ABC$$

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example ($E \rightarrow E + T$), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

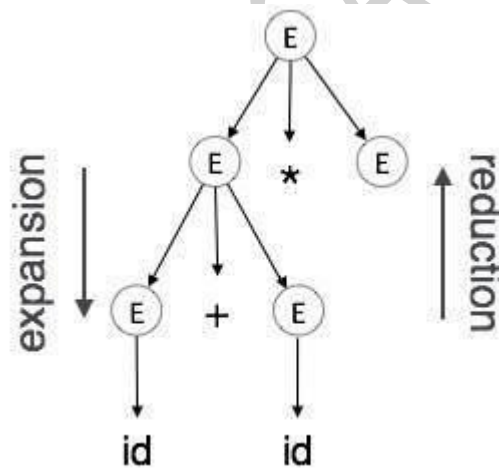
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

$S \rightarrow ABC$

A can get values from S , B and C . B can take values from S , A , and C . Likewise, C can take values from S , A , and B .

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Theory of Computation and Compilers Unit – 8

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 5;

<type, "integer">

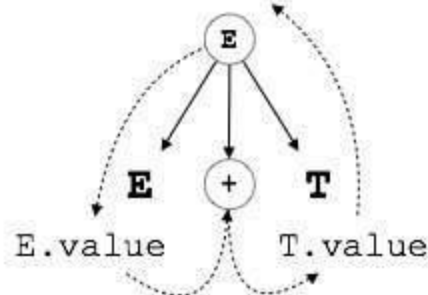
<presentvalue, "5">

For every production, we attach a semantic rule.

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

$E.value = E.value + T.value$



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

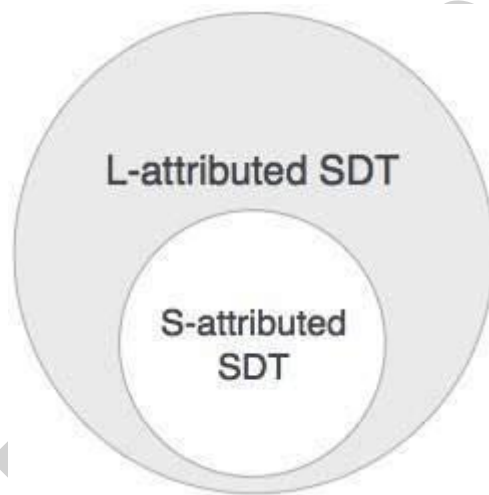
Theory of Computation and Compilers Unit – 8

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

$S \rightarrow ABC$

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Syntax directed translation

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So we can say that

1. Grammar + semantic rule = SDT (syntax directed translation)
 - In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the

Theory of Computation and Compilers Unit – 8

attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.

- In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	Semantic Rules
$E \rightarrow E + T$	$E.val := E.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T * F$	$T.val := T.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (F)$	$F.val := F.val$
$F \rightarrow num$	$F.val := num.lexval$

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
$S \rightarrow E \$$	{ printE.VAL }
$E \rightarrow E + E$	{E.VAL := E.VAL + E.VAL }
$E \rightarrow E * E$	{E.VAL := E.VAL * E.VAL }
$E \rightarrow (E)$	{E.VAL := E.VAL }
$E \rightarrow I$	{E.VAL := I.VAL }
$I \rightarrow I \text{ digit}$	{I.VAL := 10 * I.VAL + LEXVAL }
$I \rightarrow \text{digit}$	{ I.VAL:= LEXVAL }

Static checking

The term static checking refers to error checks made at compile time. The opposite of static is dynamic, which refers to the time a program is executing. The error handling described in Module 5 finds static errors which do not correspond to the BNF description of that language. BNF is a metalanguage. A metalanguage is a way of describing another language. Thus, BNF describes a class of languages called context-free. Context-free languages can describe programming language syntactic structures such as statements and loop structures. Thus, if an END is left out, the error routines described in Module 5 will discover this.

However, context-free languages cannot describe the fact that a variable has been used but not defined or that a referenced label is not there. This requires contextual constraints which context-free grammars cannot specify.

There is a widely-accepted formalism, attribute grammars, for describing the range of semantic actions needed to do static checking of programming languages.

Run-Time Environment

A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

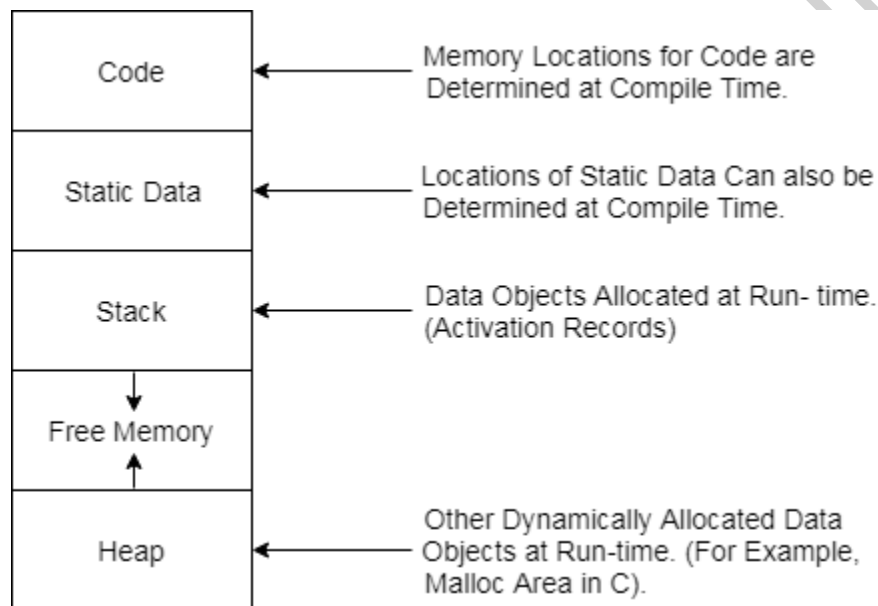
Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

Storage Organization

Theory of Computation and Compilers Unit – 8

- When the target program executes then it runs in its own logical address space in which the value of each program has a location.
- The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

Subdivision of Run-time Memory:



- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.
- Run-time storage can be subdivide to hold the different components of an executing program:
 1. Generated executable code
 2. Static data objects
 3. Dynamic data-object- heap
 4. Automatic data objects- stack

Activation Trees

Theory of Computation and Compilers Unit – 8

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

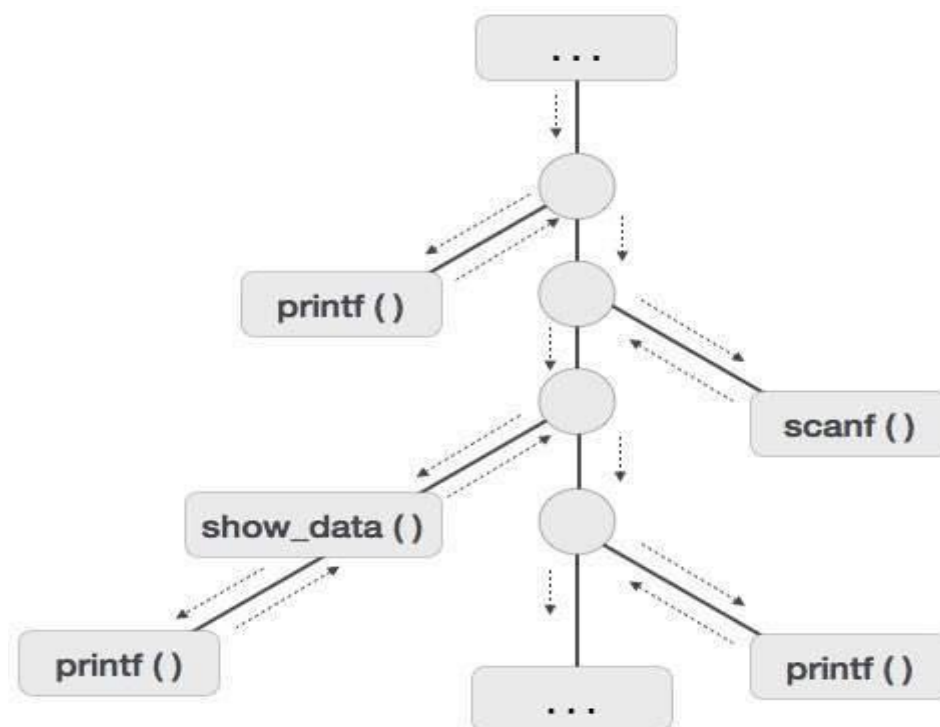
Theory of Computation and Compilers Unit – 8

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

To understand this concept, we take a piece of code as an example:

```
...  
printf("Enter Your Name: ");  
scanf("%s", username);  
show_data(username);  
printf("Press any key to continue...");  
...  
int show_data(char *user)  
{  
    printf("Your name is %s", username);  
    return 0;  
}  
...
```

Below is the activation tree of the code given.



Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

Activation Record

- Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.
- When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
- Activation record is used to manage the information needed by a single execution of a procedure.
- An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

The diagram below shows the contents of activation records:

Return value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

Return Value: It is used by calling procedure to return a value to calling procedure.

Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- **Code :** It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures :** Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables :** Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

1. Implementation of call statement:

The following code is needed to implement static allocation:

1. `MOV #here + 20, callee.static_area` /*it saves return address*/
2. `GOTO callee.code_area` /* It transfers control to the target code for the called procedure*/

Where,

callee.static_area shows the address of the activation record.

callee.code_area shows the address of the first instruction for called procedure.

#here + 20 literal are used to return address of the instruction following GOTO.

2. Implementation of return statement:

The following code is needed to implement return from procedure callee:

1. `GOTO *callee.static_area`

It is used to transfer the control to the address that is saved at the beginning of the activation record.

3. Implementation of action statement:

The ACTION instruction is used to implement action statement.

4. Implementation of halt statement:

The HALT statement is the final instruction that is used to return the control to the operating system.

Stack Allocation

Theory of Computation and Compilers Unit – 8

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Using the relative address, static allocation can become stack allocation for storage in activation records.

In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.

The following code is needed to implement stack allocation:

1. Initialization of stack:

1. `MOV #stackstart, SP` /*initializes stack*/
2. `HALT` /*terminate execution*/

2. Implementation of Call statement:

1. `ADD #caller.recordsize, SP` /* increment stack pointer */
2. `MOV #here + 16, *SP` /*Save return address */
3. `GOTO callee.code_area`

Where,

caller.recordsize is the size of the activation record

#here + 16 is the address of the instruction following the **GOTO**

3. Implementation of Return statement:

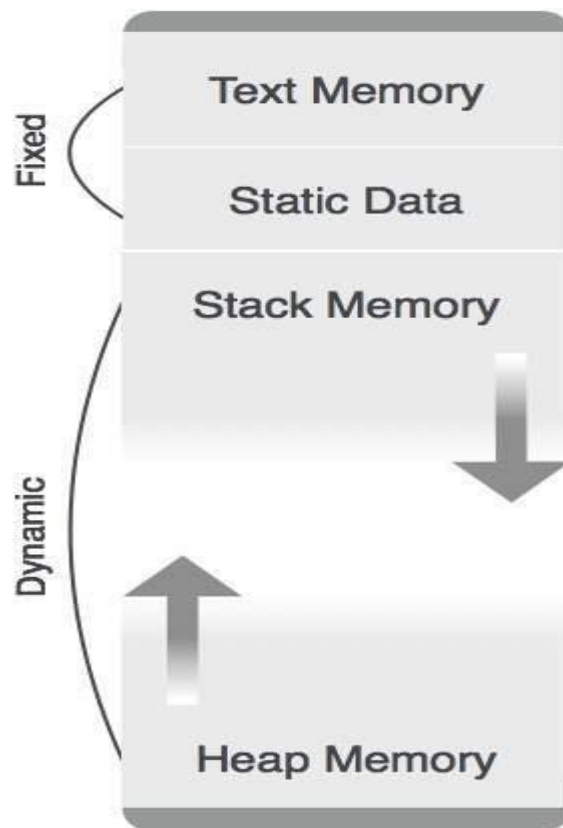
1. `GOTO *0 ( SP )` /*return to the caller */
2. `SUB #caller.recordsize, SP` /*decrement SP and restore to previous value */

Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Theory of Computation and Compilers Unit – 8

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



As shown in the image above, the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

Theory of Computation and Compilers Unit – 8

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

l-value

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;  
week = day * 7;  
month = 1;  
year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

Formal Parameters

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

Actual Parameters

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
    int actual_parameter = 10;
    call fun_two(int actual_parameter);
}
fun_two(int formal_parameter)
{
    print formal_parameter;
}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass by Value

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

Pass by Reference

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

Pass by Copy-restore

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when

the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;  
calling_procedure()  
{  
    y = 10;  
    copy_restore(y); //l-value of y is passed  
    printf y; //prints 99  
}  
copy_restore(int x)  
{  
    x = 99; // y still has value 10 (unaffected)  
    y = 0; // y is now 0  
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

Symbol Table

Symbol table is an important data structure used in a compiler.

Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

Theory of Computation and Compilers Unit – 8

The symbol table used for following purposes:

- It is used to store the name of all entities in a structured form at one place.
- It is used to verify if a variable has been declared.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

1. <symbol name, type, attribute>

For example, suppose a variable store the information about the following variable declaration:

1. **static int** salary

then, it stores an entry in the following format:

1. <salary, **int**, **static**>

The clause attribute contains the entries related to the name.

Implementation

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

A symbol table can be implemented in one of the following techniques:

- Linear (sorted or unsorted) list
- Hash table
- Binary search tree

Theory of Computation and Compilers Unit – 8

Symbol table are mostly implemented as hash table.

Operations

The symbol table provides the following operations:

Insert ()

- Insert () operation is more frequently used in the analysis phase when the tokens are identified and names are stored in the table.
- The insert() operation is used to insert the information in the symbol table like the unique name occurring in the source code.
- In the source code, the attribute for a symbol is the information associated with that symbol. The information contains the state, value, type and scope about the symbol.
- The insert () function takes the symbol and its value in the form of argument.

For example:

1. **int x;**

Should be processed by the compiler as:

1. insert (x, **int**)

lookup()

In the symbol table, lookup() operation is used to search a name. It is used to determine:

- The existence of symbol in the table.
- The declaration of the symbol before it is used.
- Check whether the name is used in the scope.

Theory of Computation and Compilers Unit – 8

- Initialization of the symbol.
- Checking whether the name is declared multiple times.

The basic format of lookup() function is as follows:

1. lookup (symbol)

This format varies according to the programming language.

Data structure for symbol table

- A compiler contains two type of symbol table: global symbol table and scope symbol table.
- Global symbol table can be accessed by all the procedures and scope symbol table.

The scope of a name and symbol table is arranged in the hierarchy structure as shown below:

```
int value=10;

void sum_num()
{
    int num_1;
    int num_2;
    {
        int num_3;
        int num_4;
    }
    int num_5;
```

```
{  
    int_num 6;  
    int_num 7;  
}  
  
Void sum_id  
{  
    int id_1;  
    int id_2;  
    {  
        int id_3;  
        int id_4;  
    }  
    int num_5;  
}
```

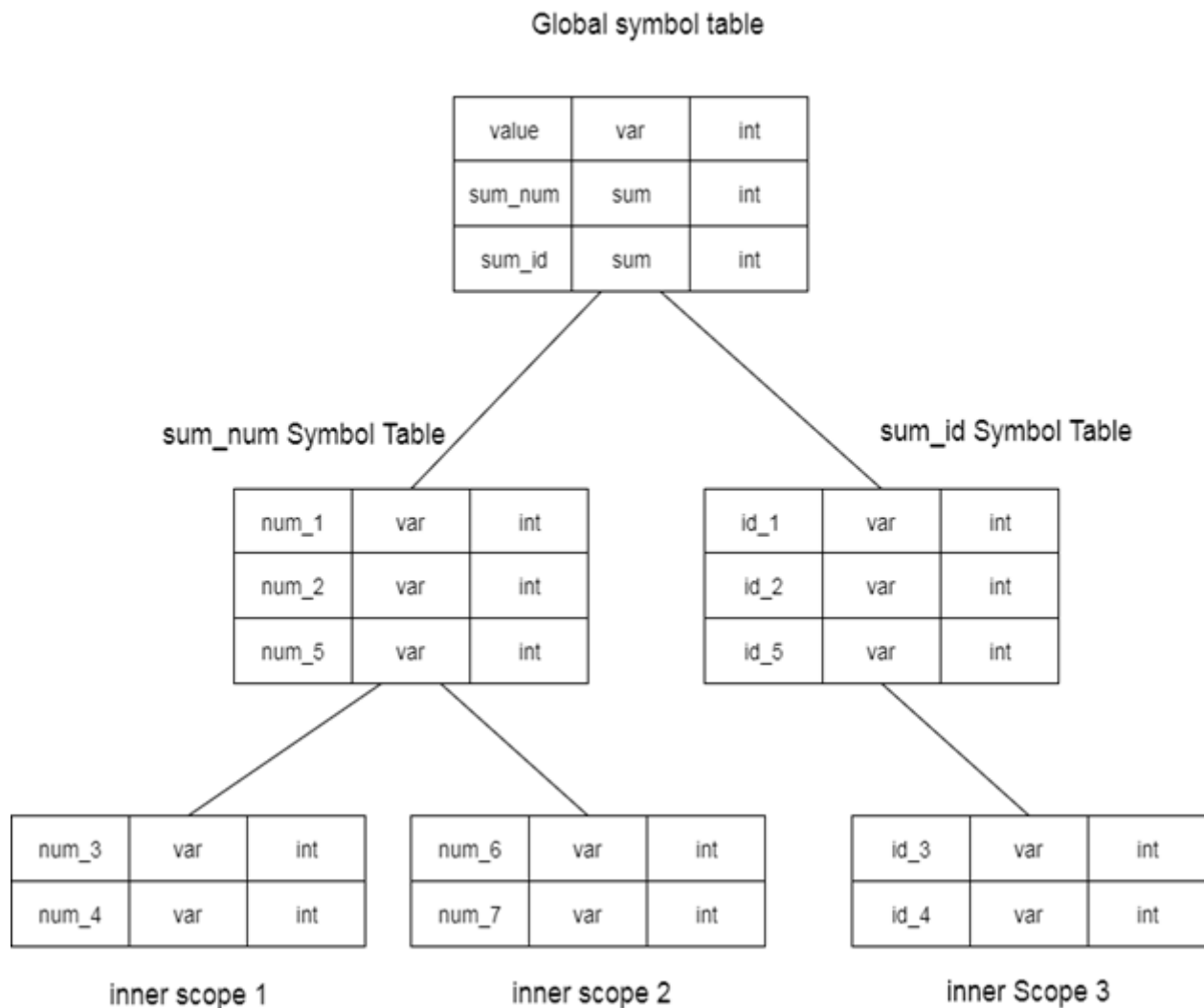
The above grammar can be represented in a hierarchical data structure of symbol tables:

Data structure hierarchy of symbol table is stored in the semantic analyzer. If you want to search the name in the symbol table then you can search it using the following algorithm:

- First a symbol is searched in the current symbol table.

Theory of Computation and Compilers Unit – 8

- If the name is found then search is completed else the name will be searched in the symbol table of parent until,
- The name is found or global symbol is searched.

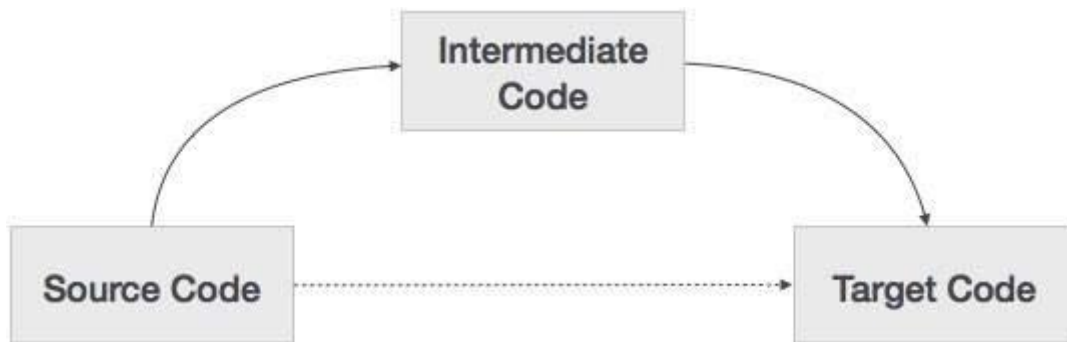


The global symbol table contains one global variable and two procedure names. The name mentioned in the sum_num table is not available for sum_id and its child tables.

Intermediate Code Generation

Theory of Computation and Compilers Unit – 8

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Theory of Computation and Compilers Unit – 8

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;  
r2 = b + r1;  
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg ₁	arg ₂	result
*	c	d	r1
+	b	r1	r2

Theory of Computation and Compilers Unit – 8

+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2. The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg ₁	arg ₂
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol

Theory of Computation and Compilers Unit – 8

table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;  
float b;
```

Allocation process:

```
{offset = 0}
```

```
int a;  
id.type = int  
id.width = 2
```

```
offset = offset + id.width  
{offset = 2}
```

```
float b;  
id.type = float  
id.width = 4
```

```
offset = offset + id.width  
{offset = 6}
```

Theory of Computation and Compilers Unit – 8

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

```
enter(name, type, offset)
```

This procedure should create an entry in the symbol table, for variable *name*, having its type set to *type* and relative address *offset* in its data area.

Translation of Assignment Statements

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Consider the grammar

1. $S \rightarrow id := E$
2. $E \rightarrow E1 + E2$
3. $E \rightarrow E1 * E2$
4. $E \rightarrow (E1)$
5. $E \rightarrow id$

The translation scheme of above grammar is given below:

Production rule	Semantic actions
$S \rightarrow id := E$	<pre>{p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }</pre>

Theory of Computation and Compilers Unit – 8

$E \rightarrow E1 + E2$	<pre>{E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }</pre>
$E \rightarrow E1 * E2$	<pre>{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }</pre>
$E \rightarrow (E1)$	<pre>{E.place = E1.place}</pre>
$E \rightarrow id$	<pre>{p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }</pre>

- The p returns the entry for id.name in the symbol table.
- The Emit function is used for appending the three address code to the output file. Otherwise it will report an error.
- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.

Boolean expressions

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

Theory of Computation and Compilers Unit – 8

1. $E \rightarrow E \text{ OR } E$
2. $E \rightarrow E \text{ AND } E$
3. $E \rightarrow \text{NOT } E$
4. $E \rightarrow (E)$
5. $E \rightarrow \text{id relop id}$
6. $E \rightarrow \text{TRUE}$
7. $E \rightarrow \text{FALSE}$

The relop is denoted by $<$, $>$, $<=$, $>=$.

```
{E.place = newtemp();  
  Emit ('if' id1.place relop.op  
  id2.place 'goto'  
  nextstar + 3);  
  EMIT (E.place ':=' '0')  
  EMIT ('goto' nextstat + 2)  
  EMIT (E.place ':=' '1')  
}
```

$E \rightarrow \text{TRUE}$

```
{E.place := newtemp();  
  Emit (E.place ':=' '1')  
}
```

$E \rightarrow \text{FALSE}$

```
{E.place := newtemp();  
  Emit (E.place ':=' '0')  
}
```

Theory of Computation and Compilers Unit – 8

The EMIT function is used to generate the three address code and the newtemp() function is used to generate the temporary variables.

The $E \rightarrow id \text{ relop } id2$ contains the next_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three address code using the above translation scheme:

$p > q$ AND $r < s$ OR $u > r$

100: **if** $p > q$ **goto** 103

101: $t1 := 0$

102: **goto** 104

103: $t1 := 1$

104: **if** $r > s$ **goto** 107

105: $t2 := 0$

106: **goto** 108

107: $t2 := 1$

108: **if** $u > v$ **goto** 111

109: $t3 := 0$

110: **goto** 112

111: $t3 := 1$

112: $t4 := t1 \text{ AND } t2$

113: $t5 := t4 \text{ OR } t3$

Procedures call

Theory of Computation and Compilers Unit – 8

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

- When a procedure call occurs then space is allocated for activation record.
- Evaluate the argument of the called procedure.
- Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
- Save the state of the calling procedure so that it can resume execution after the call.
- Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
- Finally generate a jump to the beginning of the code for the called procedure.

a grammar for a simple procedure call statement

1. $S \rightarrow \text{call id}(\text{Elist})$
2. $\text{Elist} \rightarrow \text{Elist}, E$
3. $\text{Elist} \rightarrow E$

A suitable transition scheme for procedure call would be:

Production Rule	Semantic Action
-----------------	-----------------

$S \rightarrow \text{call id}(\text{Elist})$	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)
$\text{Elist} \rightarrow \text{Elist}, E$	append E.PLACE to the end of QUEUE
$\text{Elist} \rightarrow E$	initialize QUEUE to contain only E.PLACE

Queue is used to store the list of parameters in the procedure call.

Code Generation

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

Theory of Computation and Compilers Unit – 8

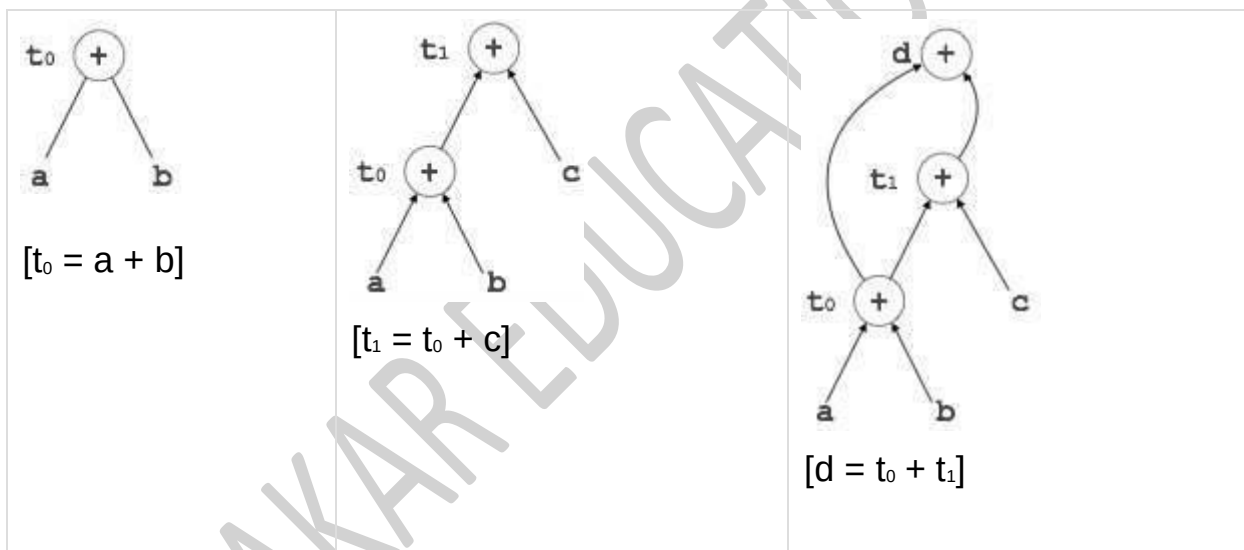
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$



Code Optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

Theory of Computation and Compilers Unit – 8

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

Theory of Computation and Compilers Unit – 8

<pre>int add_ten(int x) { int y, z; y = 10; z = x + y; return z; }</pre>	<pre>int add_ten(int x) { int y; y = 10; y = x + y; return y; }</pre>	<pre>int add_ten(int x) { int y = 10; return x + y; }</pre>	<pre>int add_ten(int x) { return x + 10; }</pre>

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

MOV x, R1

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidentally written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
    return x + 10;

    printf("value of x is %d", x);
}
```

}

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

...

MOV R1, R2

GOTO L1

...

L1 : GOTO L2

L2 : INC R1

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

...

MOV R1, R2

GOTO L2

...

L2 : INC R1

Algebraic expression simplification

Theory of Computation and Compilers Unit – 8

There are occasions where algebraic expressions can be made simple. For example, the expression $a = a + 0$ can be replaced by a itself and the expression $a = a + 1$ can simply be replaced by $\text{INC } a$.

Strength reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, $x * 2$ can be replaced by $x \ll 1$, which involves only one left shift. Though the output of $a * a$ and a^2 is same, a^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

Code Generator

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
- **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.

Theory of Computation and Compilers Unit – 8

- **Selection of instruction** : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation** : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Descriptors

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor** : Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.
- **Address descriptor** : Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

- updates the Register Descriptor R1 that has value of x and
- updates the Address Descriptor (x) to show that one instance of x is in R1.

Code Generation

Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input.

Note : If the value of a name is found at more than one place (register, cache, or memory), the register's value will be preferred over the cache and main memory. Likewise cache's value will be preferred over the main memory. Main memory is barely given any preference.

getReg : Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction $x = y \text{ OP } z$, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the output of $y \text{ OP } z$ is to be saved:

- Call function *getReg*, to decide the location of L.
- Determine the present location (register or memory) of **y** by consulting the Address Descriptor of **y**. If **y** is not presently in register L, then generate the following instruction to copy the value of **y** to L:

MOV **y'**, L

where **y'** represents the copied value of **y**.

Theory of Computation and Compilers Unit – 8

- Determine the present location of **z** using the same method used in step 2 for **y** and generate the following instruction:

OP **z'**, **L**

where **z'** represents the copied value of **z**.

- Now **L** contains the value of **y** OP **z**, that is intended to be assigned to **x**. So, if **L** is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.
- If **y** and **z** has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly way.

Global data flow analysis

- To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
- Certain optimization can only be achieved by examining the entire program. It can't be achieved by examining just a portion of the program.
- For this kind of optimization user defined chaining is one particular problem.
- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

1. $x = a + b;$
2. $x = 6 * 3$

Theory of Computation and Compilers Unit – 8

- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression $6*3$ will be computed, simplifying the second assignment statement to $x = 18$;

Some optimization needs more global information. For example, consider the following code:

1. $a = 1$;
2. $b = 2$;
3. $c = 3$;
4. **if** (...) $x = a + 5$;
5. **else** $x = b + 4$;
6. $c = x + 1$;

In this code, at line 3 the initial assignment is useless and $x + 1$ expression can be simplified as 7.

But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

- Which variables are guaranteed to have constant values
- Which variables will be used before being redefined

Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

Loop Optimization

Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.

If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.

For loop optimization the following three techniques are important:

1. Code motion
2. Induction-variable elimination
3. Strength reduction

1.Code Motion:

Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

For example

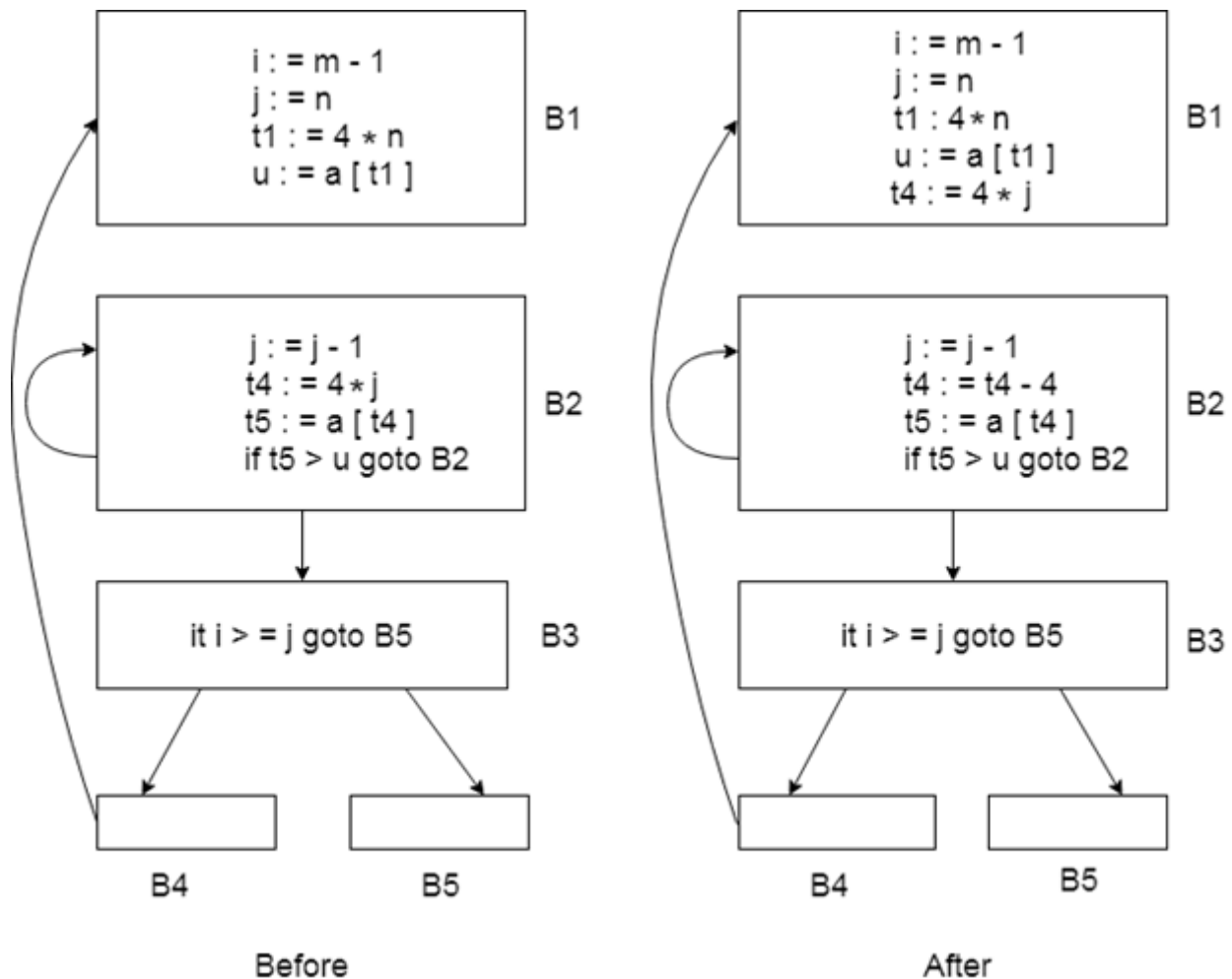
In the while statement, the limit-2 equation is a loop invariant equation.

1. **while** (i<=limit-2) /*statement does not change limit*/
2. After code motion the result is as follows:
3. a= limit-2;
4. **while**(i<=a) /*statement does not change limit or a*/

2.Induction-Variable Elimination

Induction variable elimination is used to replace variable from inner loop.

It can reduce the number of additions in a loop. It improves both code space and run time performance.



In this figure, we can replace the assignment $t4 := 4 * j$ by $t4 := t4 - 4$. The only problem which will be arose that $t4$ does not have a value when we enter block B2 for the first time. So we place a relation $t4 = 4 * j$ on entry to the block B2.

3.Reduction in Strength

- Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
- Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

Example:

```
while (i<10)
{
    j= 3 * i+1;
    a[j]=a[j]-2;
    i=i+2;
}
```

After strength reduction the code will be:

```
s= 3*i+1;
while (i<10)
{
    j=s;
    a[j]= a[j]-2;
    i=i+2;
    s=s+6;
}
```

In the above code, it is cheaper to compute $s=s+6$ than $j=3 * i$

Phases of Optimization

There are generally two phases of optimization:

- **Global Optimization:**
- **Local Optimization:**

Local Optimization

Theory of Computation and Compilers Unit – 8

Transformations are applied to small blocks of statements. The local optimization is done prior to global optimization.

The local optimization builds on a superstructure formulation, derived from the information of the genome that is to be evaluated in the EA. The superstructure is based on a rigorous MESH model with up to 190 equilibrium trays in total and a number of binary decisions w.r.t. the allocation for the feed, boil-up and recycle streams. The solution of this highly nonlinear MINLP is achieved via a polyhedral modeling approach that integrates implicit functions for VLE and enthalpy computations (Skiborowski et al., 2015b) and the solution of a series of successive relaxed NLP problems with additional NCP-functions. The necessary extensions for the optimization of the energy-integrated process variants are further described in the articles of Waltermann et al. (2016, 2018). Each of the variants builds on the initialization of the basic process variant, while specific modifications of the superstructure model are automatically introduced in a sequence of refinement steps and embedded optimization problems.

- The simplest form of optimization
- Optimize one basic block. (No need to analyze the whole procedure body).

Some statements can be deleted

$x := x + 0$

$x := x * 1$

Eliminating these instructions are great optimizations as we are removing an entire instruction.

Some statements can be simplified

$x := x * 0 \Rightarrow x := 0$

Theory of Computation and Compilers Unit – 8

Maybe the instruction on the right is faster than the instruction on the right. More importantly, assigning to a constant allows more cascading optimizations, as we will see later.

Another example:

$y := y ** 2 \Rightarrow y := y * y$

Typically, an architecture will not have the exponentiation instruction and so this operator may have to be implemented by an exponentiation loop in software. For the special cases (e.g., exponent = 2), we can simply replace by multiplication.

Another example:

$x := x * 8 \Rightarrow x := x \ll 3$

$x := x * 15 \Rightarrow t := x \ll 4; x := t - x$

Replace multiplication by a power-of-two to a shift-left operation. Can also do this for non powers-of-two. On some machines, left-shift is faster than multiply, but not on all! Modern machines have their own "rewriting logic" in hardware that can deal with these special cases really fast. In other words, some of these compiler local optimizations may become less relevant on modern hardware.

All these transformations are examples of *algebraic simplifications*.

Operations on constants can be computed at compile time

- If there is a statement $x := y \text{ op } z$
- And y and z are constants.
- Then $y \text{ op } z$ can be computed at compile time.

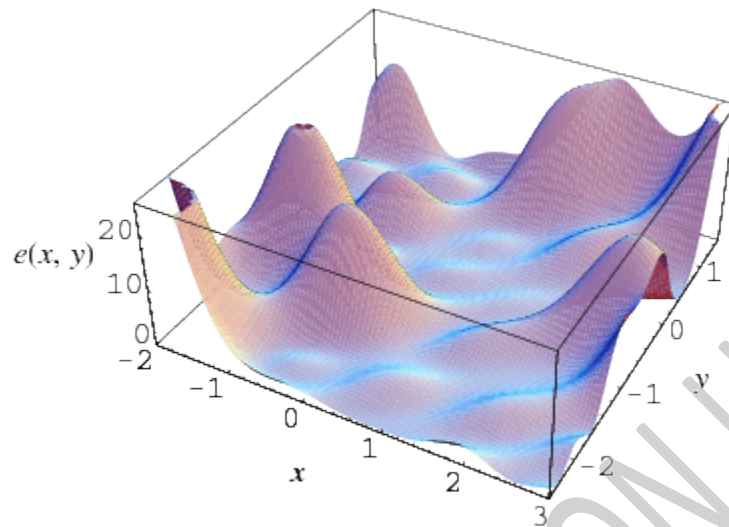
Global Optimization:

Transformations are applied to large program segments that includes functions, procedures and loops.

The objective of global optimization is to find the globally best solution of (possibly nonlinear) models, in the (possible or known) presence of multiple local optima. Formally, global optimization seeks global solution(s) of a constrained optimization model. Nonlinear models are ubiquitous in many applications, e.g., in advanced engineering design, biotechnology, data analysis, environmental management, financial planning, process control, risk management, scientific modeling, and others. Their solution often requires a global search approach.

A few application examples include acoustics equipment design, cancer therapy planning, chemical process modeling, data analysis, classification and visualization, economic and financial forecasting, environmental risk assessment and management, industrial product design, laser equipment design, model fitting to data (calibration), optimization in numerical mathematics, optimal operation of "closed" (confidential) engineering or other systems, packing and other object arrangement problems, portfolio management, potential energy models in computational physics and chemistry, process control, robot design and manipulations, systems of nonlinear equations and inequalities, and waste water treatment systems management.

To formulate the problem of global optimization, assume that the objective function f and the constraints g are continuous functions, the component-wise bounds x_l and x_u related to the decision variable vector x are finite, and the feasible set D is nonempty. These assumptions guarantee that the global optimization model is well-posed since, by the extreme value theorem, the solution set of the global optimization model is nonempty.



As an example, consider the l_2 -norm error function related to solving the pair of equations

$$x - \sin(2x + 3y) - \cos(3x - 5y) = 0 \quad (1)$$

$$y - \sin(x - 2y) + \cos(x + 3y) = 0. \quad (2)$$

Since we wish to find the globally smallest error and not just a "locally smallest" one, we need to search globally in the two-dimensional box region (see the base area of the picture). There are great many nonlinear optimization problems that possess a similar multi-modal structure.

If we use traditional local scope search methods to solve this problem, then depending on the starting point of the search, we will often find locally optimal solutions of varying quality (cf. the "valleys" in the figure above that could easily trap local search methods). In order to find the globally optimal solution, a global scope search effort is needed.

Several of the most important global optimization strategies are listed below, together with short additional remarks and references. Most global optimization software implementations are based upon one of these approaches, possibly combining ideas from several strategies.

Exact methods include:

1. naive approaches: These include the most well known passive (simultaneous) or direct (not fully adaptive) sequential global optimization strategies such as uniform grid, space covering, and pure random searches. Note that these methods are convergent under mild assumptions but, as a rule, are not practical in higher-dimensional problems.
2. complete (enumerative) search strategies: These are based upon an exhaustive (and typically streamlined) enumeration of all possible solutions. These are applicable to combinatorial problems, as well as to certain "well-structured" continuous global optimization problems such as concave programming.
3. homotopy (parameter continuation), trajectory methods, and related approaches: These methods have the "ambitious" objective of visiting all stationary points of the objective function: this, in turn, leads to the list of all (global as well as local) optima. This general approach includes differential equation model based, path-following search strategies, as well as fixed-point methods and pivoting algorithms.
4. successive approximation (relaxation) methods: In this approach, the initial optimization problem is replaced by a sequence of relaxed sub-problems that are easier to solve. Successive refinement of the sub-problems to approximate the original problem is done e.g., by cutting planes and by more general cuts, diverse minorant function constructions, nested optimization and decomposition strategies and so on. These approaches are applicable to diverse structured global optimization problems such as concave minimization and differential convex problems.
5. branch and bound algorithms: A variety of adaptive partition strategies have been proposed to solve global optimization models. These are based upon partition, sampling, and subsequent lower and upper bounding procedures: these operations are applied iteratively to the collection of active ("candidate") subsets within the feasible set D . Their exhaustive search feature is guaranteed in similar

spirit to the analogous integer linear programming methodology. Branch and bound subsumes many specific approaches, and allows for a variety of implementations. Branch and bound methods typically rely on some a priori structural knowledge about the problem. This information may relate, for instance to how rapidly each function can vary (e.g., the knowledge of a suitable "overall" Lipschitz constant, for each function f and g); or to the availability of an analytic formulation and guaranteed smoothness of all functions (for instance, in interval arithmetic-based methods). The general branch and bound methodology is applicable to broad classes of global optimization problems, e.g., in combinatorial optimization, concave minimization, reverse convex programs, DC programming, and Lipschitz optimization.

6. Bayesian search (partition) algorithms: These methods are based upon postulated statistical information, to enable a prior stochastic description of the function-class modeled. During optimization, the problem-instance characteristics are adaptively estimated and updated. Note that, as a rule, only the corresponding one-dimensional model development is exact; furthermore, that in most practical cases "myopic" approximate decisions govern the search procedure. This general approach is applicable also to (merely) continuous global optimization problems. Theoretically, convergence to the optimal solution set is guaranteed only by generating an everywhere dense set of search points. One of the obvious challenges of using statistical methods is the choice and verification of an "appropriate" statistical model, for the class of problems to which they are applied. Additionally, it seems to be difficult to implement rigorous, computationally efficient versions of these algorithms for higher dimensional optimization problems. Note, however, that if one "skips" the underlying Bayesian paradigm, then these methods can also be pragmatically viewed as adaptive partition algorithms, and, as such, they can be directly extended to higher dimensions.

7. adaptive stochastic search algorithms: This is another broad class of methods, based upon "exhaustive" random sampling in the feasible set. In its basic form, it includes various random search strategies that are convergent, with probability

one. Search strategy adjustments, clustering and deterministic solution refinement options, statistical stopping rules, etc. can also be added as enhancements. The methodology is applicable to both discrete and continuous global optimization problems under very mild conditions.

Heuristic strategies include:

1. "Globalized" extensions of local search methods: These are partially heuristic algorithms that are nonetheless often successful in practice. The essential idea is to apply a preliminary grid search or random search based global phase, followed by applying a local (convex programming) method. For instance, random multistart performs a local search from several points selected randomly from the search domain D . Note that even such sampling is not trivial, when D has a complicated shape, as being defined, e.g., by (Lipschitz-)continuous nonlinear functions. Frequently, sophisticated algorithm enhancements are added to this basic strategy. For instance, the clustering of sample points attempts to select only a single point from each sampled "basin" of f from which then a local search method is initiated.

2. Evolution strategies: These methods heuristically "mimic" biological evolution: namely, the process of natural selection and the "survival of the fittest" principle. An adaptive search procedure based on a "population" of candidate solution points is used. Iterations involve a competitive selection that drops the poorer solutions. The remaining pool of candidates with higher "fitness value" are then "recombined" with other solutions by swapping components with another; they can also be "mutated" by making some smaller-scale change to a candidate. The recombination and mutation moves are applied sequentially; their aim is to generate new solutions that are biased towards subsets of D in which good, although not necessarily globally optimized, solutions have already been found. Numerous variants of this general strategy based on diverse evolution "game rules" can be constructed. The different types of evolutionary search methods include approaches that are aimed at continuous global optimization problems,

and also others that are targeted towards solving combinatorial problems. The latter group is often called genetic algorithms.

3. simulated annealing: These techniques are based upon the physical analogy of cooling crystal structures that spontaneously attempt to arrive at some stable (globally or locally minimal potential energy) equilibrium. This general principle is applicable to both discrete and continuous global optimization problems under mild structural requirements.

4. tabu search: In this general category of meta-heuristics, the essential idea is to "forbid" search moves to points already visited in the (usually discrete) search space, at least for the upcoming few steps. That is, one can temporarily accept new inferior solutions, in order to avoid paths already investigated. This approach can lead to exploring new regions of D , with the goal of finding a solution by "globalized" search. Tabu search has traditionally been applied to combinatorial optimization (e.g., scheduling, routing, traveling salesman) problems. The technique can be made, at least in principle, directly applicable to continuous global optimization problems by a discrete approximation (encoding) of the problem, but other extensions are also possible.

5. approximate convex global underestimation: This heuristically attractive strategy attempts to estimate the (postulated) large scale, "overall" convexity characteristics of the objective function f based on directed sampling in D . Applicable to smooth problems.

6. continuation methods: These first transform the potential function into a more smooth ("simpler") function which has fewer local minimizers, and then use a local minimization procedure to trace the minimizers back to the original function. Again, this methodology is applicable to smooth problems..

7. sequential improvement of local optima: These usually operate on adaptively constructed auxiliary functions, to assist the search for gradually better optima. The general heuristic principle includes tunneling, deflation, and filled function approaches.

Global Optimization vs. Local Optimization

Global optimization refers to finding the optimal value of a given function among all possible solution whereas **local optimization** finds the optimal value within the neighboring set of candidate solution.

DIWAKAR EDUCATION HUB

Theory of Computation and Compilers Unit – 8 MCQs

As per updated syllabus

DIWAKAR EDUCATION HUB

2020

THE LEARN WITH EXPERTIES

Theory of Computation and Compilers Unit – 8 MCQs

1. Assume the R is a relation on a set A, aRb is partially ordered such that a and b are

- a) reflexive
- b) transitive
- c) symmetric
- d) reflexive and transitive

Answer: d

Explanation: A partially ordered relation refers to one which is Reflexive, Transitive and Antisymmetric.

2. The non- Kleene Star operation accepts the following string of finite length over set $A = \{0,1\}$ | where string s contains even number of 0 and 1

- a) 01,0011,010101
- b) 0011,11001100
- c) ϵ ,0011,11001100
- d) ϵ ,0011,11001100

Answer: b

Explanation: The Kleene star of A, denoted by A^* , is the set of all strings obtained by concatenating zero or more strings from A.

3. A regular language over an alphabet Σ is one that cannot be obtained from the basic languages using the operation

- a) Union
- b) Concatenation
- c) Kleene*
- d) All of the mentioned

Answer: d

Explanation: Union, Intersection, Concatenation, Kleene*, Reverse are all the closure properties of Regular Language.

4. Statement 1: A Finite automata can be represented graphically; Statement 2: The nodes can be its states; Statement 3: The edges or arcs can be used for transitions
Hint: Nodes and Edges are for trees and

forests too.

Which of the following make the correct combination?

- a) Statement 1 is false but Statement 2 and 3 are correct
- b) Statement 1 and 2 are correct while 3 is wrong
- c) None of the mentioned statements are correct
- d) All of the mentioned

Answer: d

Explanation: It is possible to represent a finite automaton graphically, with nodes for states, and arcs for transitions.

5. The minimum number of states required to recognize an octal number divisible by 3 are/is

- a) 1
- b) 3
- c) 5
- d) 7

Answer: b

Explanation: According to the question, minimum of 3 states are required to recognize an octal number divisible by 3.

6. Which of the following is a not a part of 5-tuple finite automata?

- a) Input alphabet
- b) Transition function
- c) Initial State
- d) Output Alphabet

Answer: d

Explanation: A FA can be represented as $FA = (Q, \Sigma, \delta, q_0, F)$ where Q =Finite Set of States, Σ =Finite Input Alphabet, δ =Transition Function, q_0 =Initial State, F =Final/Acceptance State).

7. If an Infinite language is passed to Machine M, the subsidiary which gives a

Theory of Computation and Compilers Unit – 8 MCQs

finite solution to the infinite input tape is _____

- a) Compiler
- b) Interpreter
- c) Loader and Linkers
- d) None of the mentioned

Answer: a

Explanation: A Compiler is used to give a finite solution to an infinite phenomenon. Example of an infinite phenomenon is Language C, etc.

8. The number of elements in the set for the Language $L = \{x \in (\Sigma^r)^* \mid \text{length of } x \text{ is at most } 2\}$ and $\Sigma = \{0,1\}$ is _____

- a) 7
- b) 6
- c) 8
- d) 5

Answer: a

Explanation: $\Sigma^r = \{1,0\}$ and a Kleene* operation would lead to the following set = COUNT $\{\epsilon, 0, 1, 00, 11, 01, 10\} = 7$.

9. For the following change of state in FA, which of the following codes is an incorrect option?

- a) $\delta(m, 1) = n$
- b) $\delta(0, n) = m$
- c) $\delta(m, 0) = \epsilon$
- d) s: accept = false; cin >> char;
if char = "0" goto n;

Answer: b

Explanation: $\delta(QX\Sigma) = Q1$ is the correct representation of change of state. Here, δ is called the Transition function.

10. Given: $\Sigma = \{a, b\}$

$L = \{x \in \Sigma^* \mid x \text{ is a string combination}\}$

Σ^4 represents which among the following?

- a) $\{aa, ab, ba, bb\}$
- b) $\{aaaa, abab, \epsilon, abaa, aabb\}$
- c) $\{aaa, aab, aba, bbb\}$
- d) All of the mentioned

Answer: b

Explanation: Σ^* represents any combination of the given set while Σ^x represents the set of combinations with length x where $x \in I$.

11. Moore Machine is an application of:

- a) Finite automata without input
- b) Finite automata with output
- c) Non- Finite automata with output
- d) None of the mentioned

Answer: b

Explanation: Finite automaton with an output is categorized in two parts: Moore M/C and Mealy M/C.

12. In Moore machine, output is produced over the change of:

- a) transitions
- b) states
- c) Both
- d) None of the mentioned

Answer: b

Explanation: Moore machine produces an output over the change of transition states while mealy machine does it so for transitions itself.

13. In mealy machine, the O/P depends upon?

- a) State
- b) Previous State
- c) State and Input
- d) Only Input

Theory of Computation and Compilers Unit – 8 MCQs

Answer: c

Explanation: Definition of Mealy Machine.

14. Which of the given are correct?

- a) Moore machine has 6-tuples
- b) Mealy machine has 6-tuples
- c) Both Mealy and Moore has 6-tuples
- d) None of the mentioned

Answer: c

Explanation: Finite Automaton with Output has a common definition for both the categories.

15. Which of the following does not belong to input alphabet if $S=\{a, b\}^*$ for any language?

- a) a
- b) b
- c) ε
- d) none of the mentioned

Answer: c

Explanation: The automaton may be allowed to change its state without reading the input symbol using epsilon but this does not mean that epsilon has become an input symbol. On the contrary, one assumes that the symbol epsilon does not belong to any alphabet.

16. Which of the following not an example Bounded Information?

- a) fan switch outputs {on, off}
- b) electricity meter reading
- c) colour of the traffic light at the moment
- d) none of the mentioned

Answer: b

Explanation: Bounded information refers to

one whose output is limited and it cannot be said what were the recorded outputs previously until memorized.

17. A Language for which no DFA exist is a_____

- a) Regular Language
- b) Non-Regular Language
- c) May be Regular
- d) Cannot be said

Answer: b

Explanation: A language for which there is no existence of a deterministic finite automata is always Non Regular and methods like Pumping Lemma can be used to prove the same.

18. A DFA cannot be represented in the following format

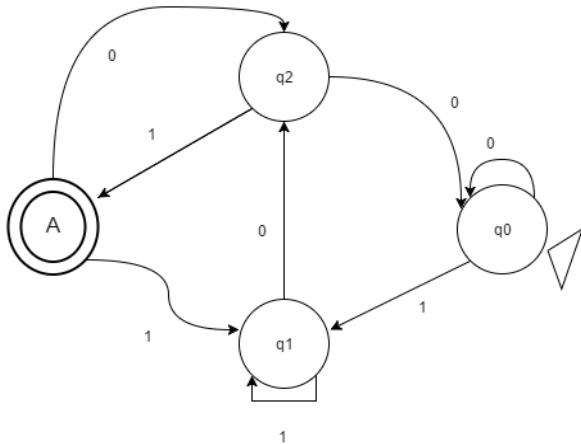
- a) Transition graph
- b) Transition Table
- c) C code
- d) None of the mentioned

Answer: d

Explanation: A DFA can be represented in the following formats: Transition Graph, Transition Table, Transition tree/forest/Any programming Language.

Theory of Computation and Compilers Unit – 8 MCQs

19. What the following DFA accepts?



- a) x is a string such that it ends with '101'
- b) x is a string such that it ends with '01'
- c) x is a string such that it has odd 1's and even 0's
- d) x is a strings such that it has starting and ending character as 1

Answer: a

Explanation: Strings such as {1101,101,10101} are being accepted while {1001,11001} are not. Thus, this conclusion leads to option a.

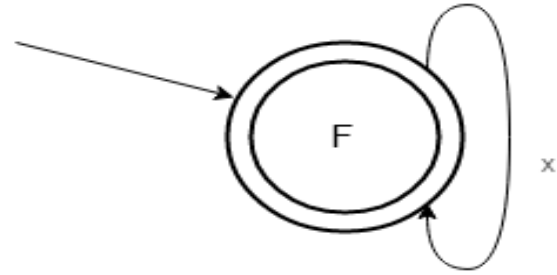
20. When are 2 finite states equivalent?

- a) Same number of transitions
- b) Same number of states
- c) Same number of states as well as transitions
- d) Both are final states

Answer: c

Explanation: Two states are said to be equivalent if and only if they have same number of states as well as transitions.

21. What does the following figure most correctly represents?

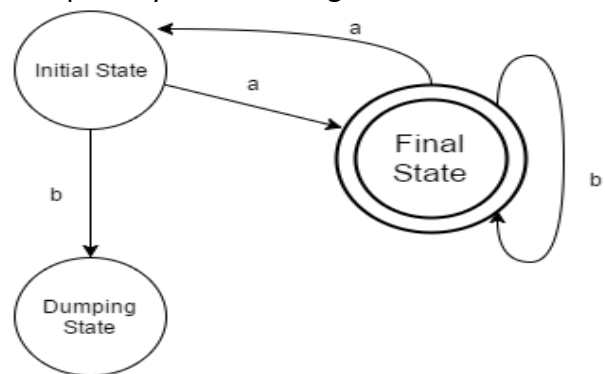


- a) Final state with loop x
- b) Transitional state with loop x
- c) Initial state as well as final state with loop x
- d) Insufficient Data

Answer: c

Explanation: The figure represents the initial as well as the final state with an iteration of x.

22. Which of the following will not be accepted by the following DFA?



- a) ababaabaa
- b) abbbaa
- c) abbbaabb
- d) abbaabbbaa

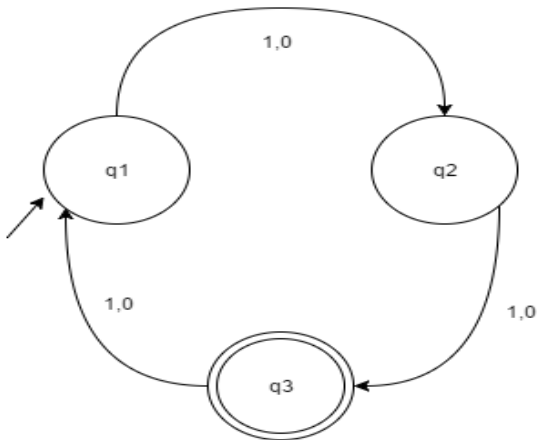
Answer: a

Explanation: All the Strings are getting accepted except 'ababaabaa' as it is directed to dumping state. Dumping state

Theory of Computation and Compilers Unit – 8 MCQs

also refers to the reject state of the automata.

23. Which of the following will the given DFA won't accept?



- a) ϵ
- b) 11010
- c) 10001010
- d) String of letter count 11

Answer: a

Explanation: As the initial state is not made an acceptance state, thus ϵ will not be accepted by the given DFA. For the automata to accept ϵ as an entity, one should make the initial state as also the final state.

24. The password to the admins account="administrator". The total number of states required to make a password-pass system using DFA would be _____

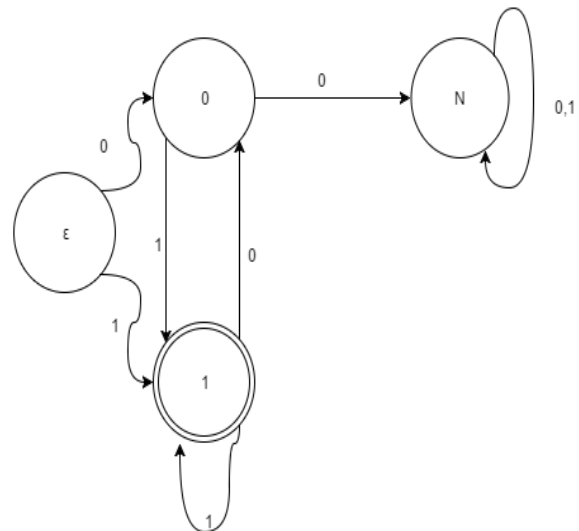
- a) 14 states
- b) 13 states
- c) 12 states
- d) A password pass system cannot be created using DFA

Answer: a

Explanation: For a string of n characters

with no repetitive substrings, the number of states required to pass the string is $n+1$.

25. Which of the following is the corresponding Language to the given DFA?



- a) $L = \{x \in \{0, 1\}^* \mid x \text{ ends in } 1 \text{ and does not contain substring } 01\}$
- b) $L = \{x \in \{0, 1\}^* \mid x \text{ ends in } 1 \text{ and does not contain substring } 00\}$
- c) $L = \{x \in \{0, 1\}^* \mid x \text{ ends in } 1 \text{ and does not contain substring } 00\}$
- d) $L = \{x \in \{0, 1\}^* \mid x \text{ ends in } 1 \text{ and does not contain substring } 11\}$

Answer: b

Explanation: The Language can be anonymously checked and thus the answer can be predicted. The language needs to be accepted by the automata (acceptance state) in order to prove its regularity.

26. Let $\Sigma = \{a, b, \dots, z\}$ and $A = \{\text{Hello, World}\}$, $B = \{\text{Input, Output}\}$, then $(A^* \cap B) \cup (B^* \cap A)$ can be represented as:

- a) $\{\text{Hello, World, Input, Output, } \epsilon\}$
- b) $\{\text{Hello, World, } \epsilon\}$

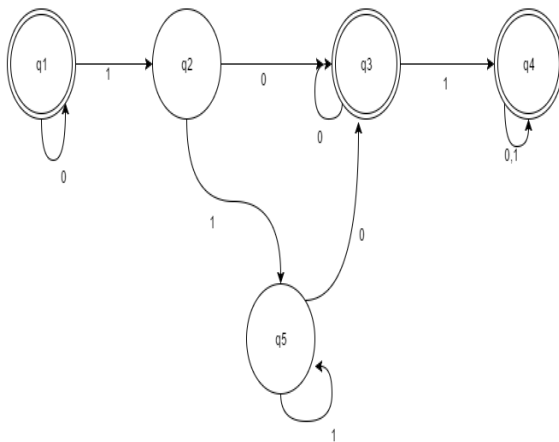
Theory of Computation and Compilers Unit – 8 MCQs

- c) {Input, Output, ϵ }
d) {}

Answer: d

Explanation: Union operation creates the universal set by combining all the elements of first and second set while intersection operation creates a set of common elements of the first and the second state.

27. Let the given DFA consist of x states.
Find x-y such that y is the number of states on minimization of DFA?



- a) 3
b) 2
c) 1
d) 4

Answer: b

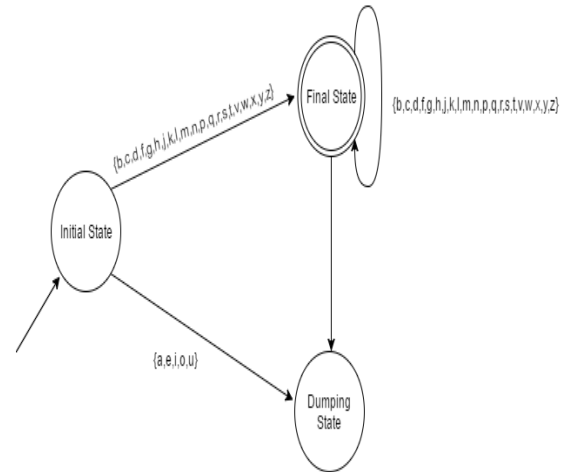
Explanation: Use the equivalence theorem or Myhill Nerode theorem to minimize the DFA.

28. For a machine to surpass all the letters of alphabet excluding vowels, how many number of states in DFA would be required?

- a) 3
b) 2
c) 22
d) 27

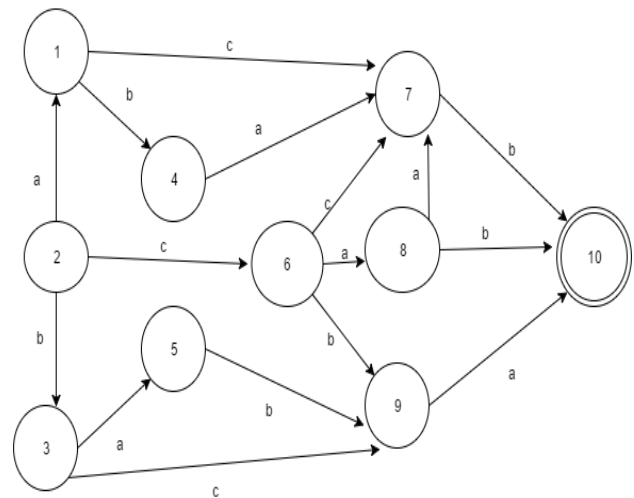
Answer: a

Explanation:



29. For the DFA given below compute the following:

Union of all possible combinations at state 7,8 and 9.



- a) {aba, ac, cc, ca, cb, bc, bab, ca}
b) {bab, bc, ac, aba, ca, aac, ccb}
c) {cc, ca, cb, aba, bab, ac}
d) {aba, ac, cc, ca, cb, bc, bab, caa}

Answer: d

Explanation: The string a state receives is the combination of all input alphabets which lie across the path covered.

Theory of Computation and Compilers Unit – 8 MCQs

30. Given $L = \{x \in \Sigma^* \mid x \text{ has equal number of } a, s \text{ and } b's\}$.

Which of the following property satisfy the regularity of the given language?

- a) Regularity is dependent upon the length of the string
- b) Regularity is not dependent upon the length of the string
- c) Can't be said for a particular string of a language
- d) It may depend on the length of the string

Answer: b

Explanation: DFA can be made for infinite language with an infinite length. Thus, dependency over length is unfruitful.

31. Given:

$L = \{x \in \Sigma^* \mid x = 0^n 1^n \text{ for } n \geq 1\}$; Can there be a DFA possible for the language?

- a) Yes
- b) No

Answer: b

Explanation: It is not possible to have a count of equal number of 0 and 1 at any instant in DFA. Thus, It is not possible to build a DFA for the given Language.

32. $\delta(A, 1) = B$, $\delta(A, 0) = A$

$\Delta(B, (0, 1)) = C$

$\delta(C, 0) = A$ (Initial state = A)

String="011001" is transit at which of the states?

- a) A
- b) C
- c) B
- d) Invalid String

Answer: a

Explanation: It is easy and simple to create the table and then the corresponding transition graph in order to get the result, at which state the given string would be accepted.

33. How many languages are over the alphabet R?

- a) countably infinite
- b) countably finite
- c) uncountable finite
- d) uncountable infinite

Answer: d

Explanation: A language over an alphabet R is a set of strings over A which is uncountable and infinite.

34. According to the 5-tuple representation i.e. $FA = \{Q, \Sigma, \delta, q, F\}$

Statement 1: $q \in Q'$; Statement 2: $F \in Q$

- a) Statement 1 is true, Statement 2 is false
- b) Statement 1 is false, Statement 2 is true
- c) Statement 1 is false, Statement 2 may be true
- d) Statement 1 may be true, Statement 2 is false

Answer: b

Explanation: Q is the Finite set of states, whose elements i.e. the states constitute the finite automata.

35. δ^* tells us the best:

- a) how the DFA S behaves on a word u
- b) the state is the dumping state
- c) the final state has been reached
- d) Kleene operation is performed on the set

Theory of Computation and Compilers Unit – 8 MCQs

Answer: a

Explanation: δ or the Transition function describes the best, how a DFA behaves on a string where to transit next, which direction to take.

36. Which of the following option is correct?

$A = \{ \{abc, aaba\}. \{ \epsilon, a, bb \} \}$

a) $abcbb \notin A$

b) $\epsilon \notin A$

c) ϵ may not belong to A

d) $abca \notin A$

Answer: b

Explanation: As the question has dot operation, ϵ will not be a part of the concatenated set. Had it been a union operation, ϵ would be a part of the operated set.

37. For a DFA accepting binary numbers whose decimal equivalent is divisible by 4, what are all the possible remainders?

a) 0

b) 0,2

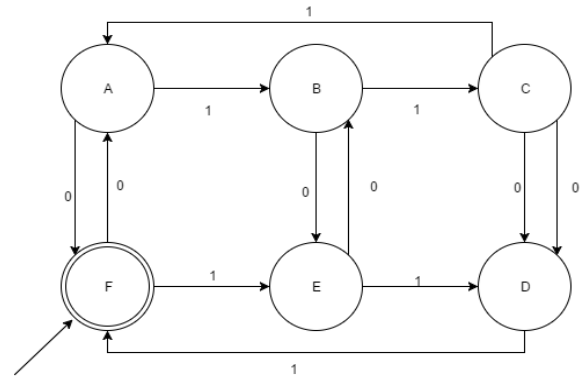
c) 0,2,4

d) 0,1,2,3

Answer: d

Explanation: All the decimal numbers on division would lead to only 4 remainders i.e. 0,1,2,3 (Property of Decimal division).

38. Which of the following x is accepted by the given DFA (x is a binary string $\Sigma = \{0,1\}$)?



a) divisible by 3

b) divisible by 2

c) divisible by 2 and 3

d) divisible by 3 and 2

Answer: d

Explanation: The given DFA accepts all the binary strings such that they are divisible by 3 and 2. Thus, it can be said that it also accepts all the strings which are divisible by 6.

39. Given:

$L1 = \{x \in \Sigma^* \mid x \text{ contains even no's of 0's}\}$

$L2 = \{x \in \Sigma^* \mid x \text{ contains odd no's of 1's}\}$

No of final states in Language $L1 \cup L2$?

a) 1

b) 2

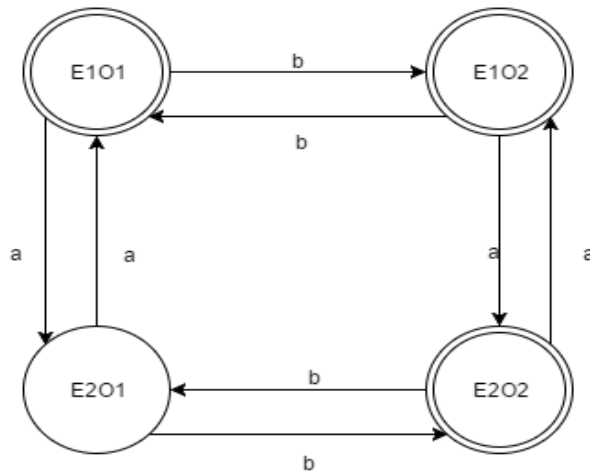
c) 3

d) 4

Theory of Computation and Compilers Unit – 8 MCQs

Answer: c

Explanation:



40. The maximum number of transition which can be performed over a state in a DFA?

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

- a) 1
- b) 2
- c) 3
- d) 4

Answer: c

Explanation: The maximum number of transitions which a DFA allows for a language is the number of elements the transitions constitute.

41. The maximum sum of in degree and out degree over a state in a DFA can be determined as:

$\Sigma = \{a, b, c, d\}$

- a) 4+4
- b) 4+16
- c) 4+0
- d) depends on the Language

Answer: d

Explanation: The out degree for a DFA I

fixed while the in degree depends on the number of states in the DFA and that cannot be determined without the dependence over the Language.

42. Transition function maps.

- a) $\Sigma * Q \rightarrow \Sigma$
- b) $Q * Q \rightarrow \Sigma$
- c) $\Sigma * \Sigma \rightarrow Q$
- d) $Q * \Sigma \rightarrow Q$

Answer: d

Explanation: Inputs are state and input string output is states.

43. Number of states require to accept string ends with 10.

- a) 3
- b) 2
- c) 1
- d) can't be represented.

Answer: a

Explanation: This is minimal finite automata.

44. Extended transition function is .

- a) $Q * \Sigma^* \rightarrow Q$
- b) $Q * \Sigma \rightarrow Q$
- c) $Q^* * \Sigma^* \rightarrow \Sigma$
- d) $Q * \Sigma \rightarrow \Sigma$

Answer: a

Explanation: This takes single state and string of input to produce a state.

45. $\delta^*(q, ya)$ is equivalent to .

- a) $\delta((q, y), a)$
- b) $\delta(\delta^*(q, y), a)$
- c) $\delta(q, ya)$
- d) independent from δ notation

Theory of Computation and Compilers Unit – 8 MCQs

Answer:b

Explanation: First it parse y string after that it parse a.

46. String X is accepted by finite automata if.

- a) $\delta^*(q, x) \in A$
- b) $\delta(q, x) \in A$
- c) $\delta^*(Q_0, x) \in A$
- d) $\delta(Q_0, x) \in A$

Answer:c

Explanation: If automata starts with starting state and after finite moves it reaches to final step then it called accepted.

47. Languages of a automata is

- a) If it is accepted by automata
- b) If it halts
- c) If automata touch final state in its life time
- d) All language are language of automata

Answer:a

Explanation: If a string accepted by automata it is called language of automata.

48. Language of finite automata is.

- a) Type 0
- b) Type 1
- c) Type 2
- d) Type 3

Answer:d

Explanation: According to Chomsky classification.

49. Finite automata requires minimum _____ number of stacks.

- a) 1
- b) 0

c) 2

d) None of the mentioned

Answer:b

Explanation: Finite automata doesn't require any stack operation .

50. Number of final state require to accept Φ in minimal finite automata.

- a) 1
- b) 2
- c) 3
- d) None of the mentioned

Answer:d

Explanation: No final state requires.

51. Given Language: $L = \{ab \cup aba\}^*$

If X is the minimum number of states for a DFA and Y is the number of states to construct the NFA,
 $|X - Y| = ?$

- a) 2
- b) 3
- c) 4
- d) 1

Answer: a

Explanation: Construct the DFA and NFA individually, and then attain the difference of states.

52. An automaton that presents output based on previous state or current input:

- a) Acceptor
- b) Classifier
- c) Transducer
- d) None of the mentioned.

Answer: c

Explanation: A transducer is an automaton

Theory of Computation and Compilers Unit – 8 MCQs

that produces an output on the basis of what input has been given currently or previous state.

53. If NFA of 6 states excluding the initial state is converted into DFA, maximum possible number of states for the DFA is ?

- a) 64
- b) 32
- c) 128
- d) 127

Answer: c

Explanation: The maximum number of sets for DFA converted from NFA would be not greater than 2^n .

54. NFA, in its name has 'non-deterministic' because of :

- a) The result is undetermined
- b) The choice of path is non-deterministic
- c) The state to be transited next is non-deterministic
- d) All of the mentioned

Answer: b

Explanation: Non deterministic or deterministic depends upon the definite path defined for the transition from one state to another or undefined(multiple paths).

55. Which of the following is correct proposition?

Statement 1: Non determinism is a generalization of Determinism.

Statement 2: Every DFA is automatically an NFA

a) Statement 1 is correct because Statement 2 is correct

b) Statement 2 is correct because Statement 2 is correct

c) Statement 2 is false and Statement 1 is false

d) Statement 1 is false because Statement 2 is false

Answer: b

Explanation: DFA is a specific case of NFA.

56. Given Language $L = \{x \in \{a, b\}^* \mid x \text{ contains aba as its substring}\}$

Find the difference of transitions made in constructing a DFA and an equivalent NFA?

- a) 2
- b) 3
- c) 4
- d) Cannot be determined.

Answer: a

Explanation: The individual Transition graphs can be made and the difference of transitions can be determined.

57. The construction time for DFA from an equivalent NFA (m number of node) is:

- a) $O(m^2)$
- b) $O(2^m)$
- c) $O(m)$
- d) $O(\log m)$

Answer: b

Explanation: From the coded NFA-DFA conversion.

58. If n is the length of Input string and m is the number of nodes, the running time of DFA is x that of NFA. Find x?

Theory of Computation and Compilers Unit – 8 MCQs

- a) $1/m^2$
- b) 2^m
- c) $1/m$
- d) $\log m$

Answer: a

Explanation: Running time of DFA: $O(n)$ and Running time of NFA $= O(m^2n)$.

59. L is a regular Language if and only if the set of _____ classes of IL is finite.

- a) Equivalence
- b) Reflexive
- c) Myhill
- d) Nerode

Answer: a

Explanation: According to Myhill Nerode theorem, the corollary proves the given statement correct for equivalence classes.

60. A language can be generated from simple primitive language in a simple way if and only if

- a) It is recognized by a device of infinite states
- b) It takes no auxiliary memory
- c) Both are correct
- d) Both are wrong

Answer: b

Explanation: A language is regular if and only if it can be accepted by a finite automaton. Secondly, It supports no concept of auxiliary memory as it loses the data as soon as the device is shut down.

61. Which of the following does not represents the given language?

Language: $\{0,01\}$

- a) $0+01$
- b) $\{0\} \cup \{01\}$
- c) $\{0\} \cup \{0\}\{1\}$
- d) $\{0\}^* \{01\}$

Answer: d

Explanation: The given option represents $\{0, 01\}$ in different forms using set operations and Regular Expressions. The operator like * , $+$, etc. are logical operation and they form invalid regular expressions when used.

62. According to the given language, which among the following expressions does it corresponds to?

Language $L = \{x \in \{0,1\}^* \mid x \text{ is of length 4 or less}\}$

- a) $(0+1+0+1+0+1+0+1)^4$
- b) $(0+1)^4$
- c) $(01)^4$
- d) $(0+1+\epsilon)^4$

Answer: d

Explanation: The extended notation would be $(0+1)^4$ but however, we may allow some or all the factors to be ϵ . Thus ϵ needs to be included in the given regular expression.

63. Which among the following looks similar to the given expression?

$((0+1). (0+1))^*$

- a) $\{x \in \{0,1\}^* \mid x \text{ is all binary number with even length}\}$
- b) $\{x \in \{0,1\}^* \mid x \text{ is all binary number with even length}\}$
- c) $\{x \in \{0,1\}^* \mid x \text{ is all binary number with odd length}\}$
- d) $\{x \in \{0,1\}^* \mid x \text{ is all binary number with odd length}\}$

Theory of Computation and Compilers Unit – 8 MCQs

Answer: a

Explanation: The given regular expression corresponds to a language of binary strings which is of even length including a length of 0.

64. A finite automaton accepts which type of language:

- a) Type 0
- b) Type 1
- c) Type 2
- d) Type 3

Answer: d

Explanation: Type 3 refers to Regular Languages which is accepted by a finite automaton.

65. Which among the following are incorrect regular identities?

- a) $\epsilon R = R$
- b) $\epsilon^* = \epsilon$
- c) $\Phi^* = \epsilon$
- d) $R\Phi = R$

Answer: d

Explanation: There are few identities over Regular Expressions which include:

$$R\Phi = \Phi R = \Phi \neq R$$

66. Simplify the following regular expression:

$$\epsilon + 1^*(011)^*(1^*(011)^*)^*$$

- a) $(1+011)^*$
- b) $(1^*(011)^*)^*$
- c) $(1+(011)^*)^*$
- d) $(1011)^*$

Answer: a

$$\text{Explanation: } \epsilon + 1^*(011)^*(1^*(011)^*)^*$$

$$\epsilon + RR^* = \epsilon + R^*R = \epsilon + R^+ = R^*$$

67. P, O, R be regular expression over Σ , P is not ϵ , then

$R = Q + RP$ has a unique solution:

- a) Q^*P
- b) QP^*
- c) Q^*P^*
- d) $(P^*O^*)^*$

Answer: b

Explanation: The given statement is the Arden's Theorem and it tends to have a unique solution as QP^* .

Let P and Q be regular expressions,

$$R = Q + RP$$

$$R = Q + (Q + RP)P$$

$$R = Q + ((Q + RP) + RP)$$

$$+ P = Q + QP + RPP + RPP = Q + QP + (Q + RP)$$

$$PP + (Q + RP)$$

$$PP = Q + QP + QPP + RPPP + QPP + RPPP,$$

If we do this recursively, we get:

$$R = QP^*$$

68. Arden's theorem is true for:

- a) More than one initial states
- b) Null transitions
- c) Non-null transitions
- d) None of the mentioned

Answer: c

Explanation: Arden's theorem strictly assumes the following;

- a) No null transitions in the transition diagrams
- b) True for only single initial state

Theory of Computation and Compilers Unit – 8 MCQs

69. Which of the following is correct?

Statement 1: ϵ represents a single string in the set.

Statement 2: Φ represents the language that consist of no string.

- a) Statement 1 and 2 both are correct
- b) Statement 1 is false but 2 is correct
- c) Statement 1 and 2 both are false
- d) There is no difference between both the statements, ϵ and Φ are different notation for same reason

Answer: a

Explanation: ϵ represents a single string in the set namely, the empty string while Statement 2 is also correct.

70. The appropriate precedence order of operations over a Regular Language is

- a) Kleene, Union, Concatenate
- b) Kleene, Star, Union
- c) Kleene, Dot, Union
- d) Star, Union, Dot

Answer: c

Explanation: If a regular language expression is given, the appropriate order of precedence if the parenthesis is ignored is: Star or Kleene, Dot or Concatenation, Union or Plus.

71. Regular Expression R and the language it describes can be represented as:

- a) R, R(L)
- b) L(R), R(L)
- c) R, L(R)
- d) All of the mentioned

Answer: c

Explanation: When we wish to distinguish

between a regular expression R and the language it represents; we write L(R) to be the language of R.

72. Let for $\Sigma = \{0,1\}$ $R = (\Sigma\Sigma\Sigma)^*$, the language of R would be

- a) $\{w \mid w \text{ is a string of odd length}\}$
- b) $\{w \mid w \text{ is a string of length multiple of 3}\}$
- c) $\{w \mid w \text{ is a string of length 3}\}$
- d) All of the mentioned

Answer: b

Explanation: This regular expression can be used to eliminate the answers and get the result. The length can be even and as well more than 3 when $R = (\Sigma\Sigma\Sigma)(\Sigma\Sigma\Sigma)$ (particular case).

73. If $\Sigma = \{0,1\}$, then Φ^* will result to:

- a) ϵ
- b) Φ
- c) Σ
- d) None of the mentioned

Answer: a

Explanation: The star operation brings together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, resulting only the empty string.

74. Which of the following is correct?

Statement 1: ϵ represents a single string in the set.

Statement 2: Φ represents the language that consist of no string.

- a) Statement 1 and 2 both are correct
- b) Statement 1 is false but 2 is correct
- c) Statement 1 and 2 both are false

Theory of Computation and Compilers Unit – 8 MCQs

d) There is no difference between both the statements, ϵ and Φ are different notation for same reason

Answer: a

Explanation: ϵ represents a single string in the set namely, the empty string while Statement 2 is also correct.

75. The appropriate precedence order of operations over a Regular Language is

- a) Kleene, Union, Concatenate
- b) Kleene, Star, Union
- c) Kleene, Dot, Union
- d) Star, Union, Dot

Answer: c

Explanation: If a regular language expression is given, the appropriate order of precedence if the parenthesis is ignored is: Star or Kleene, Dot or Concatenation, Union or Plus.

76. Regular Expression R and the language it describes can be represented as:

- a) R, R(L)
- b) L(R), R(L)
- c) R, L(R)
- d) All of the mentioned

Answer: c

Explanation: When we wish to distinguish between a regular expression R and the language it represents; we write L(R) to be the language of R.

77. Let for $\Sigma = \{0,1\}$ $R = (\Sigma\Sigma\Sigma)^*$, the language of R would be

- a) $\{w \mid w \text{ is a string of odd length}\}$
- b) $\{w \mid w \text{ is a string of length multiple of 3}\}$

c) $\{w \mid w \text{ is a string of length 3}\}$

d) All of the mentioned

Answer: b

Explanation: This regular expression can be used to eliminate the answers and get the result. The length can be even and as well more than 3 when $R = (\Sigma\Sigma\Sigma)(\Sigma\Sigma\Sigma)$ (particular case).

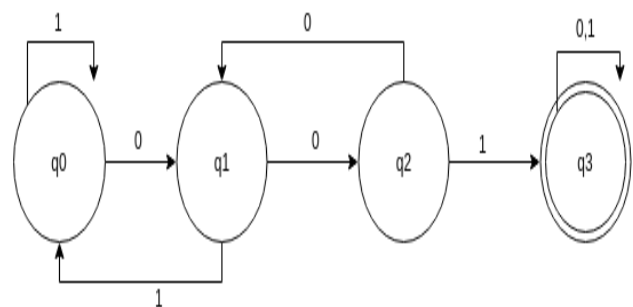
78. If $\Sigma = \{0,1\}$, then Φ^* will result to:

- a) ϵ
- b) Φ
- c) Σ
- d) None of the mentioned

Answer: a

Explanation: The star operation brings together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, resulting only the empty string.

79. Which of the following is same as the given DFA?



- a) $(0+1)^*001(0+1)^*$
- b) $1^*001(0+1)^*$
- c) $(01)^*(0+0+1)(01)^*$
- d) None of the mentioned

Theory of Computation and Compilers Unit – 8 MCQs

Answer: a

Explanation: There needs to be 001 together in the string as an essential substring. Thus, the other components can be anything, 0 or 1 or e.

80. Which of the following statements is not true?

- a) Every language defined by any of the automata is also defined by a regular expression
- b) Every language defined by a regular expression can be represented using a DFA
- c) Every language defined by a regular expression can be represented using NFA with e moves
- d) Regular expression is just another representation for any automata definition

Answer: b

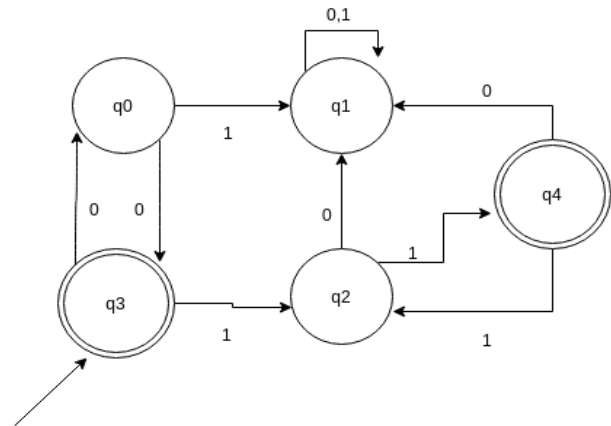
Explanation: Using NFA with e moves, we can represent all the regular expressions as an automata. As regular expressions include e, we need to use e moves.

81. The total number of states required to automate the given regular expression $(00)^*(11)^*$

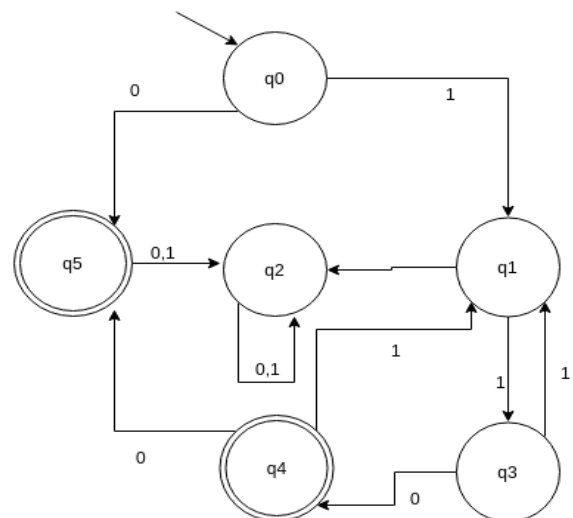
- a) 3
- b) 4
- c) 5
- d) 6

Answer: c

Explanation:



82. Which of the given regular expressions correspond to the automata shown?



- a) $(110+1)^*0$
- b) $(11+110)^*1$
- c) $(110+11)^*0$
- d) $(1+110)^*1$

Answer: c

Explanation: There is no state change for union operation, but has two different paths while for concatenation or dot operation, we have a state change for every element of the string.

Theory of Computation and Compilers Unit – 8 MCQs

83. Which of the following is an utility of state elimination phenomenon?

- a) DFA to NFA
- b) NFA to DFA
- c) DFA to Regular Expression
- d) All of the mentioned

Answer: c

Explanation: We use this algorithm to simplify a finite automaton to regular expression or vice versa. We eliminate states while converting a given finite automata to its corresponding regular expression.

84. If we have more than one accepting states or an accepting state with an outdegree, which of the following actions will be taken?

- a) addition of new state
- b) removal of a state
- c) make the newly added state as final
- d) more than one option is correct

Answer: d

Explanation: If there is more than one accepting state or if the single accepting state as an out degree, add a new accepting state, make all other states non accepting, and hold an e-transitions from each former accepting state to the new accepting state.

85. Which of the following is not a step in elimination of states procedure?

- a) Unifying all the final states into one using e-transitions
- b) Unify single transitions to multi transitions that contains union of input
- c) Remove states until there is only starting

and accepting states

- d) Get the resulting regular expression by direct calculation

Answer: b

Explanation: While eliminating the states, we unify multiple transitions to one transition that contains union of input and not the vice versa.

86. A regular language over an alphabet a is one that can be obtained from

- a) union
- b) concatenation
- c) kleene
- d) All of the mentioned

Answer: d

Explanation: None.

87. Regular expression $\{0,1\}$ is equivalent to

- a) $0 \cup 1$
- b) $0 / 1$
- c) $0 + 1$
- d) All of the mentioned

Answer: d

Explanation: All are equivalent to union operation.

88. Precedence of regular expression in decreasing order is

- a) $*$, $.$, $+$
- b) $.$, $*$, $+$
- c) $.$, $+$, $*$
- d) $+$, a , $*$

Answer: a

Explanation: None.

Theory of Computation and Compilers Unit – 8 MCQs

89. Regular expression Φ^* is equivalent to

- a) ϵ
- b) Φ
- c) 0
- d) 1

Answer: a

Explanation: None.

90. $a^?$ is equivalent to

- a) a
- b) $a+\Phi$
- c) $a+\epsilon$
- d) wrong expression

Answer: c

Explanation: Zero or one time repetition of previous character .

91. ϵL is equivalent to

- a) ϵ
- b) Φ
- c) L
- d) $L\epsilon$

Answer: c,d

Explanation: None.

92. $(a+b)^*$ is equivalent to

- a) b^*a^*
- b) $(a^*b^*)^*$
- c) a^*b^*
- d) none of the mentioned

Answer: b

Explanation: None.

93. ΦL is equivalent to

- a) $L\Phi$
- b) Φ

c) L

d) ϵ

Answer: a,b

Explanation: None.

94. Which of the following pair of regular expression are not equivalent?

- a) $1(01)^*$ and $(10)^*1$
- b) $x(xx)^*$ and $(xx)^*x$
- c) $(ab)^*$ and a^*b^*
- d) x^+ and x^*x^+

Answer: c

Explanation: $(ab)^* = (a^*b^*)^*$.

95. Consider following regular expression

i) $(a/b)^*$ ii) $(a^*/b^*)^*$ iii) $((\epsilon/a)b^*)^*$

Which of the following statements is correct

- a) i,ii are equal and ii,iii are not
- b) i,ii are equal and i,iii are not
- c) ii,iii are equal and i,ii are not
- d) all are equal

Answer: d

Explanation: All are equivalent to $(a+b)^*$.

96. How many strings of length less than 4 contains the language described by the regular expression $(x+y)^*y(a+ab)^*$?

- a) 7
- b) 10
- c) 12
- d) 11

Answer: c

Explanation: string of length 0 = Not possible (because y is always present).

string of length 1 = 1 (y)

string of length 2 = 3 (xy,yy,ya)

Theory of Computation and Compilers Unit – 8 MCQs

string of length 3 = 8

(xxy,xyy,yxy,yyy,yaa,yab,xya,yya)

97. Which of the following is true?

- a) $(01)^*0 = 0(10)^*$
- b) $(0+1)^*0(0+1)^*1(0+1) = (0+1)^*01(0+1)^*$
- c) $(0+1)^*01(0+1)^*+1^*0^* = (0+1)^*$
- d) All of the mentioned

Answer: d

Explanation: None.

98. A language is regular if and only if

- a) accepted by DFA
- b) accepted by PDA
- c) accepted by LBA
- d) accepted by Turing machine

Answer: a

Explanation: All of above machine can accept regular language but all string accepted by machine is regular only for DFA.

99. Regular grammar is

- a) context free grammar
- b) non context free grammar
- c) english grammar
- d) none of the mentioned

Answer: a

Explanation: Regular grammar is subset of context free grammar.

100. Which of the following is not a regular expression?

- a) $[(a+b)^*-(aa+bb)]^*$
- b) $[(0+1)-(0b+a1)^*(a+b)]^*$
- c) $(01+11+10)^*$
- d) $(1+2+0)^*(1+2)^*$

Answer: b

Explanation: Except b all are regular expression*.

101. Which of the following do Regexp do not find their use in?

- a) search engines
- b) word processors
- c) sed
- d) none of the mentioned

Answer: d

Explanation: Regexp processors are found in several search engines, seach and replace mechanisms, and text processing utilities.

102. Which of the following languages have built in regexps support?

- a) Perl
- b) Java
- c) Python
- d) C++

Answer: a

Explanation: Many languages come with built in support of regexps like Perl, Javascript, Ruby etc. While some provide support using standard libraries like .NET, Java, Python, C++, C and POSIX.

103. The following is/are an approach to process a regexp:

- a) Contruction of NFA and subsequently, a DFA.
- b) Thompson's Contruction Algorithm
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: A regexp processor translates

Theory of Computation and Compilers Unit – 8 MCQs

the syntax into internal representation which can be executed and matched with a string and that internal representation can have several approaches like the ones mentioned.

104. Are the given two patterns equivalent?

(1) gray|grey

(2) gr(a|e)y

a) yes

b) no

Answer: a

Explanation: Paranthesis can be used to define the scope and precedence of operators. Thus, both the expression represents the same pattern.

105. Which of the following are not quantifiers?

a) Kleene plus +

b) Kleene star *

c) Question mark ?

d) None of the mentioned

Answer: d

Explanation: A quantifier after a token specifies how often the preceding element is allowed to occur. ?, *, +, {n}, {min, }, {min, max} are few quantifiers we use in regexprs implementations.

106. Which among the following is not a UNIX command for regular expressions?

a) ed

b) sed

c) vi

d) none of the mentioned

Answer: d

Explanation: Regular expressions are used by different commands in Unix like ed, sed, grep, awk, vi, etc. Sed stands for stream editor which is exclusively used for executing scripts.

107. Lexemes can be referred to as:

a) elements of lexicography

b) sequence of alphanumeric characters in a token

c) lexical errors

d) none of the mentioned

Answer: b

Explanation: A lexeme is a string of characters that form a syntactic unit. It is reasonable to say that is the sequence of alphanumeric characters in a token.

108. If the lexical analyser finds a lexeme with the same name as that of a reserved word, it _____

a) overwrites the word

b) overwrites the functionality

c) generates an error

d) something else

Answer: c

Explanation: Reserved words are known as keywords and they are specific and reserved with its functionality to a language. Thus, getting an input with the same name by the analyzer will generate an error.

109. The methodology to show an error when the analyzer faces a keyword over an user's input is based on:

a) rule priority

Theory of Computation and Compilers Unit – 8 MCQs

- b) longest match rule
- c) keyword-out rule
- d) none of mentioned

Answer: a

Explanation: The lexical analyzer follows the rule priority where it prioritizes keywords over an input it gets with the same name as that of the keyword and thus generates an error.

110. State true or false:

Statement: A lexical analyzer reads the source code line by line.

- a) True
- b) False

Answer: b

Explanation: A lexical analyzer reads the source code letter by letter and when it encounters a space or an operator or any special character, it decides that the word is completed.

111. Which among the following statement is correct?

Statement 1: When the analyzer scans 'int' and 'intvalue', it is not able to decide whether the int leads to a keyword or an identifier.

Statement 2: Longest Match Rule

- a) Statement 1 is assertion, Statement 2 is the reason
- b) Statement 1 is assertion, Statement 2 is the solution
- c) There is no such Statement 2
- d) This is not a function of Lexical Analyzer

Answer: b

Explanation: The Longest Match rule states that the lexeme scanned should be determined on the basis of longest match among all the token available.

112. The output of the lexical and syntax analyzer can be stated as:

- a) parse stream, parse tree
- b) token tree, parse tree
- c) token stream, parse tree
- d) all of the mentioned

Answer: c

Explanation: The lexical analyzer outputs the stream of token which is taken up by syntax analyzer one by one against the production rule and parse tree is generated.

113. Which among the following is not a tool to construct lexical analyzer from a regular expression?

- a) lex
- b) flex
- c) jflex
- d) none of the mentioned

Answer: d

Explanation: Lexical analysis is done using few tools such as lex, flex and jflex. Jflex is a computer program that generates lexical analyzers (also known as lexers or scanners) and works apparently like lex and flex. Lex is commonly used with yacc parser generator.

114. A program that performs lexical analysis is termed as:

- a) scanner
- b) lexer

Theory of Computation and Compilers Unit – 8 MCQs

- c) tokenizer
- d) all of the mentioned

Answer: d

Explanation: A program which performs lexical analysis is called lexer, scanner or lex. Nowadays, lexer is combined with a parser which allows syntactic analysis.

115. Which phase of compiler includes Lexical Analysis?

- a) 1
- b) 2
- c) 3
- d) Its primary function, not in any phase

Answer: a

Explanation: The first phase of compilation process is called lexical analysis. It fragments the source code into token which is the smallest programming unit of a program.

116. The minimum length of a string $\{0,1\}^*$ not in the language corresponding to the given regular expression:

$(0^*+1^*)(0^*+1^*)(0^*+1^*)$

- a) 3
- b) 4
- c) 5
- d) 6

Answer: b

Explanation: 0101 or 1010 the strings with minimum length on $\{0,1\}^*$ which does not belong to the language of the given regular expression. Other strings like 111, 000, 1101, etc are accepted by the language.

117. Which of the following regular expression is equivalent to $R(1,0)$?

$R(1,0)=\{111^*\}^*$

- a) $(11+111)^*$
- b) $(111+1111)^*$
- c) $(111+11^*)^*$
- d) All of the mentioned

Answer: a

Explanation: What we observe from the question is that, it includes e and 11 and any number of 1's then. Therefore, its simplifies when we write the same reg. Expression as $(11+111)^*$.

118. All the regular languages can have one or more of the following descriptions:

i) DFA ii) NFA iii) e-NFA iv) Regular Expressions

Which of the following are correct?

- a) i, ii, iv
- b) i, ii, iii
- c) i, iv
- d) i, ii, iii, iv

Answer: d

Explanation: The class of languages known as the regular language has atleast four different descriptions: i) DFA ii) NFA iii) e-NFA iv) Regular Expressions

119. Which of the technique can be used to prove that a language is non regular?

- a) Ardens theorem
- b) Pumping Lemma
- c) Ogden's Lemma
- d) None of the mentioned

Answer: b

Explanation: We use the powerful

Theory of Computation and Compilers Unit – 8 MCQs

technique called Pumping Lemma, for showing certain languages not to be regular. We use Ardens theorem to find out a regular expression out of a finite automaton.

120. Which of the following language regular?

- a) $\{a^i b^i \mid i \geq 0\}$
- b) $\{a^i b^i \mid 0 < i < 5\}$
- c) $\{a^i b^i \mid i \geq 1\}$
- d) None of the mentioned

Answer: b

Explanation: Here, i has limits i.e. the language is finite, contains few elements and can be graphed using a deterministic finite automata. Thus, it is regular. Others can be proved non regular using Pumping lemma.

121. Which of the following are non regular?

- a) The set of strings in $\{a, b\}^*$ with an even number of b 's
- b) The set of strings in $\{a, b, c\}^*$ where there is no c anywhere to the left of a
- c) The set of strings in $\{0, 1\}^*$ that encode, in binary, an integer w that is a multiple of 3. Interpret the empty strings ϵ as the number 0.
- d) None of the mentioned

Answer: d

Explanation: All of the given languages are regular and finite and thus, can be represented using respective deterministic finite automata. We can also use mealy or moore machine to represent remainders for option c.

122. If L is DFA-regular, L' is

- a) Non regular
- b) DFA-regular
- c) Non-finite
- d) None of the mentioned

Answer: b

Explanation: This is a simple example of a closure property: a property saying that the set of DFA-regular languages is closed under certain operations.

123. Which of the following options is incorrect?

- a) A language L is regular if and only if $\sim L$ has finite number of equivalent classes.
- b) Let L be a regular language. If $\sim L$ has k equivalent classes, then any DFA that recognizes L must have atmost k states.
- c) A language L is NFA-regular if and only if it is DFA-regular.
- d) None of the mentioned

Answer: b

Explanation: Let L be a regular language. If $\sim L$ has k equivalent classes, then any DFA that recognizes L must have atleast k states.

124. Myhill Nerode does the following:

- a) Minimization of DFA
- b) Tells us exactly when a language is regular
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: In automata theory, the Myhill Nerode theorem provides a necessary and sufficient condition for a language to be regular. The Myhill Nerode

Theory of Computation and Compilers Unit – 8 MCQs

theorem can be used to show a language L is regular by proving that the number of equivalence classes of R_L (relation) is finite.

125. Given languages:

i) $\{a^n b^n \mid n \geq 0\}$

ii) $\langle \text{div} \rangle^n \langle / \text{div} \rangle^n$

iii) $\{w \in \{a,b\}^* \mid \#a(w) = \#b(w)\}$, # represents occurrences

Which of the following is/are non regular?

a) i, iii

b) i

c) iii

d) i, ii, iii

Answer: d

Explanation: There is no regular expression that can parse HTML documents. Other options are also non-regular as they cannot be drawn into finite automaton.

126. Finite state machine are not able to recognize Palindromes because:

a) Finite automata cannot deterministically find the midpoint

b) Finite automata cannot remember arbitrarily large amount of data

c) Even if the mid point is known, it cannot find whether the second half matches the first

d) All of the mentioned

Answer: d

Explanation: It is the disadvantage or lack of property of a DFA that it cannot remember an arbitrarily such large amount of data which makes it incapable of accepting such languages like palindrome, reversal, etc.

127. Relate the following statement:

Statement: All sufficiently long words in a regular language can have a middle section of words repeated a number of times to produce a new word which also lies within the same language.

a) Turing Machine

b) Pumping Lemma

c) Arden's theorem

d) None of the mentioned

Answer: b

Explanation: Pumping lemma defines an essential property for every regular language in automata theory. It has certain rules which decide whether a language is regular or not.

128. While applying Pumping lemma over a language, we consider a string w that belong to L and fragment it into _____ parts.

a) 2

b) 5

c) 3

d) 6

Answer: c

Explanation: We select a string w such that $w = xyz$ and $|y| > 0$ and other conditions. However, there exists an integer n such that $|w| \geq n$ for any $w \in L$.

129. If we select a string w such that $w \in L$, and $w = xyz$. Which of the following portions cannot be an empty string?

a) x

b) y

c) z

d) all of the mentioned

Theory of Computation and Compilers Unit – 8 MCQs

Answer: b

Explanation: The lemma says, the portion y in xyz cannot be zero or empty i.e. $|y| > 0$, this condition needs to be fulfilled to check the conclusion condition.

130. Let $w = xyz$ and y refers to the middle portion and $|y| > 0$. What do we call the process of repeating y 0 or more times before checking that they still belong to the language L or not?

- a) Generating
- b) Pumping
- c) Producing
- d) None of the mentioned

Answer: b

Explanation: The process of repetition is called pumping and so, pumping is the process we perform before we check whether the pumped string belongs to L or not.

131. There exists a language L . We define a string w such that $w \in L$ and $w = xyz$ and $|w| \geq n$ for some constant integer n . What can be the maximum length of the substring xy i.e. $|xy| \leq ?$

- a) n
- b) $|y|$
- c) $|x|$
- d) none of the mentioned

Answer: a

Explanation: It is the first conditional statement of the lemma that states that $|xy| \leq n$, i.e. the maximum length of the substring xy in w can be n only.

132. Fill in the blank in terms of p , where p is the maximum string length in L .

Statement: Finite languages trivially satisfy the pumping lemma by having $n = \underline{\hspace{2cm}}$

- a) $p+1$
- b) $p-1$
- c) $p-1$
- d) None of the mentioned

Answer: b

Explanation: Finite languages trivially satisfy the pumping lemma by having n equal to the maximum string length in L plus 1.

133. Which kind of proof is used to prove the regularity of a language?

- a) Proof by contradiction
- b) Direct proof
- c) Proof by induction
- d) None of the mentioned

Answer: a

Explanation: We use the method of proof by contradiction in pumping lemma to prove that a language is regular or not.

134. The language of balanced parentheses is

- a) regular
- b) non regular
- c) may be regular
- d) none of the mentioned

Answer: b

Explanation: Given n , there is a string of balanced parentheses that begins with more than p left parentheses, so that y will contain entirely of left parentheses. By repeating y , we can produce a string that does not contain the same number of left

Theory of Computation and Compilers Unit – 8 MCQs

and right parentheses, and so they cannot be balanced.

135. State true or false:

Statement: Pumping lemma gives a necessary but not sufficient condition for a language to be regular.

- a) true
- b) false

Answer: a

Explanation: The converse of the lemma is not true. There may exist some language which satisfies all the conditions of the lemma and still be non-regular.

136. Which of the following is/are an example of pigeon hole principle?

- a) Softball team
- b) Sock picking
- c) Hair counting
- d) All of the mentioned

Answer: d

Explanation: There are several applications of pigeonhole principle:

Example: The softball team: Suppose 7 people who want to play softball ($n=7$ items), with a limitation of only 4 softball teams to choose from. The pigeonhole principle tells us that they cannot all play for different teams; there must be at least one team featuring at least two of the seven players.

137. Pigeonhole principle can be applied in the following computer science algorithms:

- a) hashing algorithm
- b) lossless compression algorithm

- c) both (a) and (b)
- d) none of the mentioned

Answer: c

Explanation: Collisions are inevitable in a hash table because the number of possible keys exceeds the number of indices in the array.

138. The entity which generates Language is termed as:

- a) Automata
- b) Tokens
- c) Grammar
- d) Data

Answer: c

Explanation: The entity which accepts a language is termed as Automata while the one which generates it is called Grammar. Tokens are the smallest individual unit of a program.

139. Production Rule: $aAb \rightarrow agb$ belongs to which of the following category?

- a) Regular Language
- b) Context free Language
- c) Context Sensitive Language
- d) Recursively Enumerable Language

Answer: c

Explanation: Context Sensitive Language or Type 1 or Linearly Bounded Non deterministic Language has the production rule where the production is context dependent i.e. $aAb \rightarrow agb$.

140. Which of the following statement is false?

- a) Context free language is the subset of

Theory of Computation and Compilers Unit – 8 MCQs

context sensitive language

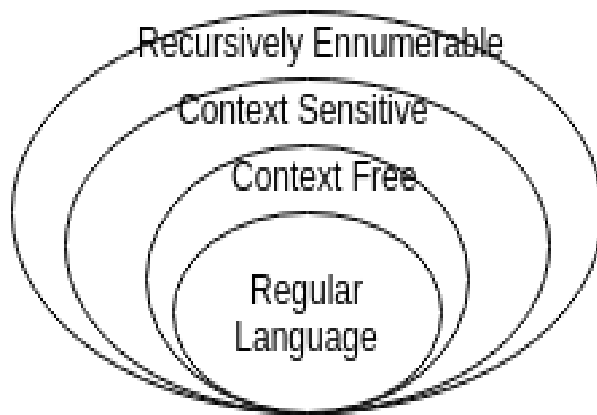
b) Regular language is the subset of context sensitive language

c) Recursively enumerable language is the super set of regular language

d) Context sensitive language is a subset of context free language

Answer: d

Explanation: Every regular language can be produced by context free grammar and context free language can be produced by context sensitive grammar and so on.



141. The Grammar can be defined as: $G=(V, \Sigma, p, S)$

In the given definition, what does S represents?

- a) Accepting State
- b) Starting Variable
- c) Sensitive Grammar
- d) None of these

Answer: b

Explanation: $G=(V, \Sigma, p, S)$, here V =Finite set of variables, Σ = set of terminals, p = finite productions, S = Starting Variable.

142. Which among the following cannot be accepted by a regular grammar?

a) L is a set of numbers divisible by 2

b) L is a set of binary complement

c) L is a set of string with odd number of 0

d) L is a set of 0^n1^n

Answer: d

Explanation: There exists no finite automata to accept the given language i.e. 0^n1^n . For other options, it is possible to make a dfa or nfa representing the language set.

143. Which of the expression is appropriate?

For production $p: a \rightarrow b$ where $a \in V$ and $b \in \underline{\hspace{2cm}}$

- a) V
- b) S
- c) $(V+\Sigma)^*$
- d) $V+\Sigma$

Answer: c

Explanation: According to the definition, the starting variable can produce another variable or any terminal or a variable which leads to terminal.

144. For $S \rightarrow OS1 \mid \epsilon$ for $\Sigma=\{0,1\}^*$, which of the following is wrong for the language produced?

- a) Non regular language
- b) $0^n1^n \mid n \geq 0$
- c) $0^n1^n \mid n \geq 1$
- d) None of the mentioned

Answer: d

Explanation: $L=\{\epsilon, 01, 0011, 000111, \dots, 0^n1^n\}$. As epsilon is a part of the set, thus all the options are correct implying none of them to be wrong.

Theory of Computation and Compilers Unit – 8 MCQs

145. The minimum number of productions required to produce a language consisting of palindrome strings over $\Sigma=\{a,b\}$ is

- a) 3
- b) 7
- c) 5
- d) 6

Answer: c

Explanation: The grammar which produces a palindrome set can be written as:

$S \rightarrow aSa \mid bSb \mid e \mid a \mid b$

$L=\{e, a, b, aba, abbbbaabbbba.....\}$

146. Which of the following statement is correct?

- a) All Regular grammar are context free but not vice versa
- b) All context free grammar are regular grammar but not vice versa
- c) Regular grammar and context free grammar are the same entity
- d) None of the mentioned

Answer: a

Explanation: Regular grammar is a subset of context free grammar and thus all regular grammars are context free.

147. Are ambiguous grammar context free?

- a) Yes
- b) No

Answer: a

Explanation: A context free grammar G is ambiguous if there is atleast one string in $L(G)$ which has two or more distinct leftmost derivations.

148. Which of the following is not a notion of Context free grammars?

- a) Recursive Inference
- b) Derivations
- c) Sentential forms
- d) All of the mentioned

Answer: d

Explanation: The following are the notions to express Context free grammars:

- a) Recursive Inferences
- b) Derivations
- c) Sentential form
- d) Parse trees

149. State true or false:

Statement: The recursive inference procedure determines that string w is in the language of the variable A, A being the starting variable.

- a) true
- b) false

Answer: a

Explanation: We apply the productions of CFG to infer that certain strings are in the language of a certain variable.

150. Which of the following is/are the suitable approaches for inferencing?

- a) Recursive Inference
- b) Derivations
- c) Both Recursive Inference and Derivations
- d) None of the mentioned

Answer: c

Explanation: Two inference approaches:

1. Recursive inference, using productions from body to head

Theory of Computation and Compilers Unit – 8 MCQs

2. Derivations, using productions from head to body

151. If w belongs to $L(G)$, for some CFG, then w has a parse tree, which defines the syntactic structure of w . w could be:

- a) program
- b) SQL-query
- c) XML document
- d) All of the mentioned

Answer: d

Explanation: Parse trees are an alternative representation to derivations and recursive inferences. There can be several parse trees for the same string.

152. Is the following statement correct?

Statement: Recursive inference and derivation are equivalent.

- a) Yes
- b) No

Answer: a

Explanation: Yes, they are equivalent. Both the terminologies represent the two approaches of recursive inferencing.

153. $A \rightarrow aA \mid a \mid b$

The number of steps to form aab :

- a) 2
- b) 3
- c) 4
- d) 5

Answer: b

Explanation: $A \rightarrow aA \Rightarrow aaA \Rightarrow aab$

154. An expression is mentioned as follows.

Figure out number of incorrect notations or symbols, such that a change in those could

make the expression correct.

$L(G) = \{w \text{ in } T^* \mid S \rightarrow^* w\}$

- a) 0 Errors
- b) 1 Error
- c) 2 Error
- d) Invalid Expression

Answer: a

Explanation: For the given expression, $L(G) = \{w \text{ in } T^* \mid S \rightarrow^* w\}$, If $G(V, T, P, S)$ is a CFG, the language of G , denoted by $L(G)$, is the set of terminal strings that have derivations from the start symbol.

155. The language accepted by Push down Automaton:

- a) Recursive Language
- b) Context free language
- c) Linearly Bounded language
- d) All of the mentioned

Answer: b

Explanation: Push down automata accepts context free language.

156. Which of the following the given language belongs to?

$L = \{a^m b^m c^m \mid m \geq 1\}$

- a) Context free language
- b) Regular language
- c) Both (a) and (b)
- d) None of the mentioned

Answer: d

Explanation: The given language is neither accepted by a finite automata or a push down automata. Thus, it is neither a context free language nor a regular language.

Theory of Computation and Compilers Unit – 8 MCQs

157. Choose the correct option:

Statement: There exists two inference approaches:

- a) Recursive Inference
- b) Derivation

- a) true
- b) partially true
- c) false
- d) none of the mentioned

Answer: a

Explanation: We apply the productions of a CFG to infer that certain strings are in a language of certain variable.

158. Choose the correct option:

Statement 1: Recursive Inference, using productions from head to body.

Statement 2: Derivations, using productions from body to head.

- a) Statement 1 is true and Statement 2 is true
- b) Statement 1 and Statement 2, both are false
- c) Statement 1 is true and Statement 2 is false
- d) Statement 2 is true and Statement 1 is true

Answer: b

Explanation: Both the statements are false. Recursive Inference, using productions from body to head. Derivations, using productions from head to body.

159. Which of the following statements are correct for a concept called inherent ambiguity in CFL?

- a) Every CFG for L is ambiguous

- b) Every CFG for L is unambiguous
- c) Every CFG is also regular
- d) None of the mentioned

Answer: a

Explanation: A CFL L is said to be inherently ambiguous if every CFG for L is ambiguous.

160. Which of the theorem defines the existence of Parikh's theorem?

- a) Parikh's theorem
- b) Jacobi theorem
- c) AF+BG theorem
- d) None of the mentioned

Answer: a

Explanation: Rohit Parikh in 1961 proved in his MIT research paper that some context free language can only have ambiguous grammars.

161. The most suitable data structure used to represent the derivations in compiler:

- a) Queue
- b) Linked List
- c) Tree
- d) Hash Tables

Answer: c

Explanation: The tree, known as "Parse tree" when used in a compiler, is the data structure of choice to represent the source program.

162. Which of the following statement is false in context of tree terminology?

- a) Root with no children is called a leaf
- b) A node can have three children
- c) Root has no parent

Theory of Computation and Compilers Unit – 8 MCQs

d) Trees are collection of nodes, with a parent child relationship

Answer: a

Explanation: A node has atmost one parent, drawn above the node, and zero or more children drawn below. Lines connect parents to children. There is one node, one root, that has no parent; this node appears to be at the top of the tree. Nodes with no children are called leaves. Nodes that are not leaves are called interior nodes.

163. In which order are the children of any node ordered?

- a) From the left
- b) From the right
- c) Arbitrarily
- d) None of the mentioned

Answer: a

Explanation: The children of a node are ordered from the left and drawn so. If N is to the left of node M, then all the descendents of N are considered to be to the left of all the descendents of M.

164. Which among the following is the root of the parse tree?

- a) Production P
- b) Terminal T
- c) Variable V
- d) Starting Variable S

Answer: d

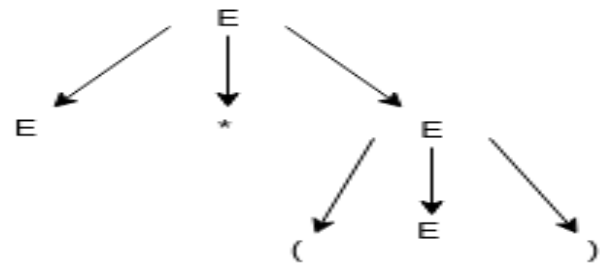
Explanation: The root is labelled by the start symbol. All the leaves are either labelled by a terminal or with e.

165. For the expression $E^*(E)$ where * and brackets are the operation, number of nodes in the respective parse tree are:

- a) 6
- b) 7
- c) 5
- d) 2

Answer: b

Explanation:

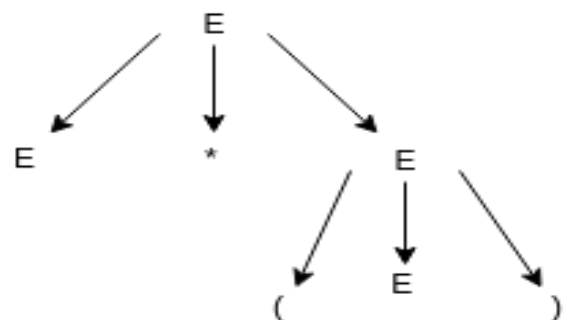


166. The number of leaves in a parse tree with expression $E^*(E)$ where * and () are operators

- a) 5
- b) 2
- c) 4
- d) 3

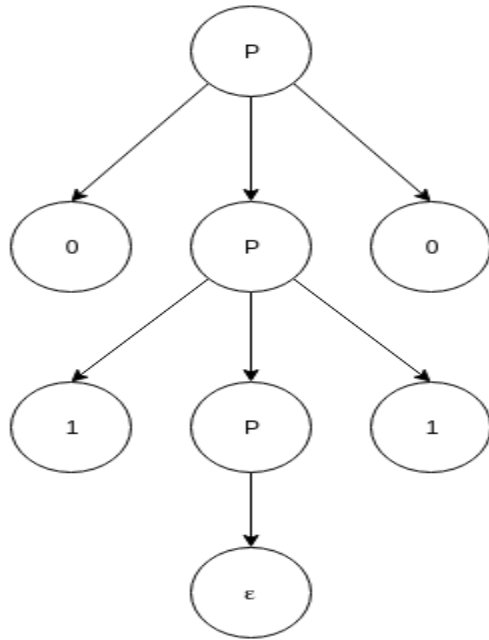
Answer: a

Explanation:



167. Which of the following does the given parse tree correspond to?

Theory of Computation and Compilers Unit – 8 MCQs



- a) $P \rightarrow 1100$
- b) $P \rightarrow 0110$
- c) $P \rightarrow 1100\epsilon$
- d) $P \rightarrow 0101$

Answer: b

Explanation: The following is a parse tree for the production 0110 over $\{0,1\}^*$.

168. A grammar with more than one parse tree is called:

- a) Unambiguous
- b) Ambiguous
- c) Regular
- d) None of the mentioned

Answer: b

Explanation: A context free grammar G is ambiguous if there is at least one string in $L(G)$ having two or more distinct derivation trees or equivalently, two or more distinct leftmost derivations.

169. _____ is the acyclic graphical representation of a grammar.

- a) Binary tree
- b) Oct tree
- c) Parse tree
- d) None of the mentioned

Answer: c

Explanation: In order to graphically represent a derivation of a grammar we need to use parse trees.

170. Grammar is checked by which component of compiler

- a) Scanner
- b) Parser
- c) Semantic Analyzer
- d) None of the mentioned

Answer: Parser or syntax analyzer is the one responsible for checking the grammar and reporting errors. In this phase, parse tree is generated and syntax is analyzed.

171. A symbol X is _____ if there exists : $S \rightarrow^* aXb$

- a) reachable
- b) generating
- c) context free
- d) none of the mentioned

Answer: a

Explanation: A symbol X is generating if there exists : $X \rightarrow^* w$ for some w that belongs to T^* .

Also, a symbol can never be context free.

172. A symbol X is called to be useful if and only if its is:

- a) generating
- b) reachable

Theory of Computation and Compilers Unit – 8 MCQs

- c) both generating and reachable
- d) none of the mentioned

Answer: c

Explanation: For a symbol X to be useful, it has to be both reachable and generating i.e. $S \rightarrow^* aXb \rightarrow^* w$ where w belongs to T^* .

173. Which of the following is false for a grammar G in Chomsky Normal Form:

- a) G has no useless symbols
- b) G has no unit productions
- c) G has no epsilon productions
- d) None of the mentioned

Answer: d

Explanation: G, a CFG is said to be in Chomsky normal form if all its productions are in one of the following form:

$A \rightarrow BC$ or $A \rightarrow a$

174. Given Checklist:

- a) G has no useless symbols
 - b) G has no unit productions
 - c) G has no epsilon productions
 - d) Normal form for production is violated
- Is it possible for the grammar G to be in CNF with the following checklisty ?
- a) Yes
 - b) No

Answer: b

Explanation: The grammar is not in CNF if it violates the normal form of the productions which is strictly restricted.

175. State true or false:

Statement: A CNF parse tree's string yield (w) can no longer be 2^{h-1} .

- a) true
- b) false

Answer: a

Explanation: It is the parse tree theorem which states:

Given: Suppose we have a parse tree for a string w, according to a CNF grammar, $G=(V, T, P, S)$. Let h be the height of the parse tree. Now, Implication: $|w| \leq 2^{h-1}$.

176. If $|w| \geq 2^h$, then its parse tree's height is at least _____

- a) h
- b) h+1
- c) h-1
- d) 2^h

Answer: b

Explanation: It is the basic implication of Parse tree theorem (assuming CNF). If the height of the parse tree is h, then $|w| \leq 2^{h-1}$.

177. If w belongs to $L(G)$, for some CFG, then w has a parse tree, which tell us the _____ structure of w.

- a) semantic
- b) syntactic
- c) lexical
- d) all of the mentioned

Answer: b

Explanation: A parse tree or concrete syntactic tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context free grammar.

Theory of Computation and Compilers Unit – 8 MCQs

178. Which of the following are distinct to parse trees?

- a) abstract parse trees
- b) sentence diagrams
- c) both abstract parse trees and sentence diagrams
- d) none of the mentioned

Answer: c

Explanation: Both of the mentioned are different from parse trees. Sentence diagrams are pictorial representations of grammatical structure of a sentence.

179. Choose the correct option:

Statement: Unambiguity is the ideal structure of a language.

- a) true
- b) partially true
- c) false
- d) cant be said

Answer: a

Explanation: Ideally, there should be only one parse tree for each string, i.e. the language should be unambiguous.

180. Is the given statement correct?

Statement: The mere existence of several derivations is not an issue, its is the existence of several parse trees that ruins a grammar.

- a) Yes
- b) No

Answer: a

Explanation: It is also true that multiple leftmost or rightmost derivations do cause ambiguity. Unfortunately, it is not possible to remove the ambiguity always.

181. To derive a string using the production rules of a given grammar, we use:

- a) Scanning
- b) Parsing
- c) Derivation
- d) All of the mentioned

Answer: b

Explanation: Parsing is required to check the acceptability of a string. Further, comes the syntactical phase which is taken care by other phases of compiler.

182. Which of the following parser reaches the root symbol of the tree at last?

- a) Top down parser
- b) Bottom up parser
- c) TOP down and Bottom up parser
- d) None of the mentioned

Answer: b

Explanation: Bottom up parser starts from the bottom with the string and comes up to the start symbol using a parse tree or a derivation tree.

183. Left corner parsing method uses which of the following?

- a) Top down parser
- b) Bottom up parser
- c) TOP down and Bottom up parser
- d) None of the mentioned

Answer: c

Explanation: It is a hybrid method which works bottom up along the left edges of each subtree, and top down on the rest of the parse tree.

Theory of Computation and Compilers Unit – 8 MCQs

184. Which of the following parser performs top down parsing?

- a) LALR parser
- b) LL parser
- c) Recursive Accent parser
- d) None of the mentioned

Answer: b

Explanation: Bottom up parsing is done by shift reduce parsers like LALR parsers, Operator precedence parsers, simple precedence parsers, etc.

185. Which of the following is true for shift reduce parsers?

- a) Scans and parses the input in one forward pass over the text, without any backup.
- b) A shift command advances in the input stream by one symbol
- c) LALR parser
- d) All of the mentioned

Answer: d

Explanation: The mentioned are the correct and proper functions of a shift reduce parsers. The parsing methods are most commonly used for parsing programming languages, etc.

186. State true or false:

Statement: LALR parsers uses tables rather than mutually recursive functions.

- a) true
- b) false

Answer: b

Explanation: It is exactly the opposite case where LALR parsers uses mutually recursive

functions instead of tables. It is a simplified version of canonical left to right parser.

187. LALR in LALR parser stands for:

- a) Left aligned left right parser
- b) Look ahead left to right parser
- c) Language Argument left to right parser
- d) None of the mentioned

Answer:

Explanation: LALR stands for Look ahead left to right parsers. It has more language recognition power than LR(0) parser.

188. Which of the following can be a LALR parser generator?

- a) YACC
- b) GNU Bison
- c) YACC and GNU Bison
- d) None of the mentioned

Answer: c

Explanation: YACC is a computer code for UNIX operating system which generates a LALR parser. On the other hand GNU Bison or Bison can generate LALR and GLR parsers.

189. Which of the following parsers do not relate to Bottom up parsing?

- a) LL parser
- b) Recursive descent parser
- c) Earley parsers
- d) All of the mentioned

Answer: d

Explanation: All the following mentioned are top down parsers and begin their operation from the starting symbol.

Theory of Computation and Compilers Unit – 8 MCQs

190. Which of the following is true for a predictive parser?

- a) Recursive Descent parser
- b) no backtracking
- c) Recursive Descent parser and no backtracking
- d) None of the mentioned

Answer: c

Explanation: Predictive parsing is possible only for the class of LL-grammars, which are the CFG for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input.

191. A push down automaton employs _____ data structure.

- a) Queue
- b) Linked List
- c) Hash Table
- d) Stack

Answer: d

Explanation: A push down automata uses a stack to carry out its operations. They are more capable than the finite automata but less than the turing model.

193. State true or false:

Statement: The operations of PDA never work on elements, other than the top.

- a) true
- b) false

Answer: a

Explanation: The term pushdown refers to the fact that the elements are pushed down in the stack and as per the LIFO principle,

the operation is always performed on the top element of the stack.

194. Which of the following allows stacked values to be sub-stacks rather than just finite symbols?

- a) Push Down Automaton
- b) Turing Machine
- c) Nested Stack Automaton
- d) None of the mentioned

Answer: c

Explanation: In computational theory, a nested stack automaton is a finite automaton which makes use of stack containing data which can be additional stacks.

195. A non deterministic two way, nested stack automaton has n -tuple definition. State the value of n .

- a) 5
- b) 8
- c) 4
- d) 10

Answer: d

Explanation: The 10-tuple can be stated as: $NSA = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F, [], \rangle$.

196. Push down automata accepts _____ languages.

- a) Type 3
- b) Type 2
- c) Type 1
- d) Type 0

Answer: b

Explanation: Push down automata is for

Theory of Computation and Compilers Unit – 8 MCQs

Context free languages and they are termed as Type 2 languages according to Chomsky hierarchy.

197. The class of languages not accepted by non deterministic, nonerasing stack automata is _____

- a) NSPACE(n^2)
- b) NL
- c) CSL
- d) All of the mentioned

Answer: d

Explanation: NSPACE or non deterministic space is the computational resource describing the memory space for a non deterministic turing machine.

198. A push down automaton with only symbol allowed on the stack along with fixed symbol.

- a) Embedded PDA
- b) Nested Stack automata
- c) DPDA
- d) Counter Automaton

Answer: d

Explanation: This class of automata can recognize a set of context free languages like $\{a^n b^n \mid n \text{ belongs to } N\}$

199. Which of the operations are eligible in PDA?

- a) Push
- b) Delete
- c) Insert
- d) Pop

Answer: a, d

Explanation: Push and pop are the operations we perform to operate a stack. A stack follows the LIFO principle, which states its rule as: Last In First Out.

200. A string is accepted by a PDA when

- a) Stack is empty
- b) Acceptance state
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: When we reach the acceptance state and find the stack to be empty, we say, the string has been accepted by the push down automata.

201. If two sets, R and T has no elements in common i.e. $R \cap T = \emptyset$, then the sets are called

- a) Complement
- b) Union
- c) Disjoint
- d) Connected

Answer: c

Explanation: Two sets are called disjoint if they have no elements in common i.e. $R \cap T = \emptyset$.

202. Which among the following is not a part of the Context free grammar tuple?

- a) End symbol
- b) Start symbol
- c) Variable
- d) Production

Answer: a

Explanation: The tuple definition of context

Theory of Computation and Compilers Unit – 8 MCQs

free grammar is: (V, T, P, S) where V =set of variables, T =set of terminals, P =production, S = Starting Variable.

203. A context free grammar is a

- a) English grammar
- b) Regular grammar
- c) Context sensitive grammar
- d) None of the mentioned

Answer: c

Explanation: Context free grammar is the set which belongs to the set of context free grammar. Similarly, Regular grammar is a set which belongs to the the set of Context free grammar.

204. The closure property of context free grammar includes :

- a) Kleene
- b) Concatenation
- c) Union
- d) All of the mentioned

Answer: d

Explanation: Context free grammars are closed under kleene operation, union and concatenation too.

205. Which of the following automata takes stack as auxiliary storage?

- a) Finite automata
- b) Push down automata
- c) Turing machine
- d) All of the mentioned

Answer: b

Explanation: Pushdown Automaton uses stack as an auxiliary storage for its

operations. Turing machines use Queue for the same.

206. Which of the following automata takes queue as an auxiliary storage?

- a) Finite automata
- b) Push down automata
- c) Turing machine
- d) All of the mentioned

Answer: c

Explanation: Pushdown Automaton uses stack as an auxiliary storage for its operations. Turing machines use Queue for the same.

207. A context free grammar can be recognized by

- a) Push down automata
- b) 2 way linearly bounded automata
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: A linearly bounded automata is a restricted non deterministic turing machine which is capable of accepting any context free grammar.

208. A null production can be referred to as:

- a) String
- b) Symbol
- c) Word
- d) All of the mentioned

Answer: a

Explanation: Null production is always taken as a string in computational theory.

209. NPDA stands for

- a) Non-Deterministic Push Down Automata

Theory of Computation and Compilers Unit – 8 MCQs

- b) Null-Push Down Automata
- c) Nested Push Down Automata
- d) All of the mentioned

Answer: a

Explanation: NPDA stands for non-deterministic push down automata whereas DPDA stands for deterministic push down automata.

210. Suppose $A \rightarrow xBz$ and $B \rightarrow y$, then the simplified grammar would be:

- a) $A \rightarrow xyz$
- b) $A \rightarrow xBz \mid xyz$
- c) $A \rightarrow xBz \mid B \mid y$
- d) none of the mentioned

Answer: a

Explanation: For the first step, substitute B in first production as it only produces terminal and remove B production as it has already been utilized.

We get $A \rightarrow xBz \mid xyz$ and now, as B has no production, we eliminate the terms which hold the variable B, thus the answer remain $A \rightarrow xyz$.

211. Given Grammar: $S \rightarrow A$, $A \rightarrow aA$, $A \rightarrow e$, $B \rightarrow bA$

Which among the following productions are Useless productions?

- a) $S \rightarrow A$
- b) $A \rightarrow aA$
- c) $A \rightarrow e$
- d) $B \rightarrow bA$

Answer: d

Explanation: Some derivations are not reachable from the starting variable. As B is not reachable from the starting variable, it

is a useless symbol and thus, can be eliminated.

212. The format: $A \rightarrow aB$ refers to which of the following?

- a) Chomsky Normal Form
- b) Greibach Normal Form
- c) Backus Naur Form
- d) None of the mentioned

Answer: b

Explanation: A context free grammar is in Greibach Normal Form if the right hand sides of all the production rules start with a terminal, optionally followed by some variables.

213. Which of the following does not have left recursions?

- a) Chomsky Normal Form
- b) Greibach Normal Form
- c) Backus Naur Form
- d) All of the mentioned

Answer: b

Explanation: The normal form is of the format:

$A \rightarrow aB$ where the right hand side production tends to begin with a terminal symbol, thus having no left recursions.

214. Every grammar in Chomsky Normal Form is:

- a) regular
- b) context sensitive
- c) context free
- d) all of the mentioned

Answer: c

Explanation: Conversely, every context free

Theory of Computation and Compilers Unit – 8 MCQs

grammar can be converted into Chomsky Normal form and to other forms.

215. Which of the production rule can be accepted by Chomsky grammar?

- a) $A \rightarrow BC$
- b) $A \rightarrow a$
- c) $S \rightarrow e$
- d) All of the mentioned

Answer: d

Explanation: in CNF, the production rules are of the form:

$A \rightarrow BC$

$A \rightarrow a$

$S \rightarrow e$

216. Given grammar G:

- (1) $S \rightarrow AS$
- (2) $S \rightarrow AAS$
- (3) $A \rightarrow SA$
- (4) $A \rightarrow aa$

Which of the following productions denies the format of Chomsky Normal Form?

- a) 2,4
- b) 1,3
- c) 1, 2, 3, 4
- d) 2, 3, 4

Answer: a

Explanation: The correct format: $A \rightarrow BC$, $A \rightarrow a$, $X \rightarrow e$.

217. Which of the following grammars are in Chomsky Normal Form:

- a) $S \rightarrow AB \mid BC \mid CD$, $A \rightarrow 0$, $B \rightarrow 1$, $C \rightarrow 2$, $D \rightarrow 3$
- b) $S \rightarrow AB$, $S \rightarrow BCA \mid 0 \mid 1 \mid 2 \mid 3$
- c) $S \rightarrow ABa$, $A \rightarrow aab$, $B \rightarrow Ac$
- d) All of the mentioned

Answer: a

Explanation: We can eliminate the options on the basis of the format we are aware of: $A \rightarrow BC$, $B \rightarrow b$ and so on.

218. Let G be a grammar. When the production in G satisfy certain restrictions, then G is said to be in _____.

- a) restricted form
- b) parsed form
- c) normal form
- d) all of the mentioned

Answer: c

Explanation: When the production in G satisfy certain restrictions, then G is said to be in 'normal form'.

219. Let G be a grammar: $S \rightarrow AB \mid e$, $A \rightarrow a$, $B \rightarrow b$

Is the given grammar in CNF?

- a) Yes
- b) No

Answer: a

Explanation: e is allowed in CNF only if the starting variable does not occur on the right hand side of the derivation.

220. Which of the following is called Bar-Hillel lemma?

- a) Pumping lemma for regular language
- b) Pumping lemma for context free languages
- c) Pumping lemma for context sensitive languages
- d) None of the mentioned

Answer: b

Explanation: In automata theory, the

Theory of Computation and Compilers Unit – 8 MCQs

pumping lemma for context free languages, also known as the Bar-Hillel lemma, represents a property of all context free languages.

221. Which of the expressions correctly is an requirement of the pumping lemma for the context free languages?

- a) uv^nw^nxy
- b) $uv^nw^nx^nxy$
- c) $uv^{2n}wx^{2n}y$
- d) All of the mentioned

Answer: b

Explanation: Let L be a CFL. Then there is an integer n so that for any u that belong to language L satisfying $|t| \geq n$, there are strings u, v, w, x, y and z satisfying

$t=uvwxy$

$|vx| > 0$

$|vwx| \leq n$ For any $m \geq 0$, $uv^mw^nxy \in L$

222. Let L be a CFL. Then there is an integer n so that for any u that belong to language L satisfying

$|t| \geq n$, there are strings u, v, w, x, y and z satisfying

$t=uvwxy$.

Let p be the number of variables in CNF form of the context free grammar. The value of n in terms of p :

- a) 2p
- b) $2p$
- c) $2p+1$
- d) p^2

Answer: c

Explanation: This inequation has been derived from derivation tree for t which must have height at least $p+2$ (It has more

than 2^p leaf nodes, and therefore its height is $> p+1$).

223. Which of the following gives a positive result to the pumping lemma restrictions and requirements?

- a) $\{a^ib^jc^i \mid i \geq 0\}$
- b) $\{0^i1^i \mid i \geq 0\}$
- c) $\{ss \mid s \in \{a,b\}^*\}$
- d) None of the mentioned

Answer: b

Explanation: A positive result to the pumping lemma shows that the language is a CFL and its contradiction or negative result shows that the given language is not a Context Free language.

224. Using pumping lemma, which of the following cannot be proved as 'not a CFL'?

- a) $\{a^ib^jc^i \mid i \geq 0\}$
- b) $\{ss \mid s \in \{a,b\}^*\}$
- c) The set legal C programs
- d) None of the mentioned

Answer: d

Explanation: There are few rules in C that are context dependent. For example, declaration of a variable before it can be used.

225. The context free languages are closed under:

- a) Intersection
- b) Complement
- c) Kleene
- d) None of the mentioned

Answer: c

Explanation: Context free languages are

Theory of Computation and Compilers Unit – 8 MCQs

closed under the following operation:
union, kleene and concatenation. For
regular languages, we can add intersection
and complement to the list.

226. Given Grammar G1:

$S \rightarrow aSb$

$S \rightarrow e$

Grammar G2:

$R \rightarrow cRd$

$R \rightarrow e$

If $L(G) = L(G1) \cup L(G2)$, the number of
productions the new starting variable would
have:

- a) 2
- b) 3
- c) 4
- d) 1

Answer: a

Explanation:

$T \rightarrow S \mid R$

$S \rightarrow aSb$

$S \rightarrow e$

$R \rightarrow cRd$

$R \rightarrow e$

227. Context free languages are not closed
under:

- a) Intersection
- b) Intersection with Regular Language
- c) Complement
- d) All of the mentioned

Answer: d

Explanation: It is a theorem which states
that, Context free languages are not closed
under operations like intersection and
complement.

228. Which of the following is incorrect?

There exists algorithms to decide if:

- a) String w is in CFL L
- b) CFL L is empty
- c) CFL L is infinite
- d) All of the mentioned

Answer: d

Explanation: These properties are termed as
decision properties of a CFL and include a
set of problems like infiniteness problem,
emptiness problem and membership
problem.

229. If the start symbol is one of those
symbols which produce no terminal through
any sequence, the CFL is said to be

- a) nullable
- b) empty
- c) eliminated
- d) none of the mentioned

Answer: b

Explanation: In the process of removing
useless symbols, if the starting symbol is
also a part, the CFL can be then termed as
empty; otherwise not.

230. Using the pumping constant n , If there
is a string in the language of length
between _____ and _____ then the language
is infinite else not.

- a) $n, 2n-1$
- b) $2n, n$
- c) $n+1, 3n+6$
- d) $0, n+1$

Answer: a

Explanation: If there is a string in the
language of length between n and $2n-1$

Theory of Computation and Compilers Unit – 8 MCQs

then the language is infinite else not. The idea is essentially the same for regular languages.

231. Which of the following is/are CFL not closed under?

- a) Reverse
- b) Homomorphism
- c) Inverse Homomorphism
- d) All of the mentioned

Answer: d

Explanation: CFL is closed under union, Kleene and concatenation along with the properties reversal, homomorphism and inverse homomorphism but not difference and intersection.

232. If L_1 and L_2 are context free languages, $L_1 - L_2$ are context free:

- a) always
- b) sometimes
- c) never
- d) none of the mentioned

Answer: c

Explanation: Context free languages are not closed under difference, intersection and complement operations.

233. A _____ is context free grammar with at most one non terminal in the right hand side of the production.

- a) linear grammar
- b) linear bounded grammar
- c) regular grammar
- d) none of the mentioned

Answer: a

Explanation: A simple linear grammar is G

with $N = \{S\}$, $\Sigma = \{a, b\}$, P with start symbol S and rules

$S \rightarrow aSb$

$S \rightarrow \epsilon$

234. There is a linear grammar that generates a context free grammar

- a) always
- b) never
- c) sometimes
- d) none of the mentioned

Answer: c

Explanation: Linear grammar is a subset of context free grammar which has at most one non terminal symbol in the right hand side of the production. Thus, there exists some languages which are generated by Linear grammars.

235. The following format of grammatical notation is accepted by which of the following:

$AB \rightarrow CD$

$A \rightarrow BC$ or

$A \rightarrow B$ or

$A \rightarrow a$

where A, B, C, D are non terminal symbols and a is a terminal symbol.

- a) Greibach Normal Form
- b) Chomsky Normal Form
- c) Kuroda Normal Form
- d) None of the mentioned

Answer: c

Explanation: Linearly Bounded grammar or Kuroda Normal Form allows the following format of grammatical analysis:

$AB \rightarrow CD$

$A \rightarrow BC$ or

Theory of Computation and Compilers Unit – 8 MCQs

A→B or

A→a

236. Every Kuroda Normal form grammar generates _____

- a) Context free grammar
- b) Context sensitive grammar
- c) Unrestricted grammar
- d) None of the mentioned

Answer: b

Explanation: Every context sensitive grammar which does not produce an empty string can be generated by a grammar in Kuroda Normal form.

237. Which of the following can generate Unrestricted grammars?

- a) Pentonnen Normal form
- b) Floyd Normal form
- c) Greibach Normal form
- d) None of the mentioned

Answer: a

Explanation: Pentonnen Normal form (for Unrestricted grammars) is a special case where there is a slight modification in the format of Kuroda Normal form.

AB→AD

A→BC

A→a

238. Given a grammar in GNF and a derivable string in the grammar with the length n, any _____ will halt at depth n.

- a) top-down parser
- b) bottom-up parser
- c) multitape turing machine
- d) none of the mentioned

Answer: a

Explanation: Given a grammar in GNF and a derivable string in the grammar with the length n, any top-down parser will halt at depth n. As the parameter 'depth' is mentioned, we will use a top-down parser. Example-LL parser.

239. Which of the following grammars is similar to Floyd Normal form?

- a) Backus Naur Form
- b) Kuroda Normal Form
- c) Greibach Normal Form
- d) Chomsky Normal Form

Answer: a

Explanation: Donald Knuth implied a BNF" syntax in which all definitions have such a form may be said to be in "Floyd Normal Form".

A→B|C

A→BC

A→a

240. Which among the following can parse a context free grammar?

- a) top down parser
- b) bottom up parser
- c) CYK algorithm
- d) all of the mentioned

Answer: d

Explanation: We use certain algorithms to parse a context free grammar which include the most popular CYK algorithm which employs the concept of bottom up parsing and dynamic parsing.

241. The standard version of CYK algorithm operates only on context free grammars in

Theory of Computation and Compilers Unit – 8 MCQs

the following form:

- a) Greibach Normal form
- b) Chomsky Normal form
- c) Backus Naur form
- d) All of the mentioned

Answer: b

Explanation: It requires the presence of a context free grammar into Chomsky Normal form to operate. However, every context free grammar can be converted into CNF for keeping the sense of grammar equivalent.

242. Which of the following is not a negative property of Context free languages?

- a) Intersection
- b) Complement
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: Context free languages are not closed under complement and intersection. Thus, are called Negative properties.

243. The intersection of context free language and regular language is

- a) regular language
- b) context free language
- c) context sensitive language
- d) non of the mentioned

Answer: b

Explanation: If a language L_1 is regular and L_2 is a context free language, then L_1 intersection L_2 will result into a context free language.

244. Which of the following is regular?

- a) $a^{100}b^{100}$
- b) $(a+b)^* - \{a^{100}b^{100}\}$
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: As the language seems to be finite, a dfa can be constructed for the same, thus is regular.

245. Which of the following is not context free?

- a) $\{w: nA=nB=nC\}$
- b) $\{a^*b^*c^*\}$
- c) $\{a^{100}b^{100}\}$
- d) All of the mentioned

Answer: d

Explanation: $\{a^*b^*c^*\}$ and (c) are regular languages while option (a) is not context free language.

246. Which of the following can be used to prove a language is not context free?

- a) Ardens theorem
- b) Power Construction method
- c) Regular Closure
- d) None of the mentioned

Answer: c

Explanation: We can use the properties of regular closure to prove that a language is not a context free language. Example: Intersection of context free language and regular language is a context free language. Proof by contradiction helps here.

247. A turing machine is a

- a) real machine

Theory of Computation and Compilers Unit – 8 MCQs

- b) abstract machine
- c) hypothetical machine
- d) more than one option is correct

Answer: d

Explanation: A turing machine is abstract or hypothetical machine thought by mathematician Alan Turing in 1936 capable of simulating any algorithm, however complicated it is.

248. A turing machine operates over:

- a) finite memory tape
- b) infinite memory tape
- c) depends on the algorithm
- d) none of the mentioned

Answer: b

Explanation: The turing machine operates on an infinite memory tape divided into cells. The machine positions its head over the cell and reads the symbol.

249. Which of the functions are not performed by the turing machine after reading a symbol?

- a) writes the symbol
- b) moves the tape one cell left/right
- c) proceeds with next instruction or halts
- d) none of the mentioned

Answer: d

Explanation: After the read head reads the symbol from the input tape, it performs the following functions:

- a) writes a symbol(some model allow symbol erasure/no writing)
- b) moves the tape left or right (some models allows no motion)
- c) proceeds with subsequent instruction or

goes either into accepting halting state or rejecting halting state.

250. 'a' in a-machine is :

- a) Alan
- b) arbitrary
- c) automatic
- d) None of the mentioned

Answer: c

Explanation: The turing machine was invented by Alan Turing in 1936. He named it as a-machine(automatic machine).

251. Which of the problems were not answered when the turing machine was invented?

- a) Does a machine exists that can determine whether any arbitrary machine on its tape is circular.
- b) Does a machine exists that can determine whether any arbitrary machine on its tape is ever prints a symbol
- c) Hilbert Entscheidungs problem
- d) None of the mentioned

Answer: d

Explanation: Invention of turing machine answered a lot of questions which included problems like decision problem, etc.) . Alan was able to prove the properties of computation using such model.

252. A turing machine that is able to simulate other turing machines:

- a) Nested Turing machines
- b) Universal Turing machine
- c) Counter machine
- d) None of the mentioned

Theory of Computation and Compilers Unit – 8 MCQs

Answer: b

Explanation: A more mathematically oriented definition with the same universal nature was introduced by church and turing together called the Church-Turing thesis(formal theory of computation).

253. Which of the problems are unsolvable?

- a) Halting problem
- b) Boolean Satisfiability problem
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: Alan turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

254. Which of the following a turing machine does not consist of?

- a) input tape
- b) head
- c) state register
- d) none of the mentioned

Answer: d

Explanation: A state register is one which stores the state of the turing machine, one of the finitely many. Among these is the special start state with which the state register is initialized.

255. The value of n if turing machine is defined using n-tuples:

- a) 6
- b) 7
- c) 8
- d) 5

Answer: b

Explanation:

The 7-tuple definition of turing machine: (Q, S, G, d, q₀, B, F)

where Q= The finite set of states of finite control

S= The finite set of input symbols

G= The complete set of tape symbols

d= The transition function

q₀= The start state, a member of Q, in which the finite control is found initially.

B= The blank symbol

F= The set of final or accepting states, a subset of Q.

256. If d is not defined on the current state and the current tape symbol, then the machine _____

- a) does not halts
- b) halts
- c) goes into loop forever
- d) none of the mentioned

Answer: b

Explanation: If we reach h_A or h_R, we say TM halts. Once it has halted, it cannot move further, since d is not defined at any pair (h_A,X) or (h_R,X) where h_A = accept halting state and h_R = reject halting state.

257. Which of the following is not true about RASP?

- a) Binary search can be performed more quickly using RASP than a turing machine
- b) Stores its program in memory external to its state machines instructions
- c) Has infinite number of distinguishable, unbounded registers
- d) Binary search can be performed less

Theory of Computation and Compilers Unit – 8 MCQs

quickly using RASP than a turing machine
e) More than two options are incorrect

Answer: d

Explanation: In theoretical computer science, the random access stored program(RASP) machine model is an abstract machine used for the purpose of algorithm development and algorithm complexity theory.

258. The class of recursively enumerable language is known as:

- a) Turing Class
- b) Recursive Languages
- c) Universal Languages
- d) RE

Answer: d

Explanation: RE or recursively enumerable is only called the class of recursively enumerable language.

259. A language L is said to be Turing decidable if:

- a) recursive
- b) TM recognizes L
- c) TM accepts L
- d) None of the mentioned

Answer: a,b

Explanation: A language L is recursively enumerable if there is a turing machine that accepts L, and recursive if there is a TM that recognizes L.(Sometimes these languages are also called Turing-acceptable and Turing-decidable respectively).

260. Which of the following statements are false?

- a) Every recursive language is recursively enumerable
- b) Recursively enumerable language may not be recursive
- c) Recursive languages may not be recursively enumerable
- d) None of the mentioned

Answer: c

Explanation: Every recursive language is recursively enumerable but there exists recursively enumerable languages that are not recursive. If L is accepted by a Non deterministic TM T, and every possible sequence of moves of T causes it to halt, then L is recursive.

261. Choose the correct option:

Statement: If L1 and L2 are recursively enumerable languages over S, then the following is/are recursively enumerable.

- a) $L1 \cup L2$
- b) $L2 \cap L2$
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: Both the union and intersection operations preserve the property of recursive enumerability(Theorem).

262. If L is a recursive language, L' is:

- a) Recursive
- b) Recursively Enumerable
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: If T is a turing machine

Theory of Computation and Compilers Unit – 8 MCQs

recognizing L , we can make it recognize L' by interchanging the two outputs. And every recursive language is recursively enumerable.

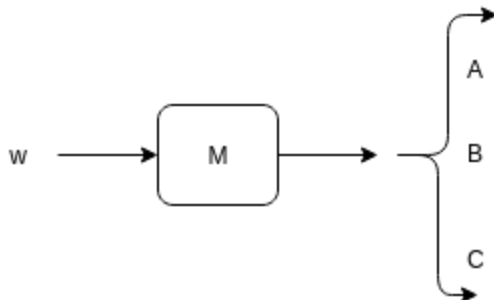
263. Choose the appropriate option:
Statement: If a language L is recursive, it is closed under the following operations:

- a) Union
- b) Intersection
- c) Complement
- d) All of the mentioned

Answer: d

Explanation: The closure property of recursive languages include union, intersection and complement operations.

264. A language L is recursively enumerable if $L=L(M)$ for some Turing machine M .

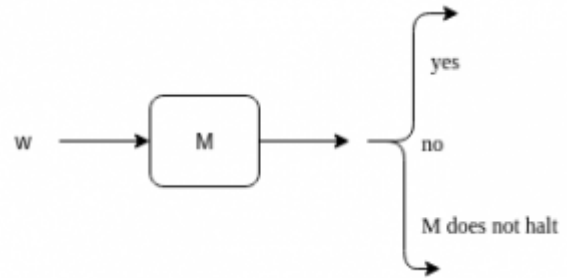


Which among the following cannot be among A, B and C?

- a) yes $w \in L$
- b) no $w \notin L$
- c) M does not halt $w \notin L$
- d) None of the mentioned

Answer: d

Explanation:



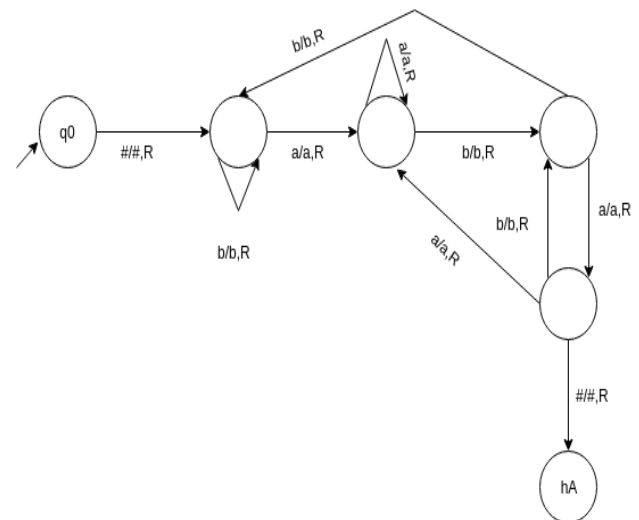
265. A Language L may not be accepted by a Turing Machine if:

- a) It is recursively enumerable
- b) It is recursive
- c) L can be enumerated by some Turing machine
- d) None of the mentioned

Answer: b

Explanation: A language L is recursively enumerable if and only if it can be enumerated by some Turing machine. A recursively enumerable language may or may not be recursive.

266. Which of the following regular expression resembles the given diagram?



Theory of Computation and Compilers Unit – 8 MCQs

- a) $\{a\}^*\{b\}^*\{a,b\}$
- b) $\{a,b\}^*\{aba\}$
- c) $\{a,b\}^*\{bab\}$
- d) $\{a,b\}^*\{a\}^*\{b\}^*$

Answer: b

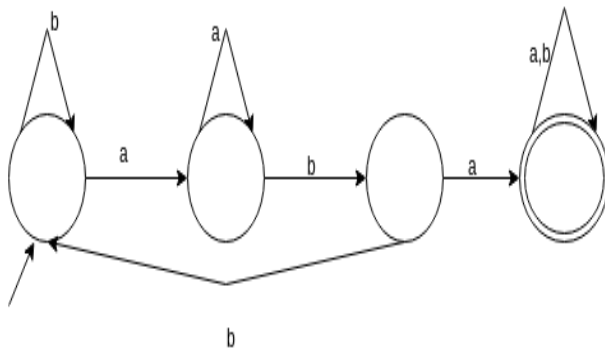
Explanation: The given diagram is a transition graph for a turing machine which accepts the language with the regular expression $\{a,b\}^*\{aba\}$.

267. The number of states required to automate the last question i.e. $\{a,b\}^*\{aba\}\{a,b\}^*$ using finite automata:

- a) 4
- b) 3
- c) 5
- d) 6

Answer: a

Explanation: The finite automata can be represented as:



268. The machine accept the string by entering into hA or it can:

- a) explicitly reject x by entering into hR
- b) enter into an infinite loop
- c) Both (a) and (b)
- d) None of the mentioned

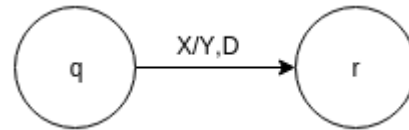
Answer: c

Explanation: Three things can occur when a

string is tested over a turing machine:

- a) enter into accept halting state
- b) enter into reject halting state
- c) goes into loop forever

269. $d(q,X)=(r,Y,D)$ where D cannot be:



- a) L
- b) R
- c) S
- d) None of the mentioned

Answer: c

Explanation: D represents the direction in which automata moves forward as per the queue which surely cannot be a starting variable.

270. Which of the following can accept even palindrome over $\{a,b\}$

- a) Push down Automata
- b) Turing machine
- c) NDFA
- d) All of the mentioned

Answer: c

Explanation: A language generating strings which are palindrome is not regular, thus cannot be represented using a finite automaton.

271. A turing machine has _____ number of states in a CPU.

- a) finite
- b) infinite
- c) May be finite
- d) None of the mentioned

Theory of Computation and Compilers Unit – 8 MCQs

Answer: a

Explanation: A turing machine has finite number of states in its CPU. However the states are not small in number. Real computer consist of registers which can store values (fixed number of bits).

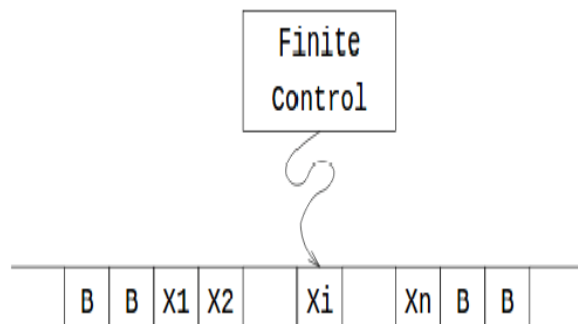
272. Suppose we have a simple computer with control unit holding a PC with a 32 bit address + Arithmetic unit holding one double length 64 bit Arithmetic Register. The number of states the finite machine will hold:

- a) $2^{(32*64)}$
- b) 2^{96}
- c) 96
- d) 32

Answer: b

Explanation: According to the statistics of the question, we will have a finite machine with 2^{96} states.

273. In one move a turing machine will:



- a) Change a state
- b) Write a tape symbol in the cell scanned
- c) Move the tape head left or right
- d) All of the mentioned

Answer: d

Explanation: A move of a turing machine is

the function of the state of finite control and the tape symbol just scanned.

273. A turing machine with several tapes is known as:

- a) Multi-tape turing machine
- b) Poly-tape turing machine
- c) Universal turing machine
- d) All of the mentioned

Answer: a

Explanation: A multitape turing machine is an ordinary turing machine with multiple tapes. Each tape has its own head to control the read and write.

274. A multitape turing machine is _____ powerful than a single tape turing machine.

- a) more
- b) less
- c) equal
- d) none of the mentioned

Answer: a

Explanation: The multitape turing machine model seems much powerful than the single tape model, but any multi tape machine, no matter how many tapes, can be simulated by single taped TM.

275. In what ratio, more computation time is needed to simulate multitape turing machines using single tape turing machines?

- a) doubly
- b) triple
- c) quadratically
- d) none of the mentioned

Theory of Computation and Compilers Unit – 8 MCQs

Answer: c

Explanation: Thus, multitape turing machines cannot calculate any more functions than single tape machines.

276. State true or false:

Statement: Multitape turing machine have multi tapes where each tape is accessed with one head.

- a) true
- b) false

Answer: b

Explanation: Multitape turing machines do have multiple tapes but they are accessed by separate heads.

277. X is a simple mathematical model of a computer. X has unrestricted and unlimited memory. X is a FA with R/W head. X can have an infinite tape divided into cells, each cell holding one symbol.

Name X?

- a) Push Down Automata
- b) Non deterministic Finite Automata
- c) Turing machines
- d) None of the mentioned

Answer: c

Explanation: Turing machine is known as universal computer. It is denoted by $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

278. Which of the following is/are not an application of turing machine?

- a) Language Recognition
- b) Computers of functions on non negative numbers
- c) Generating devices
- d) None of the mentioned

Answer: d

Explanation: A turing machine can have many applications like : Enumerator (A turing machine with an output printer), function computer, etc.

279. State true or false:

Statement: Turing Machine can change symbols on its tape, whereas the FA cannot change symbols on tape.

- a) true
- b) false

Answer: a

Explanation: The following mentioned is the difference between 2-way FA and TM.

Another instance is that TM has a read/write tape head while FA doesn't.

280. Which of the following cannot be a possibility of a TM while it processes an input?

- a) Enters accepting state
- b) Enters non-accepting state
- c) Enters infinite loop and never halts
- d) None of the mentioned

Answer: d

Explanation: The following mentioned are the only possibilities of operating a string through a turing machine.

281. Pick the odd one out.

- a) Subroutines
- b) Multiple tracks
- c) Shifting over
- d) Recursion

Answer: d

Explanation: Except Recursion, all the other

Theory of Computation and Compilers Unit – 8 MCQs

options are techniques of Turing Machine construction which further includes, Checking off symbols and Storage in finite control.

282. Fill in the blank with an appropriate option.

In automata theory, _____ is said to be Computationally Universal if can be used to simulate any single taped Turing Machine.

- a) Computer's instruction set
- b) A programming language
- c) Cellular Automaton
- d) All of the mentioned

Answer: d

Explanation: Computationally Universal or Turing Complete is a set of data manipulation rules if it can be used to simulate a single-taped turing machine.

283. Give a classic example of the concept of turing complete.

- a) lambda calculus
- b) C++
- c) Lisp
- d) All of the mentioned

Answer: d

Explanation: Most of the programming languages, conventional or unconventional are turing complete. Functional languages like Lisp and Haskell are also turing complete.

284. Let two machines be P and Q. The state in which P can simulate Q and Q can simulate P is called:

- a) Turing Equivalence

- b) State Equivalence
- c) Universal Turing Machine
- d) None of the mentioned

Answer: a

Explanation: It is a closely related concept with Turing complete. It says, two computers P and Q are called equivalent if P can simulate Q and Q can simulate P.

285. Which of the following technique is used to find whether a natural language isnt recursive enumerable?

- a) Diagonalization
- b) Recursive Induction
- c) Both (a) and (b)
- d) None of the mentioned

Answer: a

Explanation: To find a non recursively enumerable language, we use the technique of diagonalization.

286. Diagonalization can be useful in:

- a) To find a non recursively enumerable language
- b) To prove undecidablility of haltig problem
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: Diagonalization is a technique we use for the following operations:

- a) To find a non recursively enumerable language.
- b) To prove undecidablility of halting problem.

Theory of Computation and Compilers Unit – 8 MCQs

287. Which of the following are undecidable problems?

- a) Determining whether two grammars generate the same language
- b) Determining whether a grammar is ambiguous
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: In contrast we can put up an algorithm for checking whether two FA's are equivalent and this program can be implemented as a program.

288. Which of the following are incorrect options?

- a) Informally, problem is a yes/no question about an infinite set of possible instances
- b) Formally, a problem is a language
- c) Both (a) and (b)
- d) None of the mentioned

Answer: d

Explanation: Example: Does a graph G has a Hamilton cycle?

=>Each undirected graph is an instance of Hamilton cycle problem.

289. If a problem has an algorithm to answer it, we call it _____

- a) decidable
- b) solved
- c) recognizable
- d) none of the mentioned

Answer: a

Explanation: An algorithm is a TM that halts on all inputs, accepted or not. Putting other

way, decidable problems are recursive languages.

290. Which of the following are decidable problems?

- a) Can a particular line of code in a program ever be executed?
- b) Do two given CFG's generate the same language
- c) Is a given CFG ambiguous?
- d) None of the mentioned

Answer: d

Explanation: All of the mentioned problems are undecidable.

291. Which of the following is true for The Halting problem?

- a) It is recursively enumerable
- b) It is undecidable
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: Halting problem: Does a given Turing machine M halt on a given input w?

292. The decision problem is the function from string to _____

- a) char
- b) int
- c) boolean
- d) none of the mentioned

Answer: c

Explanation: The decision problem requires checking of input (string) has some property or not. That is a string to boolean transaction.

Theory of Computation and Compilers Unit – 8 MCQs

293. A language L is said to be _____ if there is a turing machine M such that $L(M)=L$ and M halts at every point.

- a) Turing acceptable
- b) decidable
- c) undecidable
- d) none of the mentioned

Answer: b

Explanation: Decidability refers to the decision problem and existence of a effective method for determining membership, and return true and false accordingly rather than going into a loop forever.

294. Which among the following are undecidable theories?

- a) The first order theory of boolean algebra
- b) The first order theory of Euclidean geometry
- c) The first order theory of hyperbolic geometry
- d) The first order theory of the natural number with addition, multiplication, and equality

Answer: d

Explanation: Tarski and Mostowski in 1949, established that the first order theory of natural numbers with addition, multiplication, and equality is an undecidable theory. Others mentioned are decidable theories.

295. $\text{Rec-DFA} = \{ \mid M \text{ is a DFA and } M \text{ recognizes input } w \}$.

Fill in the blank:

Rec-DFA is _____

- a) Undecidable
- b) Decidable
- c) Non finite
- d) None of the mentioned

Answer: b

Explanation: Under decidability of regular language properties we have the following lemma which states that a DFA which recognizes an input w is decidable.

296. Which among the following are semi decidable?

- a) Empty-DFA
- b) Rec-NFA
- c) Infinite-DFA
- d) All of the mentioned

Answer: d

Explanation: All are the properties of regular languages and all are decidable languages.

297. The language accepted by a turing machine is called _____

- a) Recursive Enumerable
- b) Recursive
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: The language accepted by Turing machines are called recursively enumerable (RE), and the subset of RE languages that are accepted by a turing machine that always halts are called recursive.

298. Decidable can be taken as a synonym to:

Theory of Computation and Compilers Unit – 8 MCQs

- a) recursive
- b) non recursive
- c) recognizable
- d) none of the mentioned

Answer: a

Explanation: We can refer to languages as 'recursive' and problems as 'decidable'. If a language is not recursive, then we call the problem expressed by that language undecidable.

299. The problems which have no algorithm, regardless of whether or not they are accepted by a turing machine that fails to halts on some input are referred as:

- a) Decidable
- b) Undecidable
- c) Computable
- d) None of the mentioned

Answer: b

Explanation: The problems that can be solved by a turing machine can divided into two classes:

- a) Those that have an algorithm
- b) Intractable problems: Those that are only solved by a turing machine that may run forever on inputs they do not accept.

300. Recursive languages are also known as:

- a) decidable
- b) undecidable
- c) sometimes decidable
- d) none of the mentioned

Answer: a

Explanation: A language is recursive if there exists a turing machine such that it halts i.e. accepts if the input belongs to the language

else rejects. It is better called Turing decidable language.

301. The class of recursive language is known as:

- a) R
- b) RC
- c) RL
- d) All of the mentioned

Answer: a

Explanation: R is the set of all recursive languages, a class of decision problems solvable by turing machines. Although, R is also used for the class RP.

302. Which of the following was not a part of Chomsky hierarchy ?

- a) Context sensitive grammar
- b) Unrestricted grammar
- c) Recursive grammar
- d) None of the mentioned

Answer: c

Explanation: All recursive languages are recursively enumerable. All regular, context free and context sensitive languages are recursive.

303. According to the rice's theorem, If P is a non trivial property, L_P is :

- a) infinite
- b) decidable
- c) undecidable
- d) none of the mentioned

Answer: c

Explanation: Rice's theorem states that 'Any non trivial property about the language

Theory of Computation and Compilers Unit – 8 MCQs

recognized by a turing machine is undecidable’.

304. Fill in the blank with reference to Rice’s theorem.

For any non-trivial property of _____ no general or effective method can decide whether an algorithm computes it with that property.

- a) partial functions
- b) piecewise functions
- c) both (a) and (b)
- d) none of the mentioned

Answer: a

Explanation: A property of partial functions is called trivial if it holds for all partial computable functions or for none, and an effective decision method is called general if it decides correctly for every algorithm.

305. Which of the following is incorrect according to rice theorem?

Let S be a set of language hat is non trivial:

- a) there exists a TM that recognizes the language in S
- b) there exists a TM that recognizes the language not in S
- c) both (a) and (b)
- d) none of the mentioned

Answer: c

Explanation: According to rice theorem, it is undecidable to determine whether the language recognized by an arbitrary turing machine lies in S.

306. Which of the following set of computable functions are decidable?

- a) The class of computable functions that

are constant, and its complement

- b) The class of indices for computable functions that are total
- c) The class of indices for recursively enumerable sets that are cofinite
- d) All of the mentioned

Answer: d

Explanation: According to Rice’s theorem, if there exists atleast one computable function in a particular class C of computable functions and another computable function not in C then the problem deciding whether a particular program computes a function in C is undecidable.

307. Which of the following statements are undecidable?

For a given Turing Machine M,

- a) does M halt on an empty input tape
- b) does M halt for any inputs at all?
- c) is L(M) regular? Context free? Turing decidable?
- d) all of the mentioned

Answer: d

Explanation: All of the following mentioned are immediate results of Rice’s theorem and thus, undecidable.

308. Post Correspondence problem is

- a) decidable decision problem
- b) undecidable decision problem
- c) not a decision problem
- d) none of the mentioned

Answer: b

Explanation: Post Correspondence problem is an undecidable decision problem that

Theory of Computation and Compilers Unit – 8 MCQs

was introduced by Emil Post in 1946. Being simpler than halting problem, it can be used in proofs of undecidability.

309. State true or false:

Statement: The difference between PCP and MPCP is that in MPCP, a solution is required to start with the first string on each list.

- a) true
- b) false

Answer: a

Explanation: The MPCP is : Given lists A and B of K strings ,say $A = w_1, w_2, \dots, w_k$ and $B = x_1, x_2, \dots, x_k$ does there exists a sequence of integers $i_1, i_2, \dots, i_r$ such that $w_{i_1}w_{i_2}\dots w_{i_r} = x_{i_1}x_{i_2}\dots x_{i_r}$?

310. PCP stands for?

- a) Post Correspondence Problem
- b) Post Corresponding Problem
- c) Pre Correspondence problem
- d) None of the mentioned

Answer: a

Explanation: PCP or Post Correspondence problem is an undecidable decision problem.

311. Can a Modified PCP problem be reduced to PCP?

- a) yes
- b) no

Answer: a

Explanation: Yes, it can be. There exists a theorem and as well as its proof which supports the assertion.

312. If the number of steps required to solve a problem is $O(n^k)$, then the problem is said to be solved in:

- a) non-polynomial time
- b) polynomial time
- c) infinite time
- d) none of the mentioned

Answer: b

Explanation: Most of the operations like addition, subtraction, etc as well as computing functions including powers, square roots and logarithms can be performed in polynomial time. In the given question, n is the complexity of the input and k is some non negative integer.

313. The value of constants like p and e can be calculated in:

- a) polynomial time
- b) non-polynomial time
- c) cannot be calculated
- d) none of the mentioned

Answer: a

Explanation: The value of such constants can be calculated using algorithms which have time complexity in terms of $O(n^k)$ i.e polynomial time.

313. Which of the following cannot be solved using polynomial time?

- a) Linear Programming
- b) Greatest common divisor
- c) Maximum matching
- d) None of the mentioned

Answer: d

Explanation: In graph theory, a matching or independent edge set in a graph G is a set

Theory of Computation and Compilers Unit – 8 MCQs

of edges without common vertices. Given a graph (V, E) , a matching M in G is a set of pairwise non adjacent edges i.e. no two edges share a common vertex.

314. The complexity class P consist of all the decision problems that can be solved by _____ using polynomial amount of computation time.

- a) Push Down automata
- b) DFA
- c) NDFA
- d) Deterministic Turing machine

Answer: d

Explanation: All the decision problems that can be solved using a Deterministic turing machine using polynomial time to compute, all belong to the complexity class P .

315. A generalization of P class can be:

- a) PTIME
- b) DTIME
- c) NP
- d) None of the mentioned

Answer: c

Explanation: P is a specific case of NP class, which is the class of decidable problems decidable by a non deterministic turing machine that runs in polynomial time.

316. Which of the following options are correct with reference to P -complete problems?

- a) used for the problems which are difficult to solve in limited space
- b) every problem in P can be reduced to it using proper reductions

- c) complete problem for complexity class P
- d) all of the mentioned

Answer: d

Explanation:

The notion of P -complete decision problems is useful in the analysis of:

- a) which problems are tough to parallelize effectively
- b) which problems are difficult to solve in limited space

317. What does NP stands for in complexity classes theory?

- a) Non polynomial
- b) Non-deterministic polynomial
- c) Both (a) and (b)
- d) None of the mentioned

Answer: b

Explanation: NP is said to be one of the most fundamental complexity classes. NP is an acronym for Non deterministic polynomial time.

318. The hardest of NP problems can be:

- a) NP-complete
- b) NP-hard
- c) P
- d) None of the mentioned

Answer: a

Explanation: NP class contains many important problems, the hardest of which is NP-complete, whose solution is sufficient to deal with any other NP problem in polynomial time.

319. Travelling sales man problem belongs to which of the class?

Theory of Computation and Compilers Unit – 8 MCQs

- a) P
- b) NP
- c) Linear
- d) None of the mentioned

Answer: b

Explanation: Travelling Salesman Problem:
Given an input matrix of distances between n cities, this problem is to determine if there is a route visiting all cities with total distance less than k .

320. State true or false?

Statement: If a problem X is in NP and a polynomial time algorithm for X could also be used to solve problem Y in polynomial time, then Y is also in NP.

- a) true
- b) false

Answer: a

Explanation: This is just a commutative property of NP complexity class where a problem is said to be in NP if it can be solved using an algorithm which was used to solve another NP problem in polynomial amount of time.

321. A problem which is both _____ and _____ is said to be NP complete.

- a) NP, P
- b) NP, NP hard
- c) P, P complete
- d) None of the mentioned

Answer: a

Explanation: A problem is said to be NP Hard if an algorithm for solving the problem can be translated from for solving any other

problem. It is easier to show a problem NP than showing it NP Hard.

322. In terms of NTIME, NP problems are the set of decision problems which can be solved using a non deterministic machine in _____ time.

- a) $O(n)$
- b) $O(n^{1/2})$
- c) $O(n^k)$, $k \in \mathbb{N}$
- d) None of the mentioned

Answer: c

Explanation: The complexity class NP can be defined in terms of NTIME as:
 $NP = O(n^k)$ for $k \in \mathbb{N}$.

323. Which of the following can be used to define NP complexity class?

- a) Verifier
- b) Polynomial time
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: NP can be defined using deterministic turing machines as verifiers.

324. Which of the following are not in NP?

- a) All problems in P
- b) Boolean Satisfiability problems
- c) Integer factorization problem
- d) None of the mentioned

Answer: d

Explanation: This is a list of some problems which are in NP:

- a) All problems in P
- b) Decision version of Integer factorization method

Theory of Computation and Compilers Unit – 8 MCQs

- c) Graph Isomorphism Problem
- d) All NP complete problems, etc.

325. Which of the following does not belong to the closure properties of NP class?

- a) Union
- b) Concatenation
- c) Reversal
- d) Complement

Answer: d

Explanation: It is unknown about the closure property-complement for the complexity class NP. The question is so called NP versus co-NP problem.

326. Which of the given problems are NP-complete?

- a) Node cover problems
- b) Directed Hamilton Circuit Problem
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: Vertex cover or Node cover problem, and Hamilton Circuit problem, both are NP complete type of problems.

327. Which of the following problems do not belong to Karp's 21 NP-complete problems?

- a) Vertex Cover problems
- b) Knapsack
- c) 0-1 integer programming
- d) None of the mentioned

Answer: d

Explanation: There exists a set of 21

problems that are NP-complete and the set is called Karp's 21 NP-complete problems.

328. Which of the following problems were reduced to Knapsack?

- a) Exact Cover
- b) Max Cut
- c) 0-1 integer programming
- d) None of the mentioned

Answer: a

Explanation: Exact cover is a decision problem in computer science to determine if an exact cover exists.

329. An exact cover problem can be represented using:

- a) incidence matrix
- b) bipartite graph
- c) both (a) and (b)
- d) none of the mentioned

Answer: c

Explanation: The relation 'contains' can be represented using a bipartite graph. The vertices of the graph can be divided into two disjoint sets, one representing the subset S and the other representing the elements of P and one edge for each subset in S; each node is included in exactly one of the edges forming the cover.

330. For which of the following, greedy algorithm finds a minimal vertex cover in polynomial time?

- a) tree graphs
- b) bipartite graphs
- c) both (a) and (b)
- d) none of the mentioned

Theory of Computation and Compilers Unit – 8 MCQs

Answer: a

Explanation: For bipartite graphs, Konigs theorem allows the bipartite vertex problem to be solved in polynomial time.

331. All set of polynomial questions which can be solved by a turing machine using a polynomial amount of space:

- a) PSPACE
- b) NPSPACE
- c) EXPSPACE
- d) None of the mentioned

Answer: a

Explanation: PSPACE is the problem class which contains all set of decision problems which can be solved using a turing machine taking polynomial amount of space.

332. A randomized algorithm uses random bits as input inorder to achieve a _____ good performance over all possible choice of random bits.

- a) worst case
- b) best case
- c) average case
- d) none of the mentioned

Answer: c

Explanation: A randomized algorithm is an algorithm that employs a degree of randomness as a part of its logic using random bits as inputs and in hope of producing average case good performace.

333. Which of the following options match the given statement:

Statement: The algorithms that use the random input to reduce the expected running time or memory usage, but always terminate with a correct result in a bounded amount of time.

- a) Las Vegas Algorithm
- b) Monte Carlo Algorithm
- c) Atlantic City Algorithm
- d) None of the mentioned

Answer: a

Explanation: The other type of algorithms are probabalistic algorithms, which depending upon the random input, have a chance of producing incorrect results or fail to produce a result.

334. Which of the following are probalistic algorithms?

- a) Las Vegas Algorithm
- b) Monte Carlo Algorithm
- c) Atlantic City Algorithm
- d) All of the mentioned

Answer: d

Explanation: Monte Carlo algorithms are very vast, but only probably correct. On thr other side, Las Vegas algorithms are always correct, but probably fast.

335. Which of the following algorithms are probably correct as well as fast?

- a) Las Vegas Algorithm
- b) Monte Carlo Algorithm
- c) Atlantic City Algorithm
- d) All of the mentioned

Answer: c

Explanation: The atlantic city algorithms which are bounded polynomial time algorithms are probably correct and probably fast. It is correct more than 75% of the times.

336. Prisonner's dilemma can be related to the following:

- a) cooperative behaviour
- b) graph theory
- c) Both (a) and (b)

Theory of Computation and Compilers Unit – 8 MCQs

d) None of the mentioned

Answer: a

Explanation: Prisoner's dilemma is a standard example of a game analysed in game theory where rational cooperative behaviour is judged on the basis of rewards and punishment.

337. State true or false:

Statement: A turing machine has the capability of using randomly 'generated' numbers.

- a) true
- b) false

Answer: a

Explanation: Complexity theories models randomized algorithms as probabilistic turing machines. A probabilistic turing machine is a non deterministic turing machine which randomly chooses between the available transitions at each point according to some probabilistic distribution.

338. For the given algorithm, find the probability of finding after k iterations:

```
find_a(array A, n, k)
begin
  i=0
  repeat
    Randomly select one element out of n
    elements
    i=i+1
  until i=k or a is found
end
```

- a) $(1/2)^k$
- b) $(1-(1/3))^k$
- c) $1-(1/2)^k$
- d) None of the mentioned

Answer: c

Explanation: The given is known as Monte

Carlo Algorithm. If a is found, the algorithm succeeds, else the algorithm fails. The algorithm doesn't guarantee success but the run time is bounded.

339. Which of the following can be solved in computer science?

- a) P=BPP problem
- b) NP=co-NP problem
- c) Do one way problems exist?
- d) All of the mentioned

Answer: d

Explanation: There exists a list of unsolved problems in computational theory which includes many problems including the ones given.

340. Which of the following can be referred to as applications of Randomized algorithm?

- a) Quicksort
- b) Min Cut
- c) Verifying Matrix Multiplication
- d) All of the mentioned

Answer: d

Explanation: Freivalds algorithm is a probabilistic randomized algorithm we use to verify matrix multiplication. On the other hand, Randomness can be useful in quicksort. If the algorithm selects pivot element uniformly at random, it has a probably high probability of finishing the work in $O(n \log n)$ time regardless of the input.

341. Which among the following is smallest for $n=50$

- a) $2n^2$
- b) n^2+3n+7
- c) n^3
- d) $2n$

Theory of Computation and Compilers Unit – 8 MCQs

Answer: b

Explanation:

$$2n^2=5000$$

$$n^2+3n+7=2567$$

$$n^3=125000$$

$$2n=1.13 \times 10^{15}$$

342. The space complexity of a turing machine is undefined if:

- a) It is a multitape turing machine
- b) If no string of length n causes T to use infinite number of tape squares
- c) If some input of length n causes T to loop forever
- d) None of the mentioned

Answer: c

Explanation: If there exists an input string of length n that causes T to use an infinite number of tape squares, the space complexity of the turing machine is undefined.

343. In order to reduce the run time of a turing machine:

- a) we can reduce the number of tapes
- b) we can increase the number of tapes
- c) use infinite tapes
- d) none of the mentioned

Answer: One way to reduce the run time can be to increase the number of tapes. Sometimes, using two tapes can be used to avoid back and forth motions altogether.

344. Which of the following are basic complexity classes for a function $f:N \rightarrow N$?

- a) $Ntime(f)$
- b) $Nspace(f)$

c) $Space(f)$

d) All of the mentioned

Answer: d

Explanation: $Ntime(f)$: is a set of languages that can be accepted by a NTM T with non deterministic time complexity function $t \leq f$. In all four cases, the machines are allowed to be multitape TM's.

345. A function f is called _____ if there exists a TM T so that for any n and any input string of length n , T halts in exactly $f(n)$ moves.

- a) Step function
- b) Step counting function
- c) Inplace functions
- d) None of the mentioned

Answer: b

Explanation: If f is a step counting function, T is a TM halting in $f(n)$ moves where n is the length of input string.

346. Let $f: N \rightarrow N$ be a step counting function. Then for some constant C , $Time(f)$ is a proper subset of $Time(\text{_____})$

- a) $O(nf)$
- b) $O(n+f)$
- c) $O(n^2f^2)$
- d) None of the mentioned

Answer: c

Explanation: Using the encoding function, it is possible to show that if the function f is a step counting function, then the function $Cn^2(f(n))^2$ is the total number of moves required.

Theory of Computation and Compilers Unit – 8 MCQs

347. Which of the following are valid membership algorithms?

- a) CYK algorithm
- b) Exhaustive search parser
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: CYK algorithm is a parsing algorithm for context free grammars, which employs bottom up parsing and dynamic programming.

348. Which of the following belong to the steps to prove emptiness?

- a) Remove useless variable
- b) Check if a start variable S is useless
- c) Both (a) and (b)
- d) None of the mentioned

Answer: c

Explanation: The empty-language question can be stated as: For context free grammar G find if $L(G) = \emptyset$?

349. Which of the following is true for CYK Algorithm?

- a) Triangular Table
- b) Circular Chart
- c) Linked List
- d) None of the mentioned

Answer: a

Explanation: A triangular table is constructed to facilitate the solution of membership problem using bottom up parsing and dynamic programming.

350. Which of the following steps are wrong with respect to infiniteness problem?

- a) Remove useless variables
- b) Remove unit and epsilon production
- c) Create dependency graph for variables
- d) If there is a loop in the dependency graph the the language is finite else infinite

Answer: d

Explanation: If we are able to detect a loop in the formed dependency graph, then the language is infinite.

351. State true or false:

Statement: Every context free language can be generated by a grammar which contains no useless non terminals.

- a) true
- b) false

Answer: a

Explanation: At first, we detect useless symbols and discard them. Inorder to find whether a symbol is useless, just make it the starting symbol and check for emptiness.

352. Consider the following grammar.

```
S -> S * E
S -> E
E -> F + E
E -> F
F -> id
```

Consider the following LR (0) items corresponding to the grammar above.

- (i) $S \rightarrow S * .E$
- (ii) $E \rightarrow F. + E$
- (iii) $E \rightarrow F + .E$

Given the items above, which two of them will appear in the same set in the canonical

Theory of Computation and Compilers Unit – 8 MCQs

sets-of-items for the grammar?

- a) (ii)
- b) (i) and (iii)
- c) (iii)
- d) None of the mentioned

Answer: C

Explanation: If $S \rightarrow S$: E is in LR (0) then $E \rightarrow F +$: E will also be there because both of them has ' : ' before E.

353. Consider the following grammar:

```
S -> FR
R -> * S | ε
F -> id
```

In the predictive parser table, M, of the grammar the entries M [S, id] and M [R, \$] respectively.

- a) {S " FR} and {R " ε}
- b) {S " FR} and {}
- c) {S " FR} and {R " * S}
- d) {F " id} and {R " ε}

Answer: a

Explanation: The predictive parser table is given as. Non Terminal) id \$ S S "FR F F "id R R ") S R " ! R " ! So at M [S, id] = { S " FR} M [R,\$] = {R " !}.

354. Consider the following translation scheme.

```
S -> ER
R -> * E{print(' * ');}
R | f
E -> F + E{print(' + ');} | F
F -> (S) | id{print(id.value);}
```

Here id is a token that represents an integer and id. value represents the corresponding integer value. For an input '2 * 3 + 4', this translation scheme prints?

- a) 2 * 3 + 4

- b) 2 * + 3 4
- c) 2 3 * 4 +
- d) 2 3 4 + *

Answer: b

Explanation: Input string 2) 3 + 4 S " ER FR idR {print(2)} id)ER {print())} id) F+ER {print(+)}id) id + ER {print(3)} id) id) id +id So 2)+ 3 4 are printed.

355. Consider the following C code segment.

```
for for if i # i } }
```

Which one to the following false?

- a) The code contains loop-in variant computation
- b) There is scope of common sub-expression elimination in this code
- c) There is scope strength reduction in this code
- d) There is scope of dead code elimination in this code

Answer: d

Explanation: All the statements are true except option (There is scope of dead code elimination in this code) since there is no dead code to get eliminated.

356. Which of these is also known as look-head LR parser?

- a) SLR
- b) LR
- c) LLR
- d) None of the mentioned

Answer: c

Explanation: LLR is the look ahead parser.

357. What is the similarity between LR, LALR and SLR?

Theory of Computation and Compilers Unit – 8 MCQs

- a) Use same algorithm, but different parsing table
- b) Same parsing table, but different algorithm
- c) Their Parsing tables and algorithm are similar but uses top down approach
- d) Both Parsing tables and algorithm are different

Answer: a

Explanation: The common grounds of these 3 parser is the algorithm but parsing table is different.

358. An LR-parser can detect a syntactic error as soon as _____

- a) The parsing starts
- b) It is possible to do so a left-to-right scan of the input
- c) It is possible to do so a right-to-left scan of the input
- d) Parsing ends

Answer: b

Explanation: Error is found when it the input string is scanned.

359. Which of these is true about LR parsing?

- a) Is most general non-backtracking shift-reduce parsing
- b) It is still efficient
- c) Is most general non-backtracking shift-reduce parsing & It is still efficient
- d) None of the mentioned

Answer: c

Explanation: LR parsers are a type of bottom-up parsers that efficiently handle

deterministic context-free languages in guaranteed linear time.

360. Which of the following is incorrect for the actions of A LR-Parser I) shift s ii) reduce A->β iii) Accept iv) reject?

- a) Only I)
- b) I) and ii)
- c) I), ii) and iii)
- d) I), ii) , iii) and iv)

Answer: c

Explanation: Only reject out of the following is a correct LR parser action.

361. What is terminal table?

- a) Contains all constants in the program
- b) Is a permanent table of decision rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure
- c) Consist of a full or partial list of the token is as they appear in the program created by lexical analysis and used for syntax analysis and interpretation
- d) Is a permanent table which lists all keywords and special symbols of the language in symbolic form

Answer: d

Explanation: A permanent database that has entry for each terminal symbols such as arithmetic operators, keywords, punctuation characters such as ‘;’, ‘,’ etc
Fields: Name of the symbol.

362. Advantage of incorporating the macro-processor into pass 1 is that _____

- a) Many functions have to be implemented twice

Theory of Computation and Compilers Unit – 8 MCQs

- b) Functions are combined not necessarily creating intermediate files as output from the macro-processor and input to the assembler
- c) More flexibility is provided to the programmer in that he may use all the features of the assembler in conjunction with macros
- d) All of the mentioned

Answer: d

Explanation: A general-purpose macro processor or general purpose pre-processor is a macro designed primarily for string manipulation, macro definition.

363. Which of the following is a phase of a compilation process?
- a) Lexical Analysis
- b) Code Generation
- c) Lexical Analysis & Code Generation
- d) None of the mentioned

Answer: c

Explanation: Lexical analysis and code generation is a phase of compilation process.

364. System program such as compiler are designed so that they are _____
- a) Re-enterable
- b) Non reusable
- c) Serially usable
- d) None of the mentioned

Answer: a

Explanation: Re-enterable is the keyword for compiler being designed.

365. A series of statements explaining how the data is to be processed is called _____

- a) Assembly
- b) Machine
- c) COBOL
- d) Program

Answer: d

Explanation: A program is a sequence of instructions, written to perform a task by computer. It requires programs to function, typically executing the program's instructions in a central processor.

366. In a single pass assembler, most of the forward references can be avoided by putting the restriction _____
- a) On the number of strings/life reacts
- b) Code segment to be defined after data segment
- c) On unconditional rump
- d) None of the mentioned

Answer: b

Explanation: A single pass assembler scans the program only once and creates the equivalent binary program.

367. The method which merges the bodies of two loops is?
- a) Loop rolling
- b) Loop jamming
- c) Constant folding
- d) None of the mentioned

Answer: b

Explanation: In computer science, loop fusion (or loop jamming) is a compiler

Theory of Computation and Compilers Unit – 8 MCQs

optimization and loop transformation which replaces multiple loops with a single one.

368. Assembly code data base is associated with _____

- a) Code is converted into assembly
- b) Table of rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure
- c) All of the mentioned
- d) None of the mentioned

Answer: a

Explanation: An assembly language is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions.

369. The process manager has to keep track of _____

- a) Status of each program
- b) Information to a programmer using the system
- c) Status of each program & Information to a programmer using the system
- d) None of the mentioned

Answer: c

Explanation: Process manager keep track of the status and info about the program.

370. What is the function of the syntax phase?

- a) recognize the language and to call the appropriate action routines that will generate the intermediate form or matrix for these constructs

- b) Build a literal table and an identifier table
- c) Build a uniform symbol table
- d) Parse the source program into the basic elements or tokens of the language

Answer: a

Explanation: In this phase symbol table is created by the compiler which contains the list of lexemes or tokens.

371. Which is not true about syntax and semantic parts of a computer language?

- a) Semantics is checked mechanically by a computer
- b) Semantics is the responsibility of the programmer
- c) All of the mentioned
- d) None of the mentioned

Answer: d

Explanation: Both in terms of semantics is not true.

372. Which of the following statement is true?

- a) SLR powerful than LALR
- b) LALR powerful than Canonical LR parser
- c) Canonical LR powerful than LALR parser
- d) The parsers SLR= Canonical LR=LALR

Answer: c

Explanation: LR > LALR > SLR In terms of the parser.

373. Which of the following features cannot be captured by CFG?

- a) Syntax of if-then-else statements
- b) Syntax of recursive procedures
- c) A variable is declared before its use
- d) Matching nested parenthesis

Theory of Computation and Compilers Unit – 8 MCQs

Answer: d

Explanation: It is because, it is equivalent to recognizing us, where the first w is the declaration and the second is its use, we are not a CFG.

374. In which of the following no information hiding is done?

- a) Compile prig 1, prig 2
- b) Run test, prog
- c) Load R1, A
- d) 001001000010101

Answer: d

Explanation: The entire binary symbol gives information.

375. The identification of common sub-expression and replacement of run-time computations by compile-time computations is _____

- a) Local optimization
- b) Loop optimization
- c) Constant folding
- d) Data flow analysis

Answer: c

Explanation: Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals they may also be variables whose values are assigned at compile time.

376. Dynamic linking can cause security concerns because _____

- a) Security is dynamic
- b) None of the mentioned
- c) Security is dynamic & Cryptographic

procedures are not available for dynamic linking

d) Cryptographic procedures are not available for dynamic linking

Answer: d

Explanation: Required by the program are linked during run time.

377. Which of the following statements is FALSE?

- a) In up-typed languages, values do not have any types & Dynamic languages have no variables types
- b) In up-typed languages, values do not have any types
- c) Dynamic languages have no variables types
- d) None of the mentioned

Answer: c

Explanation: They have dynamically changing types.

378. What is grammar?

$S \rightarrow CC$

$C \rightarrow eC \mid d$

- a) LL (1)
- b) SLR (1) but not LL (1)
- c) LALR (1) but not SLR (1)
- d) LR (1) but not LALR (1)

Answer: c

Explanation: The grammar is CLR and can be reduced to LALR.

379. The 3-address code sequence generated by which definition?

Theory of Computation and Compilers Unit – 8 MCQs

`S -> id: = E "`

`newtemp ();`

`gen(t . place . place;); .place t} "`

`.place .place;}`

For the statement ' $X = Y + Z$ ',

a) $X = Y + Z$

b) $t1 = Y + Z; X = t1$

c) $t1 = Y; t2 = t1 + Z; X = t2$

d) $t1 = Y; t2 = Z; t3 = t2; X = t3$

Answer: d

Explanation: In 3-address code we use temporary variables.

380. If the programming language uses static scoping and call by need parameter passing mechanism, the values printed will be?

a) 115, 220

b) 25, 220

c) 25, 15

d) 115, 105

Answer: d

Explanation: So $i = 100$ & $j = 5$ $P(i + j) = P(100 + 5) = P(105)$ So $x = 105$ $x + 10 = 105 + 10 = 115$ So 115 & 105 will be printed.

381. Consider the following two statements:

P: Every regular grammar is LL(1)

Q: Every regular set has LR(1) grammar

Which of the following is TRUE?

a) Both P and Q are true

b) P is true and Q is false

c) P is false and Q is true

d) Both P and Q are false

Answer: a

Explanation: LL(1) parsers can recognize the regular grammars also LL(1) is subset of LR(1) or CLR grammar so it also recognizes regular sets. So both accept regular grammar.

382. In a simplified computer the instructions are:

OP R j, Ri – Performs $R_j \text{ OP } R_i$ and stores the result in register Ri

OP m, Ri – Performs val OP Ri and stores the result in Ri. value denotes the content of memory location m.

MCVm, Ri – Moves the content off memory location m to register Ri.

MCVm, Ri, m – Moves the content of register Ri to memory location m.

The computer has only two registers, and OP is either ADD or SUB. Consider the following basic block:

$t1 = a + b$

$t2 = c + d$

$t3 = e - t2$

$t4 = t1 - t2$

Assume that all operands are initially in memory. The final value of the computation should be in memory. What is the minimum number of MOV instructions in the code generated for this basic block?

a) 2

b) 3

Theory of Computation and Compilers Unit – 8 MCQs

- c) 5
d) 6

Answer: b

Explanation: The operation sequence would be

MOV a, R1
ADD b, R1 {R 1 = t1
MOV c, R2
ADD d, R2 {R 2 = t2
SUB e, R2 {t 3 = e – R 2 = R2
SUB R 1, R2 {R 2 = t4
MOV R 2, t4 {finally in memory
Totally no. of move operation is 3.

383. Which of the following strings is generated by the grammar?

S->bA S->aB

A->a B->b

A->aS B->bS

A->bAA B->aBB

- a) aaaabb
b) aabbbb
c) aabbab
d) abbbba

Answer: c

Explanation: aabbab S " aB " aaBB " aabSB " aabbAB " aabbab

384. How many derivation trees are there?

S->bA S->aB

A->a B->b

A->aS B->bS

A->bAA B->aBB

- a) 1
b) 2
c) 3
d) 4

Answer: b

Explanation: For the derivation two trees are possible So due to ambiguity 2 trees are possible.

385. Which of the following describes a handle (as applicable to LR-parsing) appropriately?

- a) It is the position in a sentential form where the next shift or reduce operation will occur
b) It is a non-terminal whose production will be used for reduction in the next step
c) It is a production that may be used for reduction in a future step along with a position in the sentential form where the next shift or reduce operation will occur.
d) It is the production p that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found

Answer: d

Explanation: Handles are the part of sentential form, & they are identified as the right side of any given production which will be used for reduction in the next step.

386. Which languages necessarily need heap allocation in the runtime environment?

- a) Those that support recursion

Theory of Computation and Compilers Unit – 8 MCQs

- b) Those that use dynamic scoping
- c) Allow dynamic data structure
- d) Those that use global variables

Answer: c

Explanation: E.g.: Heap.

387. Given the language $L=\{ab, aa, baa\}$, which of the following strings are in LG?

- 1) abaabaaabaa
 - 2) aaaabaaaa
 - 3) baaaaabaaaab
 - 4) baaaaabaa
- a) 1,2 and 3
 - b) 2,3 and 4
 - c) 1,2 and 4
 - d) 1,3 and 4

Answer: c

Explanation: Generates strings like
aaaabaaaa
baaaaabaa

388. In a two pass assembler, adding literals to literal table and address resolution of local symbols are done using?

- a) First pass and second respectively
- b) Both second pass
- c) Second pass and first respectively
- d) Both first pass

Answer: d

Explanation: A two pass assembler does two passes over the source file (the second pass can be over a file generated in the first pass).

389. In Two pass assembler the object code generation is done during the

-
- a) Second pass
 - b) First pass
 - c) Zeroth pass
 - d) Not done by assembler

Answer: a

Explanation: On the second pass, the assembler:

- source statements into machine code
- error messages, if error has occurred.

390. Pick the machine independent phase of the compiler.

- a) Syntax analysis
- b) Code generation
- c) Lexical analysis
- d) All of the mentioned

Answer: d

Explanation: Machine independent phases are Lexical analysis, Syntax analysis, Semantic analysis, Intermediate code generation and sometime code optimization.

391. The load instruction is mostly used to designate a transfer from memory to a processor register known as _____

- a) Accumulator
- b) Instruction Register
- c) Program counter
- d) Memory address Register

Answer: a

Explanation: Accumulator is the process register.

Theory of Computation and Compilers Unit – 8 MCQs

392. A group of bits that tell the computer to perform a specific operation is known as _____

- a) Instruction code
- b) Micro-operation
- c) Accumulator
- d) Register

Answer: a

Explanation: Instruction code is the set of specific tasks to be performed.

393. The time interval between adjacent bits is called the _____

- a) Word-time
- b) Bit-time
- c) Turnaround time
- d) Slice time

Answer: b

Explanation: Bit time is the interval time here.

394. A k-bit field can specify any one of _____

- a) 3k registers
- b) 2k registers
- c) K2 registers
- d) K3 registers

Answer: b

Explanation: 2k is the answer.

395. MIMD stands for _____

- a) Multiple instruction multiple data
- b) Multiple instruction memory data
- c) Memory instruction multiple data
- d) Multiple information memory data

Answer: a

Explanation: MIMD stands for multiple instructions multiple data.

396. The instruction 'ORG O' is a _____

- a) Machine Instruction
- b) Pseudo instruction
- c) High level instruction
- d) Memory instruction

Answer: b

Explanation: It is a pseudo instruction.

397. 'Aging registers' are _____

- a) Counters which indicate how long ago their associated pages have been Referenced
- b) Registers which keep track of when the program was last accessed
- c) Counters to keep track of last accessed instruction
- d) Counters to keep track of the latest data structures referred

Answer: a

Explanation: This is known as aging.

398. Memory unit accessed by content is called _____

- a) Read only memory
- b) Programmable Memory
- c) Virtual Memory
- d) Associative Memory

Answer: d

Explanation: Associative Memory is accessed by content.

399. The main memory in a Personal Computer (PC) is made of _____

- a) Cache memory

Theory of Computation and Compilers Unit – 8 MCQs

- b) Static RAM
- c) Dynamic Ram
- d) Cache memory & RAM

Answer: d

Explanation: It is made up of cache memory and RAM.

400. Cache memory works on the principle of _____

- a) Locality of data .
- b) Locality of memory
- c) Locality of reference
- d) Locality of reference & memory

Answer: c

Explanation: Cache works on Locality of reference.

401. An n-bit microprocessor has _____

- a) n-bit program counter
- b) n-bit address register
- c) n-bit ALU
- d) n-bit instruction register

Answer: d

Explanation: N bit microprocessor has bit instruction register.

402. When CPU is executing a Program that is part of the Operating System, it is said to be in _____

- a) Interrupt mode
- b) System mode
- c) Half mode
- d) Simplex mode

Answer: b

Explanation: It is in System Mode.

403. Logic X-OR operation of (4ACO) H & (B53F) H results _____

- a) AACB
- b) 0000
- c) FFFF
- d) ABCD

Answer: c

Explanation: Xoring the two operands we get FFFF.

403. Generally Dynamic RAM is used as main memory in a computer system as it _____

- a) Consumes less power
- b) Has higher speed
- c) Has lower cell density
- d) Needs refreshing circuitry

Answer: b

Explanation: Because of higher speed it is Dynamic.

404. Write Through technique is used in which memory for updating the data _____

- a) Virtual memory
- b) Main memory
- c) Auxiliary memory
- d) Cache memory

Answer: d

Explanation: The answer is cache memory.

405. Cache memory acts between _____

- a) CPU and RAM
- b) RAM and ROM
- c) CPU and Hard Disk
- d) None of the mentioned

Theory of Computation and Compilers Unit – 8 MCQs

Answer: a

Explanation: It acts between CPU and RAM.

406. The idea of cache memory is based

- a) On the property of locality of reference
- b) On the heuristic 90-10 rule
- c) On the fact that references generally tend to cluster
- d) All of the mentioned

Answer: a

Explanation: Cache memory is based on the locality of references.

407. Which of the following is not a weighted code?

- a) Decimal Number system
- b) Excess 3-cod
- c) Binary number System
- d) None of the mentioned

Answer: b

Explanation: Excess 3 is not a weighted code.

408. The average time required to reach a storage location in memory and obtain its contents is called the _____

- a) Seek time
- b) Turnaround time
- c) Access time
- d) Transfer time

Answer: c

Explanation: Times used to access the contents.

409. (2FAOC) 16 is equivalent to

- a) (195 084)10

b) (001011111010 0000 1100)2

c) (195 084)10 & (001011111010 0000 1100)2

d) None of the mentioned

Answer: b

Explanation: It is equivalent to (001011111010 0000 1100)2.

410. The circuit used to store one bit of data is known as _____

- a) Register
- b) Encoder
- c) Decoder
- d) Flip Flop

Answer: d

Explanation: 1 bit circuit is known as Flip Flop.

411. Computers use addressing mode techniques for _____

- a) Giving programming versatility to the user by providing facilities as pointers to memory counters for loop control
- b) To reduce number of bits in the field of instruction
- c) Specifying rules for modifying or interpreting address field of the instruction
- d) All of the mentioned

Answer: d

Explanation: All of these are addressing mode techniques.

412. Which of the following derivations does a top-down parser use while parsing an input string?

- a) Leftmost derivation
- b) Leftmost derivation in reverse

Theory of Computation and Compilers Unit – 8 MCQs

- c) Rightmost derivation
- d) Rightmost derivation in reverse

Answer: a

Explanation: In top down parser takes input from Left to right constructing leftmost derivation of the sentence.

413. The process of assigning load addresses to the various parts of the program and adjusting the code and data in the program to reflect the assigned addresses is called?

- a) Assembly
- b) Parsing
- c) Relocation
- d) Symbol resolute

Answer: c

Explanation: Relocation is the process of replacing symbolic references or names of libraries with actual usable addresses in memory before running a program. Linker performs it during compilation.

414. Which of the following statements is false?

- a) Left as well as right most derivations can be in Unambiguous grammar
- b) An LL (1) parser is a top-down parser
- c) LALR is more powerful than SLR
- d) Ambiguous grammar can't be LR (k)

Answer: a

Explanation: If a grammar has more than one leftmost (or rightmost) derivation the grammar is ambiguous. Sometimes in unambiguous grammar the rightmost derivation and leftmost derivations may differ.

415. Which of the following grammar rules violate the requirements of an operator grammar?

- (i) $P \rightarrow QR$
- (ii) $P \rightarrow QsR$
- (iii) $P \rightarrow \epsilon$
- (iv) $P \rightarrow QtRr$

- a) (i) only
- b) (i) and (iii) only
- c) (ii) and (iii) only
- d) (iii) and (iv) only

Answer: b

Explanation: An operator precedence parser is a bottom-up parser that interprets an operator-precedence grammar.

Consider the grammar with the following translation rules and E as the start symbol.

$A \rightarrow A1 \# B \{A.value = A1.value * B.value\}$

$| B \{A.value = B.value\}$

$B \rightarrow B1 \& F \{B.value = B1.value + C.value\}$

$| C \{B.value = C.value\}$

$C \rightarrow num \{C.value = num.value\}.$

416. Find the TRUE statement?

I. There exist parsing algorithms for some programming languages which has $O(3)$ complexity.

II. A programming language which allows recursion can be implemented with static storage allocation.

III. No L-attributed definition can be evaluated in The framework of bottom-up parsing.

Theory of Computation and Compilers Unit – 8 MCQs

IV. Code improving transformations can be performed at both intermediate code level and source Language.

- a) I and II
- b) I and IV
- c) III and IV
- d) I III and IV

Answer: b

Explanation: In recursion, space used but recursive call can't be calculated by the compiler.

417. Which of the following describes a handle (as applicable to LR-parsing) appropriately?

- a) Position where next reduce or shift operation will occur
- b) The next step has use of Non-terminal for reduction
- c) Used for reduction in a coming-up step along with a position in the sentential form where the next shift or reduce operation will occur
- d) Used in the next step for reduction along with a position in the sentential form where the right hand side of the production may be found

Answer: d

Explanation: the next step in LR parsing shall have a Reduction.

418. Which one of the following is a top-down parser?

- a) Recursive descent parser
- b) Operator precedence parser
- c) An LR(k) parser
- d) An LALR(k) parser

Answer: a

Explanation: Recursive Descent also known as top down parsing also known to be LL(1).

Consider the following two statements:

P: Every regular grammar is LL(1)

Q: Regular is LR(1) grammar.

419. Which of the following is TRUE?

- a) Both P and Q are true
- b) P is true and Q is false
- c) P is false and Q is true
- d) Both P and Q are false

Answer: c

Explanation: Ambiguity can be seen in regular grammar

$S \rightarrow aA/a$

$A \rightarrow aA/\epsilon$

In above grammar, string 'a' has two leftmost derivations.

$S \rightarrow aA$

$S \rightarrow a$

$S \rightarrow a$ (using $A \rightarrow \epsilon$).

420. What is the grammar for the below equations?

$S \rightarrow C C$

$C \rightarrow c C \mid d$

- a) LL(1)
- b) SLR(1) but not LL(1)
- c) LALR(1) but not SLR(1)
- d) LR(1) but not LALR(1)

Answer: a

Explanation: Since there is no conflict, the grammar is LL (1) hence a predictive parse table with no conflicts can be constructed.

Theory of Computation and Compilers Unit – 8 MCQs

421. Which of the following statements is false?

- a) Unambiguous grammar has both kind of derivations
- b) An LL(1) parser is a top-down parser
- c) LALR is more powerful than SLR
- d) Ambiguous grammar can't be LR(k)

Answer: a

Explanation: If a grammar has more than one leftmost (or rightmost) derivation the grammar is ambiguous.

422. Given the following expression grammar:

$E \rightarrow E * F \mid F + E \mid F$

$F \rightarrow F - F \mid id$

Which of the following is true?

- a) * has higher precedence than +
- b) – has higher precedence than *
- c) + and — have same precedence
- d) + has higher precedence than *

Answer: b

Explanation: e.g. input is $3*4-5$ r

E

/ | \

E * F

| / | \

F F - F

| | |

id (3) id (4) id (5)

First '-' is be evaluated then '*'

423. Which one of the following is true at any valid state in shift-reduce parsing?

- a) At the bottom we find the prefixes
- b) None of the mentioned
- c) Stack contains only viable prefixes
- d) Stack consists of viable prefixes

Answer: c

Explanation: The prefixes on the stack of a shift-reduce parser are called viable prefixes.