

# Theory of Computation Lecture Notes

---

Abhijat Vichare

August 2005

## Contents

---

- [1 Introduction](#)
- [2 What is Computation ?](#)
- [3 The  \$\lambda\$  Calculus](#)
  - [3.1 Conversions:](#)
  - [3.2 The calculus in use](#)
  - [3.3 Few Important Theorems](#)
  - [3.4 Worked Examples](#)
  - [3.5 Exercises](#)
- [4 The theory of Partial Recursive Functions](#)
  - [4.1 Basic Concepts and Definitions](#)
  - [4.2 Important Theorems](#)
  - [4.3 More Issues in Computation Theory](#)
  - [4.4 Worked Examples](#)
  - [4.5 Exercises](#)
- [5 Markov Algorithms](#)
  - [5.1 The Basic Machinery](#)
  - [5.2 Markov Algorithms as Language Acceptors and Recognisers](#)
  - [5.3 Number Theoretic Functions and Markov Algorithms](#)
  - [5.4 A Few Important Theorems](#)
  - [5.5 Worked Examples](#)
  - [5.6 Exercises](#)
- [6 Turing Machines](#)
  - [6.1 On the Path towards Turing Machines](#)
  - [6.2 The Pushdown Stack Memory Machine](#)
  - [6.3 The Turing Machine](#)
  - [6.4 A Few Important Theorems](#)
  - [6.5 Chomsky Hierarchy and Markov Algorithms](#)
  - [6.6 Worked Examples](#)
  - [6.7 Exercises](#)
- [7 An Overview of Related Topics](#)
  - [7.1 Computation Models and Programming Paradigms](#)
  - [7.2 Complexity Theory](#)
- [8 Concluding Remarks](#)
- [Bibliography](#)

## 1 Introduction

---

# Theory of Computation Lecture Notes

---

Abhijat Vichare

August 2005

## Contents

---

- [1 Introduction](#)
- [2 What is Computation ?](#)
- [3 The  \$\lambda\$  Calculus](#)
  - [3.1 Conversions:](#)
  - [3.2 The calculus in use](#)
  - [3.3 Few Important Theorems](#)
  - [3.4 Worked Examples](#)
  - [3.5 Exercises](#)
- [4 The theory of Partial Recursive Functions](#)
  - [4.1 Basic Concepts and Definitions](#)
  - [4.2 Important Theorems](#)
  - [4.3 More Issues in Computation Theory](#)
  - [4.4 Worked Examples](#)
  - [4.5 Exercises](#)
- [5 Markov Algorithms](#)
  - [5.1 The Basic Machinery](#)
  - [5.2 Markov Algorithms as Language Acceptors and Recognisers](#)
  - [5.3 Number Theoretic Functions and Markov Algorithms](#)
  - [5.4 A Few Important Theorems](#)
  - [5.5 Worked Examples](#)
  - [5.6 Exercises](#)
- [6 Turing Machines](#)
  - [6.1 On the Path towards Turing Machines](#)
  - [6.2 The Pushdown Stack Memory Machine](#)
  - [6.3 The Turing Machine](#)
  - [6.4 A Few Important Theorems](#)
  - [6.5 Chomsky Hierarchy and Markov Algorithms](#)
  - [6.6 Worked Examples](#)
  - [6.7 Exercises](#)
- [7 An Overview of Related Topics](#)
  - [7.1 Computation Models and Programming Paradigms](#)
  - [7.2 Complexity Theory](#)
- [8 Concluding Remarks](#)
- [Bibliography](#)

## 1 Introduction

---

In this module we will concern ourselves with the question:

What does computation mean ?

We first look at the reasons why we must ask this question in the context of the studies on Modeling and Simulation.

We view a *model* of an event (or a phenomenon) as a "list" of the essential features that characterize it. For instance, to model a traffic jam, we try to identify the essential characteristics of a traffic jam. Overcrowding is one principal feature of traffic jams. Yet another feature is the lack of any movement of the vehicles trapped in a jam. To avoid traffic jams we need to study it and develop solutions perhaps in the form of a few traffic rules that can avoid jams. However, it would not be feasible to study a jam by actually trying to create it on a road. Either we study jams that occur by themselves "naturally" or we can try to *simulate* them. The former gives us "live" information, but we have no way of knowing if the information has a "universal" applicability - all we know is that it is applicable to at least one real life situation. The latter approach - simulation - permits us to experiment with the assumptions and collate information from a number of live observations so that good general, universal "principles" may be inferred. When we infer such principles, we gain knowledge of the issues that cause a traffic jam and we can then evolve a list of traffic rules that can avoid traffic jams.

To *simulate*, we need a model of the phenomenon under study. We also need another well known system which can incorporate the model and "run" it. Continuing the traffic jam example, we can create a simulation using the principles of mechanical engineering (with a few more from other branches like electrical and chemical engineering thrown in if needed). We could create a sufficient number of toy vehicles. If our traffic jam model characterizes the vehicles in terms of their speed and size, we **must** ensure that our toy vehicles can have varying masses, dimensions and speeds. Our model might specify a few properties of the road, or the junction - for example the length and width of the road, the number of roads at the junction etc. A toy mechanical model must be crafted to *simulate* the traffic jam!

Naturally, it is required that we be well versed with the principles of mechanical engineering - what it can do and what it cannot. If road conditions cannot be accurately captured in the mechanical model<sup>1</sup>, then the mechanical model would be correct only within a limited range of considerations that the simulation system - the principles of mechanical engineering, in our example - can capture.

Today, computers are predominantly used as the system to perform simulation. In some cases usual engineering is still used - for example the test drive labs that car manufacturers use to test new car designs for, say safety. Since computers form the main system on which models are implemented for simulation, we need to study computation theory - the basic science of computation. This study gives us the knowledge of what computers can and cannot do.

## 2 What is Computation ?

---

Perhaps it may surprise you, but the idea of computation has emerged from deep investigation into the foundations of Mathematics. We will, however, motivate ourselves intuitively without going into the actual Mathematical issues. As a consequence, our approach in this module would be to *know* the Mathematical results in theory of Computation without regard to their proofs. We will treat excursions into the Mathematical foundations for historical perspectives, if necessary. Our definitions and statements will be rigorous and accurate.

Historically, at the beginning of the 20<sup>th</sup> century, one of the questions that bothered mathematicians was about what an *algorithm* actually is. We informally know an algorithm: a certain sort of a general method to solve a family of related questions. Or a bit more precisely: a finite sequence of steps to be performed to reach a desired result. Thus, for instance, we have an addition algorithm of integers represented in the decimal form: Starting from the least significant place, add the corresponding digits and carry forward to the next place if needed, to obtain the sum. Note that an algorithm is a *recipe* of operations to be performed. It is an appreciation of the process, independent of the actual objects that it acts upon. It therefore must use the information about the *nature* (properties) of the objects rather than the objects themselves. Also, the steps are such that *no intelligence* is required - even a *machine*<sup>2</sup> can do it! Given a pair of numbers to be added, just mechanically perform the steps in the algorithm to obtain the sum. It is this demand of not requiring any intelligence that makes computing machines possible. More important: it *defines* what computation is!

Let me illustrate the idea of an algorithm more sharply. Consider adding two natural numbers<sup>3</sup>. The *process* of addition generates a third natural number given a pair of them. A simple way to *mechanically* perform addition is to tabulate all the pairs and their sum, i.e. a table of triplets of natural number with the first two being the numbers to be added and the third their sum. Of course, this table is infinite and the tabulation process cannot be completed. But for the purposes of *mechanical* - i.e. without "intelligence" - addition, the tabulation idea can work except for the inability to "finish" tabulation. What we would really like to have is some kind of a "black box machine" to which we "give" the two numbers to be added, and "out" comes their sum. The kind of operations that such a box would essentially contain is given by the addition *algorithm* above: for integers represented in the decimal form, start from the least significant place, add the corresponding digits and carry forward to the next place if needed, for all the digits, to obtain the sum. Notice that the "algorithm" is *not* limited by issues like our inability to finish the table. Any natural number, howsoever large, is represented by a finite number of digits and the algorithm will eventually stop! Further, the algorithm is not particularly concerned about the pair of numbers that it receives to be processed. For any, and every, pair of natural numbers it works. The algorithm captures the computation process of addition, while the tabulation does *not*. The addition algorithm that we have presented, is however intimately tied to the *representation scheme* used to write the natural numbers. Try the algorithm for a

Roman representation of the natural numbers!

We now have an intuitive feel of what computation seems to be. Since the 1920s Mathematics has concerned itself with the task of clearly understanding what computation is. Many models have been developed, and are being developed, that try to sharpen our understanding. In this module we will concern ourselves with four different approaches to modeling the idea of computation. The following sections, we will try to intuitively motivate them. Our approach is necessarily introductory and we leave a lot to be done. The approaches are:

1. The  $\lambda$  Calculus,
2. The theory of Partial Recursive Functions,
3. Markov Algorithms, and
4. Turing Machines.

## 3 The $\lambda$ Calculus

This is the first systematic attempt to understand Computation. Historically, the issue was what was meant by an algorithm. A logician, Alonzo Church, created the  $\lambda$  calculus in order to understand the nature of an algorithm. To get a feel of the approach, let us consider a very simple "activity" that we perform so routinely that we almost forget it's algorithmic nature - counting.

An algorithm, or synonymously - a computation, would need some object to work upon. Let us call it  $x$ . In other words, we need an ability to *name* an object. The algorithm would transform this object into something (possibly itself too). This transformation would be the actual "operational details" of the algorithm "black box". Let us call the resultant object  $y$ . That there is some rule that transforms  $x$

to  $y$  is written as:  $\lambda x \rightarrow y$ . Note that we concentrate on the process of transforming  $x$  to  $y$ , and

we have merely created a notation of expressing a transformation. Observe that this transformation process itself is another object, and hence can be named! For example, if the transformation generates the square of the number to which it is applied, then we *name*

the transformation as: *square*. We write this as:  $square : \lambda x \rightarrow x^2$ . The final ability that an

algorithm needs is that of its transformation, named  $f$  being applied on the object named  $x$ . This

is written as  $(f x)$ . Thus when we want to square a natural number  $a$ , we write it as  $(square a)$ .

An algorithm is characterized by three abilities:

1. Object naming; technically the named object is called as a *variable*,
2. Transformation specification, technically known as *abstraction*, and
3. Transformation application, technically known as *application*.

These three abilities are technically called as  $\lambda$  terms.

The addition process example can be used to illustrate the use of the above syntax of  $\lambda$  calculus through the following remarks. (To relate better, we *name* variables with more than one letter words enclosed in single quotes; each such multi-letter name should be treated as one single symbol!)

1. 'add', 'x', 'y' and '1' are variables (in the  $\lambda$  calculus sense).
2.  $\lambda x \rightarrow \lambda y \rightarrow x + y$  is the "addition process" of *bound variables*  $x$  and  $y$ . The bound variables "hold the place" in the transformation to be performed. They will be replaced by the actual numbers to be added when the addition process gets "applied" to them - See remark 1. Also the process specification has been done using the usual laws of arithmetic, hence  $x + y$  on the right hand side<sup>4</sup>.
3.  $((\lambda x \rightarrow \lambda y \rightarrow x + y) 1)$  is the application of the abstraction in remark 1 to the  $\lambda$  term 1.

An *application* means replacing every occurrence of the first bound variable, if any, in the body of the term to be applied (the left term) by the term being applied to (the right term).  $x$  being the first bound variable, its every occurrence in the body  $(x + y)$  is replaced by  $1$  due to the application.

This gives us the  $\lambda$  term:  $\lambda y \rightarrow 1 + y$ , i.e. a process that can add the value 1 to its input as "signalled" by the bound variable that "holds the place" in the processing.

4. We usually name  $\lambda y \rightarrow 1 + y$  as 'inc' or '1+'.

### 3.1 Conversions:

We have acquired the ability to express the essential features of an algorithm. However, it still remains to capture the *effect* of the computation that a given algorithmic process embodies. A process involves replacing one set of symbols corresponding to the input with another set of symbols corresponding to the output. Symbol replacement is the *essence* of computing. We now present the "manipulation" rules of the  $\lambda$  calculus called the *conversion* rules.

We first establish a notation to express the act of *substituting* a variable  $x$  in an expression  $E$  by another variable  $y$  to obtain a new expression  $E'$  as:  $E' \equiv E[x/y]$  ( $E'$  is  $E$  whose every  $x$  is *replaced* by  $y$ ). Since the  $\lambda x \rightarrow E$  specifies the binding of a variable  $x$  in  $E$ , it follows that  $x$  must occur *free* in  $E$ . Further, if  $y$  occurs free in  $E$  then this state of  $y$  must be preserved after substitution - the  $y$  in  $E$  and the  $y$  that would be substituting  $x$  are different! Hence we must demand that if  $y$  is to be used to substitute  $x$  in  $E$  then it must **not** occur free in  $E$ . And finally, if  $y$  occurs bound in  $E$  then this state of  $y$  too must be preserved after substitution. We must therefore have that  $y$  must not occur bound in  $E$ . In other words, the variable  $y$  does not occur (neither free nor bound) in expression  $E$ .

The conversions are:

$\alpha$

Since a bound variable in a  $\lambda$  expression is simply a place holder, all that is required is that unique place holders be used to designate the correct places of each bound variable in the expression. As long as the uniqueness is preserved, it does *not* matter what name is actually used to refer to their respective places<sup>5</sup>. This freedom to associate any name to a bound variable is expressed by the  $\alpha$  conversion rule which states the equivalence of  $\lambda$  expressions whose bound variables have been merely renamed. The renaming of a bound variable  $x$  in an expression  $\lambda x \rightarrow E$  to a variable  $y$  that does not occur in  $E$  is the  $\alpha$  conversion:

$\alpha$  conversion: lff  $y$  does not occur in  $E$ ,

$$\lambda x \rightarrow x \equiv \lambda y \rightarrow E[x/y](1)$$

As a consequence of  $\alpha$  conversion, it is possible for us to substitute while avoiding accidental change in the nature of occurrences.  $\alpha$  conversion is necessary to maintain the equivalence of the expressions before and after the substitution.

$\beta$

This is the heart of capturing computation in the  $\lambda$  calculus style as this conversion expresses the exact symbol replacement that computation essentially consists of. We observe that an application represents the *action* of an abstraction - the computational process - on some "target" object. Thus as a result of application, the "target" symbol must replace every occurrence of the bound variable in the abstraction that is being applied on it. This is the  $\beta$  conversion rule expressed using substitution as:

$\beta$  conversion: lff  $y$  does not occur in  $E$ ,

$$((\lambda x \rightarrow E) y) \equiv E[x/y](3)$$

Since computation essentially *is* symbol replacement, "executing an algorithm on an input" is expressed in the  $\lambda$  calculus as "performing  $\beta$  conversions on applications until no more conversion is possible". The expression obtained when no more conversions are possible is the "output" or the "answer".

For example, suppose we wish to apply the  $\lambda$  expression  $(\lambda x \rightarrow \lambda y \rightarrow x + y)$  to  $y$ , i.e.

$(\lambda x \rightarrow \lambda y \rightarrow x + y) y$ . But  $y$  already occurs bound in the old expression. Thus we first

rename the  $y$  in the old expression to (say)  $z$  using  $\alpha$  conversion to get:

$((\lambda x \rightarrow \lambda z \rightarrow x + z) y)$  and then substitute every  $x$  by  $y$  using  $\beta$  conversion.

$\eta$

It expresses the fact that the expression that is free of any occurrences of the binding variable in a  $\lambda$  abstraction is the expression itself. Thus:

$\eta$  conversion: lff  $x$  does not occur in  $E$ , then

$$(\lambda x \rightarrow E x) \equiv E(5)$$

If a  $\lambda$  expression  $E$  is transformed to an expression  $E'$  by the application of any of the above conversion rules, we say that  $E$  *reduces* to  $E'$  and denote it as  $E \rightarrow E'$ . If no more conversion rules are applicable to an expression  $E$ , then it is said to be in its *normal form*. An expression  $E$  to which a conversion is applicable is referred to as the corresponding *redex* (*reducible expression*). Thus we speak of  $\beta$  redex,  $\eta$  redex etc.

## 3.2 The $\lambda$ calculus in use

### 3.2.1 The Natural Numbers in $\lambda$ calculus

Natural numbers are the set  $\mathcal{N} = \{0, 1, 2, \dots\}$ . We "know" them as a set of values. However, we need to look at their behavioral properties to see their computational nature. We demonstrate this using the *counting* process. We associate a natural number with the instances of counting that are being applied to the object being counted. For instance, if the counting *process* is applied "zero" times to the object (i.e. the object does not exist for the purposes of being counted), then we have the specification, i.e. a  $\lambda$  term, for the natural number "zero". If the counting process is applicable to the object just once (i.e. there is only one instance of the object), then the function for that process represents the natural number "one", and so on. Let us *name* the counting process by the symbol  $f$ . If  $x$  is the object that is being counted, then this motivates a  $\lambda$  term for a "zero" as<sup>6</sup>:

$$\underline{0}: \lambda f, x \rightarrow x(7)$$

where the  $x$  remains as it is in our thoughts, but **no** counting has been applied to it. Hence  $x$  forms the body of the  $\lambda$  abstraction. A ``one'', a ``two'', or ``three'' are defined as:

$$\underline{1}: \lambda f, x \rightarrow (f x)(9)$$

$$\underline{2}: \lambda f, x \rightarrow (f (f x))(10)$$

$$\underline{3}: \lambda f, x \rightarrow (f (f (f x)))(11)$$

$$\underline{n}: \lambda f, x \rightarrow (f \dots (f x))(10)$$

A look at Eqns. (□□) shows that a natural number is given by the number of occurrences of the application of  $f$  - our name for the counting process.

At this point, let us pause for a moment and compare this way of thinking about numbers with the ``conventional'' way. Conventionally, we tend to associate numbers with objects rather than the process. Contrast: ``I counted *ten* tables'' with ``I could apply counting *ten* times to objects that were tables''. In the first case, ``ten'' is associated subconsciously to ``table'', while in the second case it is associated with the ``counting'' process! We are accustomed to the former way of looking at numbers, but there is no reason to **not** do it the second way.

And finally, to present the power of pure symbolic manipulation, we observe that although we have motivated the above  $\lambda$  expressions of the natural numbers as a result of applying the counting process  $f$ , *any* process that can be sensibly applied to an object can be used in place of  $f$ .

For example, if  $f$  were the process that generates the double of a number, then the above  $\lambda$  expressions could be used to generate the even numbers by a simple ``application'' of  $f$  once (i.e.  $\underline{1}$ ) to get the first even number, twice (i.e.  $\underline{2}$ ) to get the second even number etc. We have simply used  $f$  to denote the counting process to get a feel of how the  $\lambda$  expressions above make sense. A natural number  $n$  is just  $n$  applications of (some)  $f$  to  $x$ , i.e.  $\underline{n} = (n f x)$ .

We now present the addition process<sup>7</sup> from this  $\lambda$  calculus view. The addition of two natural numbers  $m$  and  $n$  is simply the total number of *applications* of the counting process. To get the  $\lambda$  expression that captures the addition process, we observe that the sum of  $m$  and  $n$  is just  $m$  further applications of the counting process  $f$  to  $n$  which has already been generated by using  $(n f x)$ . Hence addition can be defined as:

$$\text{add}: \lambda m, n, f, x \rightarrow ((m f) (n f x))(11)$$

Note that in Eq. (□), the  $\lambda$  expression  $(m f)$  is *applied* to the  $\lambda$  expression  $(n f x)$ . Consider adding  $\underline{1}$  and  $\underline{2}$ :

$$\begin{aligned}
\text{add } \underline{1} \ \underline{2} &\equiv ((\lambda m, n, f, x \rightarrow ((m \ f) \ (n \ f \ x))) \ \underline{1} \ \underline{2}) \\
&\langle \beta \text{ reduce once} \rangle \\
&\equiv ((\lambda n, f, x \rightarrow ((\underline{1} \ f) \ (n \ f \ x))) \ \underline{2}) \\
&\langle \beta \text{ reduce again} \rangle \\
&\equiv (\lambda f, x \rightarrow ((\underline{1} \ f) \ (\underline{2} \ f \ x))) \\
&\langle \text{Substitute definition of } \underline{1} \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda f, x \rightarrow (f \ x) \ f) \ (\underline{2} \ f \ x))) \\
&\langle \alpha \text{ convert (rename inner } f \text{ to } c) \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda c, x \rightarrow (c \ x) \ f) \ (\underline{2} \ f \ x))) \\
&\langle \text{Apply} \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda x \rightarrow (f \ x)) \ (\underline{2} \ f \ x))) \\
&\langle \text{Substitute definition of } \underline{2} \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda x \rightarrow (f \ x)) \ (\lambda f, x \rightarrow (f \ (f \ x)) \ f \ x))) \\
&\langle \alpha \text{ convert (rename inner } f \text{ to } c) \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda x \rightarrow (f \ x)) \ (\lambda c, x \rightarrow (c \ (c \ x)) \ f \ x))) \\
&\langle \alpha \text{ convert (rename inner } x \text{ to } y) \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda x \rightarrow (f \ x)) \ (\lambda c, y \rightarrow (c \ (c \ y)) \ f \ x))) \\
&\langle \text{Apply once} \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda x \rightarrow (f \ x)) \ (\lambda y \rightarrow (f \ (f \ y)) \ f \ x))) \\
&\langle \text{Apply again} \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda x \rightarrow (f \ x)) \ (f \ (f \ x)))) \\
&\langle \alpha \text{ convert (rename inner } x \text{ to } y) \rangle \\
&\equiv (\lambda f, x \rightarrow ((\lambda y \rightarrow (f \ y)) \ (f \ (f \ x)))) \\
&\langle \beta \text{ convert} \rangle \\
&\equiv (\lambda f, x \rightarrow (f \ (f \ (f \ x)))) \\
&\langle \text{By definition} \rangle \\
&\equiv \underline{3}(13)
\end{aligned}$$

The `add`  $\lambda$  expression takes the  $\lambda$  expression form of two natural numbers  $m$  and  $n$  to be added

and yields a  $\lambda$  expression that behaves exactly as the sum of these two numbers. Note that this resulting  $\lambda$  expression expects two arguments namely  $f$  and  $x$  to be supplied when it is to be

applied. The  $\lambda$  expression that we write in the  $\lambda$  calculus are simply some process specifications including of those objects that we formerly thought of as "values".

This view of looking at computation from the "processes" point of view is referred to as the *functional paradigm* and this style of programming is called *functional programming*. Programming languages like Lisp, Scheme, ML and Haskell are based on this kind of view of programming - i.e. expressing "algorithms" as  $\lambda$  expression. In fact, Scheme is often viewed as " $\lambda$  calculus on a computer". For instance, we associate a name "square" to the operation "multiply  $x$  (some object) by itself" as `(define square (lambda (x) (* x x)))`. In our  $\lambda$  calculus notation, this would look like  $square : \lambda x \rightarrow (mul\ x\ x)$ .

### 3.2.2 The Booleans

Conventionally, we have two "values" of the boolean type: `True` and `False`. We also have the conventional boolean "functions" like `NOT`, `AND` and `OR`. From a purely formal point of view, `True` and `False` are merely symbols; one and only one of each is returned as the "result"/"value" of a boolean expression (which we would like to view as a  $\lambda$  expression). Therefore, a (simple!) encoding of these values is through the following two  $\lambda$  abstractions:

$$\begin{aligned} \underline{\text{True}} &: \lambda x, y \rightarrow x & (13) \\ \underline{\text{False}} &: \lambda x, y \rightarrow y & (14) \end{aligned}$$

Note that Eqn.(13) is an abstraction that *encodes* the behavior of the value `True` and is thus a very *computational* view of the value<sup>8</sup>. Similarly Eqn.(14) is an abstraction that *encodes* the behavior of the value `False`. Since the encodings represent the selection of mutually "opposite" expressions from the two that would be given by a particular (function) application, we can say that the above equations indeed capture the *behaviors* of these "values" as "functions". This is also evident when we examine the  $\lambda$  abstraction for (say) the `IF` boolean function and apply it to each of the above equations. The `IF` function behaves as: given a boolean expression and two  $\lambda$  terms, return the first  $\lambda$  term if the expression is "`True`" else return the second  $\lambda$  term. As a  $\lambda$  abstraction it is expressed as:

$$\text{IF} : \lambda c, x, y \rightarrow c\ x\ y \quad (14)$$

i.e. *apply* the boolean expression  $c$  to  $x$  and  $y$ . If  $c$  is `True` (i.e.  $c$  *reduces* to the  $\lambda$  term `True`),

then we must get  $x$  as a result of applying various conversions to Eqn.(13), else we must get  $y$ .

The `AND` boolean function behaves as: "If  $p$  then  $q$  else false". Accordingly, it can be encoded as the following  $\lambda$  abstraction:

$$\begin{aligned} \text{AND} &: \lambda p, q \rightarrow (((\lambda c, x, y \rightarrow c\ x\ y)\ p)\ q)\ \underline{\text{False}} \\ &\langle \text{IF applied to } p\ q\ \text{and } \underline{\text{False}} \rangle \\ &\rightarrow p\ q\ \underline{\text{False}} \\ &\langle \beta \text{ reductions} \rangle \\ &\rightarrow p\ q\ (\lambda x, y \rightarrow y) \quad (16) \\ &\langle \text{Definition of } \underline{\text{False}} \text{ in Eqn.}(??) \rangle \end{aligned}$$

Note that in Eqn.(1) further reductions depend on the actual forms of  $P$  and  $Q$ . To see that the

$\lambda$  abstractions indeed behave as our "usual" boolean functions and values, two approaches are possible. Either work out the reductions in detail for the complete truth tables of both the boolean functions, or noting the *behavioral* properties of these functions and the "values" that they could take, (intuitively ?) reason out the behavior. I will try the latter technique. Consider the AND function

defined by Eqn.(2). It takes two arguments  $P$  and  $Q$ . If we *apply* it to Eqn.(1) and Eqn.(1) (i.e.

AND TRUE FALSE), then the reduction would substitute Eqn.(1) for every occurrence of  $P$  and Eqn.(

1) for every occurrence of  $Q$  in Eqn.(2). This gives us a  $\lambda$  abstraction to which have been *applied*

two arguments, namely  $Q$  and  $\lambda x, y \rightarrow y$ ! This abstraction *behaves* like TRUE and hence it yields

it's first argument as the result. That is, a reduction of this  $\lambda$  abstraction yields  $\lambda x, y \rightarrow y$ , i.e.  $Q$  -

the expected output. Note that no further reductions are possible.

### 3.3 Few Important Theorems

---

At this point, we would like to mention that the  $\lambda$  calculus is extensively used to mathematically model and study computer programming languages. Very exciting and significant developments have occurred, and are occurring, in this field.

**Theorem 1** *A function is representable in the  $\lambda$  calculus if and only if it is Turing computable.*

**Theorem 2** *If  $E_1 = E_2$  then there exists  $E$  such that  $E_1 \rightarrow E$  and  $E_2 \rightarrow E$ .*

**Theorem 3** *If an expression  $E$  has a normal form, then repeatedly reducing the leftmost  $\beta$  or  $\eta$  redex - with any required  $\alpha$  conversion, will terminate in the normal form.*

### 3.4 Worked Examples

---

We apply the IF  $\lambda$  expression to TRUE i.e. we work out an application of Eqn.(1) to Eqn.(1):

$$\begin{aligned}
& ((\lambda c, x, y \rightarrow c \ x \ y) \ \underline{\text{True}}) \equiv ((\lambda c, x, y \rightarrow c \ x \ y) \ \underline{\text{True}}) \\
& \quad \langle \beta \text{ reduction} \rangle \\
& \equiv (\lambda x, y \rightarrow \underline{\text{True}} \ x \ y) \\
& \quad \langle \text{Substitute Eqn. (??), get a Application} \rangle \\
& \equiv (\lambda x, y \rightarrow (\lambda x, y \rightarrow x) \ x \ y) \\
& \quad \langle \alpha \text{ reduction, twice} \rangle \\
& \equiv (\lambda x, y \rightarrow ((\lambda a, b \rightarrow a) \ x \ y)) \\
& \quad \langle \beta \text{ reduction} \rangle \\
& \equiv (\lambda x, y \rightarrow ((\lambda b \rightarrow x) \ y)) \\
& \quad \langle \beta \text{ reduction} \rangle \\
& \equiv (\lambda x, y \rightarrow x) \\
& \equiv \underline{\text{True}}
\end{aligned}$$

which will return the first object of the two to which the  $\text{IF}$  will actually get applied to (i.

e.  $((\lambda c, x, y \rightarrow c \ x \ y) \ \underline{\text{True}} \ a \ b)$ ). Note that Eqns. (□, □) capture the behavior of the objects

that we are accustomed to see as "values". I cannot stress more that the "valueness" of these objects is not at all relevant to us from the  $\lambda$  calculus point of view. The "valueness" cannot be captured as a "computational" process while the behavior can be. And if the behavior of the computational process is in every way identical to the value, there is little reason to impose any differentiation of the object as a "value" or as a "function". On the other hand, insisting on the "valueness" of the objects given by those equations forces us to invent unique symbols to be permanently bound to them. I also believe that it makes the essentially computational nature of these objects opaque to us.

Let me also illustrate the construction of the *succ* function that yields the successor of the number given to it. This function will be used in the Partial Recursive Functions model. The *succ* process is one more application of the counting process to the given natural number. We recall the definitions of natural numbers from Eqns. (□, □). We observe that the natural number is defined by the number of applications of the counting process  $f$  to some object  $x$ . Hence the bodies of the corresponding

$\lambda$  expressions involve application of  $f$  to  $x$ . This gives us a way to define the *succ* as an application

of the process  $f$  to  $n$ , the given natural number. This is what was used to define the *add* process in Eqn.

(□). Thus:

$$\text{succ: } \lambda n, f, x \rightarrow (n \ f) \ (f \ x) \quad (16)$$

### 3.5 Exercises

- Construct the  $\lambda$  expression for the following:
  - The **OR** boolean function which behaves as "If p then true else q".
  - The **NOT** boolean function which behaves as "If p then False else True".

2. Evaluate, i.e. perform necessary conversions of the following  $\lambda$  expression.

1. (IF FALSE)
2. (AND TRUE FALSE)
3. (OR TRUE FALSE)
4. (NOT TRUE)
5. (succ 1)

## 4 The theory of Partial Recursive Functions

---

We now introduce ourselves to another model that studies computation. This model appears to be most mathematical of all. However, in fact **all** the models are equally mathematical and **exactly** equivalent to each other. This approach was pioneered by the logician Kurt Gödel and almost immediately followed  $\lambda$  calculus. Our purpose of introducing this view of computation is much more philosophical than any practical one that can directly be used in day to day software practice<sup>9</sup>. We would like to give a flavor of the questions that are asked for developing the theory of computation further. In this module, we will **not** concern ourselves about the developments that *are* occurring in this rich field, but we will give an idea of *how* the developments occur by giving a sample of questions (some of which have already been answered) that are asked.

To capture the idea of computation, the theory of Partial Recursive Functions asks: Can we view a computational process as being generated by combining a few basic processes? It therefore tries to identify the basic processes, called the *initial functions*. It then goes on to identify combining techniques, called *operators* that can generate new processes from the basic ones. The choice of the initial functions and the operators is quite arbitrary and we have our first set of questions that can develop the theory of computation further. For example,

- Is the choice of the initial functions unique?
- Similarly, is the choice of the operators unique?
- If different initial functions or operators are chosen will we have a more restricted theory of computation or a more general theory of computation?

### 4.1 Basic Concepts and Definitions

---

We define the initial functions and the operators over the set  $\mathcal{N}$  of natural numbers<sup>10</sup>.

#### Initial Functions

**Definition 1** the *succ* function,

**Definition 2** the  $k$ -ary constant-0 function,  $C_0^k$  for  $k \geq 0$ ,  $k \in \mathcal{N}$ , i.e.  $C_0^0$ ,  $C_0^1$ ,  $C_0^2$ ,  $\dots$

The superscript  $k$  in  $C_0^k$  denotes the *number* of arguments that the constant function takes and the subscript is the *value* of the function, 0 in this case. Thus given  $k$  natural numbers  $a_1, a_2, \dots, a_k$ , we have

$$C_0^k \equiv C_0(a_1, a_2, \dots, a_k) = 0$$

**Definition 3** the projection functions,  $p_j^k$  for  $k \geq 1$  and  $1 \leq j \leq k$ , i.e.  $p_1^1$ ,  $p_1^2$ ,  $p_2^2$ ,  $p_1^3$ ,  $p_2^3$ ,

$p_3^3$ ,  $p_1^4$ ,  $\dots$ . The superscript is the number of arguments of the particular projection function and

the subscript is the argument to which the particular projection function projects. Thus given  $k$  natural numbers  $a_1, a_2, \dots, a_k$ , we have

$$p_j^k \equiv p_j(a_1, a_2, \dots, a_k) = a_j$$

## Function forming Operators

### Definition 4 Generalized Function Composition:

Given:  $f : \mathcal{N}^m \rightarrow \mathcal{N}, m \geq 1$ , and  $g_1, g_2, \dots, g_m$  each  $\mathcal{N}^k \rightarrow \mathcal{N}, k \geq 0$ ,

Then: a new function  $h : \mathcal{N}^k \rightarrow \mathcal{N}$  is obtained by the schema:

$$h(n_1, n_2, \dots, n_k) = f( \begin{array}{l} g_1(n_1, n_2, \dots, n_k), \\ g_2(n_1, n_2, \dots, n_k), \\ \dots, \\ g_m(n_1, n_2, \dots, n_k) \end{array} )$$

Sometimes the notation  $\vec{n}$  is used for  $(n_1, n_2, \dots, n_k)$  giving a more compact  $h(\vec{n}) = f(g_1(\vec{n}), g_2(\vec{n}), \dots, g_m(\vec{n}))$ . This schema is called function composition and is denoted as **Comp**. Thus  $h = \mathbf{Comp}[f, g_1, g_2, \dots, g_m]$ . When this schema is applied to a set of arguments  $\vec{n}$ , we have

$$\mathbf{Comp}[f, g_1, g_2, \dots, g_m](\vec{n}) = f(g_1(\vec{n}), g_2(\vec{n}), \dots, g_m(\vec{n}))$$

### Definition 5 Primitive Recursion:

Given:  $f : \mathcal{N}^k \rightarrow \mathcal{N}, k \geq 0$  and  $g : \mathcal{N}^{k+2} \rightarrow \mathcal{N}$ ,

Then: a new function  $h : \mathcal{N}^{k+1} \rightarrow \mathcal{N}$  is obtained by the schema

1. Base case:  $h(\vec{n}, 0) = f(\vec{n})$
2. Inductive case:  $h(\vec{n}, m + 1) = g(\vec{n}, m, h(\vec{n}, m))$

This schema is called primitive recursion and is denoted by **Pr**. Thus  $h = \mathbf{Pr}[f, g]$ .

### Definition 6 Minimization:

Given:  $f : \mathcal{N}^{k+1} \rightarrow \mathcal{N}, k \geq 0$

Then: a new function  $g : \mathcal{N}^k \rightarrow \mathcal{N}$  is obtained by the schema

$$g(\vec{n}) = \mu m [f(\vec{n}, m) = 0],$$

such that  $\forall k < m, f(\vec{n}, j)$  is defined and  $\neq 0$ . This schema is denoted by **Mn**[ $f$ ] and the notation  $\mu m [C(\vec{n}, m)]$  is the least natural number  $m$  such that  $C(\vec{n}, m)$  holds; we vary  $m$  for a given "fixed"  $\vec{n}$  and look out for the least of those  $m$  for which the  $(k+1)$ -ary predicate  $C(\vec{n}, m)$  holds. We also write

$$\mu m[f(\vec{n}, m) = 0]$$

to mean the least number  $m$  such that  $f(\vec{n}, m)$  is 0. The  $\mu$  is referred to as the least number operator.

The set of functions obtained by the use of all the operators *except* the minimization operator, on the initial functions is called the set of *primitive recursive functions*. The set of functions obtained by the use of three operators on the initial functions is called the set of *partially recursive functions* or  $\mu$  *recursive functions*.

#### 4.1.0.1 Remarks on Minimization:

We are trying to develop a mathematical model of the *intuitive* idea of computation. The initial functions and the function forming operations that we have defined until minimization guarantee a value for every input combination<sup>11</sup>. However, there are computable processes that may **not** have values for some of the inputs, for instance division. We have not been able to capture the aspect of computation where results are available partially. The minimization schema is an attempt to capture this intuitive behaviour of computation - that sometimes we may have to deal with computational processes that may not *always* have a defined result.

Note that  $f$  has the property that an  $m$  exists for every  $\vec{n}$ , then  $g$  is computable; given  $\vec{n}$ , we need to simply evaluate  $f(\vec{n}, 0), f(\vec{n}, 1), \dots$  until we find an  $m$  such that  $f(\vec{n}, m) = 0$ . Such an  $f$  is said to be *regular*.

Now note that given some function we can check that it is primitive recursive. The next natural question is: can we check that it is  $\mu$  recursive too? But being  $\mu$  recursive means that the minimization has

been done over regular functions. After checking that the function is primitive recursive, we must further check if the minimization has been done over regular functions. "Checking" essentially means that for our "candidate" function, we determine if it is a regular function or not, i.e. if an  $m$  exists for every  $\vec{n}$ . Conceptually, we can list out all the regular functions and then compare the given function with each member of the list. Suppose all the possible regular functions are listed as  $f_1^k, f_2^k, \dots$

which means for every  $f_i^k$  for monotonically increasing  $k$  and  $i, k \in \mathcal{N}$  there is an  $m_i^k$  for  $\vec{n}_i^k$

such that  $f_i^k(\vec{n}_i^k, m_i^k) = 0$ . Since  $k$  is monotonically increasing, the  $f_i^k$ 's are ordered.

Consider the set of functions when  $k = 1$ . These are all the regular functions that take 2 (1 + 1) arguments. That is the set  $f_1(n, m_1), f_2(n, m_2), \dots$  etc. A simple way to construct a

computable function that is **not** regular is to have  $f_i(n, m_i) = 1$  for the corresponding regular

function  $f_i(n, m_i) = 0$ . We can, therefore, always construct a function that is computable in the intuitive sense, but will *not* be a member of the list. Notice that this construction is based on ensuring that whatever  $m$  existed earlier, we simply make it non-existent! We can surely have computable functions for which the  $m$  may not exist even if  $f$  were well defined everywhere.

Our ability to construct such a computable function is based on the assumption that we can form a list of regular functions. This ability to construct the list was required to determine - check - if a given function is regular! Accepting this assumption to be true would mean that we are still dealing with only well formed functions and an aspect of the notion of computability is not being taken into account. However, by not assuming an ability to determine the regular nature of a function, we *can* bring this aspect of computability into our mathematical structure. If we want an exhaustive system for representing all the computable functions, then we either have to give up the idea that only well defined functions will be represented or we must accept that the class of computable functions that will

not be completely representable - i.e. they may be partial! Note that the inability to determine if  $f$

is regular makes  $g$  a partial function since the *least*  $m$  may not necessarily exist and hence  $g$  could

be undefined even if  $f$  is defined! In the interest of having an exhaustive system, we make the

latter choice that the regular functions would not be listed<sup>12</sup>.

## End Remarks

By the way, a different choice of initial functions as: *equal*, *succ* and *zero* and the operators as: *Comp* and *Conditional* have the same power as the above formulation. This choice is due to John McCarthy and is the basis of the LisP programming language along with the  $\lambda$  calculus. The *Conditional*, which is the same as the  $\text{IF}$  in the  $\lambda$  calculus, can do both: primitive recursion and minimalization.

## 4.2 Important Theorems

---

**Theorem 4** Every Primitive Recursive function is total<sup>13</sup>.

**Theorem 5** There exists a computable function that is total but not primitive recursive.

**Theorem 6** A number theoretic function is partial recursive if and only if it is Turing computable.

## 4.3 More Issues in Computation Theory

---

The remarks on minimization in section (□) give rise to a number of questions. In particular, they point out to the possibility that there may be some processes that are uncomputable - we cannot have an algorithm to do the job. For instance, the regular functions cannot be listed<sup>14</sup>.

### 4.3.1 What can and cannot be computed

It can be argued in many ways that there are some problems which cannot have an algorithm, i.e. they cannot be computed. For instance, note that the partial recursive functions model of computation uses natural numbers as the basis set over which computation is defined. Theorem □ demonstrates that this model of computation is *exactly equivalent* to the Turing model (to be introduced later). Alternately, consider the  $\lambda$  calculus model where Church numerals have been defined by explicitly invoking the counting method over: natural numbers again! Moreover, it is also (hopefully) evident that the process  $f$  that is used to refer to counting can actually be replaced by any procedure that operates over natural numbers, for example the "doubling" procedure. We also know by Theorem □ that this model of computability is equivalent to the Turing model! Hence it is also equivalent to the partial recursive functions perspective! Thus it appears that natural numbers and operations over them are the basic "primitives" of computation. The counting arguments extend to the set of rational numbers which are said to be *countable*, but *infinite* since they can be placed in a 1-1 correspondence with the set of natural numbers. However, when irrationals are introduced into the system, we are unable to use the counting arguments to come up with a "new"/"better" model of computation! This means that the current model of computation is unable to deal with processes that operate over irrationals, reals and so on. For instance, the limit of a sequence **cannot** be computed, i.e. there is no mechanical procedure (an algorithm) that we can use to compute the limit of a given sequence<sup>15</sup>.

In general, the observations of the above paragraphs lead us to the fact that: there are processes which are **not** expressible as algorithms - i.e. they cannot be computed! To make things more difficult, the equivalences between the different perspectives of computation prompted Church and Turing to hypothesize<sup>16</sup> that: *Any model of computation cannot exceed the Turing model in power*. In other words, we may not have a better model of computation. As yet this hypothesis has neither been mathematically proven, nor have we been able to come up with a better computation model!

### 4.3.2 The Halting Problem

The classic demonstration of the fact that there are some processes which cannot have an algorithm comes in the form of the *Halting* problem. The problem is: Can we conceive an algorithm that can tell us whether or not a given algorithm will terminate? The answer is: **NO**. The argument that there cannot exist such an algorithm goes as: If there indeed were such an algorithm, say  $H$  (halt), then we could construct a process, say  $U$  (unhalt), that would use this algorithm as follows: If an algorithm  $A$  is certified to terminate by  $H$ , then  $U$  loop infinitely, else  $U$  would itself halt. Now if we use  $U$  on  $H$  itself, the situation becomes:  $U$  halts if  $H$  does not and  $U$  does not halt if  $H$  does. Finally, now if  $H$  is asked to tell us if  $U$  halts we land up in the following scenario:  $H$  would

halt only if  $H$  would not halt (since  $U$  makes a "crooked" use of  $H$ ) and  $H$  would not halt if  $H$  halts. This contradiction can only be resolved if  $H$  does **not** exist - i.e. there can not exist an algorithm that can certify if a given algorithm halts or not.

The Halting problem demonstrates that we can imagine processes, but that does not mean that we can have an algorithm for them. The theory of partial recursive functions isolates this peculiar characteristic in the minimization operator **Mn**. Notice that the operator is defined using an existential process - i.e. we are required to find the *least*  $m$  amongst all possible  $f(\vec{n}, m) = 0$  for

a given  $f$ . This  $m$  may or may not exist! The initial functions and other operators, **Comp** and **Pr** do not have such a peculiar characteristic! We refer to those problems for which an algorithm can be conceived as being *decidable*. Notice that the primitive recursive functions - the initial functions with the **Comp** and **Pr** operators - are decidable. In contrast to other models, the theory of partial recursive functions isolates the undecidability issue explicitly in the **Mn** operator. In situations when we need to be concerned of the solvability of the problem, it might help to examine the consequences of the **Mn** operator. Other models, though equivalent, may not prove to be so focussed. This illustrates that we can use the different models in appropriate situations to most simply solve the problem at hand.

## 4.4 Worked Examples

---

**Q.** Show that the addition function is primitive recursive.

**A.** We express the addition function over  $\mathcal{N}$  recursively as:

1.  $plus(n, 0) = n$
2.  $plus(n, m + 1) = succ(plus(n, m))$

Observing that:  $n = p_1^1(n)$  we can write  $plus(n, 0) = n$  as  $plus(n, 0) = p_1^1(n)$ . We

also express  $plus(n, m)$  as  $p_3^3(n, m, plus(n, m))$ . Therefore,

$$\begin{aligned} plus(n, m+1) &= succ(p_3^3(n, m, plus(n, m))) \\ &= Comp[succ, p_3^3](n, m, plus(n, m)) \end{aligned}$$

Hence we can write the recursive definition of addition as:

1.  $p_3^3(n, 0, plus(n, 0)) = p_1^1(n)$
2.  $p_3^3(n, m + 1, plus(n, m + 1)) = Comp[succ, p_3^3](n, m, plus(n, m))$

But this is the primitive recursion schema. Hence

$$\text{addition} : Pr[p_1^1, Comp[succ, p_3^3]]$$

## 4.5 Exercises

---

1. Given the recursive definition of multiplication as:
  1.  $mult(n, 0) = 0$ ,

2.  $\text{mult}(n, m+1) = \text{mult}(n, m) + n$ ,  
use the initial functions to show that multiplication is primitive recursive.
2. Show the the exponentiation function (over natural numbers) is primitive recursive.

## 5 Markov Algorithms

---

We now examine the third of our chosen approaches towards developing the idea of a computation - Markov Algorithms. The essence of this approach, first presented by A. Markov, is that a computation can be looked upon as a specification of the symbol replacements that must be done to obtain the desired result. This is based on the appreciation that a computation process, in it's raw essence replaces one symbol by another, and the specification is made in terms of rules - quite naturally called as *production rules* - that produce symbols<sup>17</sup>. We need to first introduce a number of concepts before we can show that Markov Algorithms can (and do) represent the computations of number theoretic functions. However, along the way we wish to show that the Markov Algorithm view of computation yields another interesting perspective: computation as string processing. We will just mention that a language called SNOBOL evolved from this perspective, although it is no longer much in use. However, languages like Perl - which are very much in use in practice, are excellent vehicles to study this approach and I believe that our abilities with Perl can be enhanced by the study of this approach.

### 5.1 The Basic Machinery

---

Let  $\Sigma = \{a, b, c, d\}$  be an alphabet (i.e. a set of (some) characters). By a *Markov Algorithm*

*Scheme* (MAS) or *schema* we mean a finite sequence of productions, i.e. rewrite rules. Consider a two member sequence of productions:

1.  $a \rightarrow c$
2.  $b \rightarrow \epsilon$

A *word* over  $\Sigma$  is any sequence, including the empty sequence  $\epsilon$ , of alphabets from  $\Sigma$ . The set of all words over  $\Sigma$  is called a *language* and typically denoted by  $L$  or  $\Sigma^*$ . Consider an input word,  $w = \text{``baba''}$ . Applying a production rule means substituting it's right hand side for the leftmost occurrence of it's left hand side in the input word. Thus, we apply rule  $\square$  to the input word  $w$  to get  $w_1$  as:

```
w = baba
= bcba    By rule ??
≡ w1
```

We keep on applying rule  $\square$  to  $w$  to obtain  $w_1, w_2, \dots$  until it no longer applies. Then we repeat the process using the next rule. Hence,

```
w1 = bcba
= bcbc    By rule ??
≡ w2
```

The production rule  $\square$  can no longer be applied to  $w_2$ . So we start applying the next rule starting from

the leftmost character of  $w_2$ .

$$\begin{aligned}
 w_2 &= bc bc \\
 &= \epsilon bc \quad \text{By rule ??} \\
 &= bc \quad \text{definition of } \epsilon - \text{empty string} \\
 &\equiv w_3
 \end{aligned}$$

The production rule  $\square$  is not applicable to  $w_3$ . We are required to attempt applying it, determine its inapplicability and continue to the next rule. Rule  $\square$  is applicable to  $w_3$ . Hence:

$$\begin{aligned}
 w_3 &= bc bc \\
 &= \epsilon \epsilon c \quad \text{By rule ??} \\
 &= cc \quad \text{definition of } \epsilon - \text{empty string} \\
 &\equiv w_4
 \end{aligned}$$

Neither rule  $\square$  nor  $\square$  can be applied to  $w_4$ . The substitution process *stops* at this point. The above

MAS has transformed the input "baba" to "cc". We write this as: "baba"  $\Rightarrow$  \* "cc"<sup>18</sup>. The general effect of the above MAS is to replace every occurrence of 'a' in the input by 'c' and to eliminate every occurrence of 'b' in the input. The table below illustrates its working for a few more input strings.

Input words	Transformed to
abcd	$\Rightarrow$ * ccd
bbbb	$\Rightarrow$ * $\epsilon$
cdcd	$\Rightarrow$ * cdcd
$\epsilon$	$\Rightarrow$ * $\epsilon$

The substitution process terminates if the attempt to apply the last production rule is unsuccessful. The string that remains is the *output* of the MAS. Note that the MAS *captures* a certain substitution process - that of replacing every 'a' by 'c' and eliminating 'b' (i.e. replacing every 'b' by  $\epsilon$ ). The process that the MAS captures is called as the **Markov Algorithm**. MASs are usually denoted by  $S$  and the corresponding Markov algorithms are denoted by  $AS$ . There is another way an MAS

can terminate for an input: we may have a rule whose application itself terminates the "substitution process"! Such a rule is expressed by having its right hand side start with a "." (dot) and is called as a *terminal production*. Note that for a given input, it is possible that a terminal production rule may **not** be applicable at all! We repeat: For a terminal production, the substitution process ceases immediately upon successful application even if the production could yet be applied to the resulting word.

In summary, a MAS is applied to an input string as: Start applying from the topmost rule to the string. Start from the leftmost substring in the string to find a match with the left hand side of the current production rule. If a match is found, then replace that substring with the right hand side of the production rule to obtain a new string which is given as the *next* input to the MAS (i.e. we start the process of applying the MAS again). If no substring matches the left hand side of the rule, continue to the next rule. If we encounter a terminal production, **or** if no left hand side matches are successful, then we terminate and the resulting string is the output of the MAS. Worked example  $\square$  illustrates the use of a terminal production.

### 5.1.0.1 Using Marker Symbols in MAS:

Sometimes it is useful to have special symbols like #, \$, %, ★ such as markers in addition to the alphabet  $\Sigma$ . However, the words that go as input and emerge at the output of an MAS always come from  $\Sigma$  and the markers only are used to express the substitutions that need to be performed. The set of markers that a particular MAS uses adds to the set  $\Sigma$  and forms the *work alphabet* that is denoted by  $\Gamma$ . Consider  $\Sigma = \{a, b\}$ ,  $\Gamma = \Sigma \cup \{\#\}$  and a MAS as

1.  $\#a \rightarrow a\#$
2.  $\#b \rightarrow b\#$
3.  $\# \rightarrow .ab$
4.  $\epsilon \rightarrow \#$

The MA corresponding to the above MAS appends "ab" to any string over  $\Sigma$ . Note that the output string of the above MAS is necessarily a word from  $\Sigma$ .

We now define a MAS formally.

**Definition 7 Markov Algorithm Schema:** A Markov Algorithm Schema  $S$  is any triple  $\langle \Sigma, \Gamma, \Pi \rangle$

where  $\Sigma$  is a non empty input alphabet,  $\Gamma$  is a finite work alphabet with  $\Sigma \subset \Gamma$  and  $\Pi$  is a finite ordered sequence of production rules either of the form  $\alpha \rightarrow \beta$  or of the form  $\alpha \rightarrow .\beta$  where both  $\alpha$  and  $\beta$  are possibly non empty words over  $\Gamma$ .

## 5.2 Markov Algorithms as Language Acceptors and Recognisers

This section mainly preparatory one for the "machine" view of computation that will be later useful when discussing Turing machines. Since by computation we mean a procedure that is so clearly mechanical that a machine can do it, we frequently use the word "machine" in place of an "algorithm".

**Definition 8** A machine<sup>19</sup> *accepts* a language  $L = \Sigma^*$  if

1. given an arbitrary word  $w \in L$ , the machine responds "affirmatively", and
2. if  $w \notin \Sigma^*$ , the machine does not respond affirmatively.

What will constitute an affirmative response must be separately stated in advance. A non affirmative response for  $w \notin L$  would mean that either that the machine is unable to respond, or it responds negatively. A machine that responds negatively for  $w \notin L$  is said to be a **recogniser**.

We conventionally denote a machine state that **accepts** a word by the symbol 1. The symbol 0 is used to denote the **rejection** state (for a language recogniser).

**Definition 9** Let  $S$  be a MAS with input alphabet  $\Sigma$  and a work alphabet  $\Gamma$  with  $1 \in (\Gamma - \Sigma)$ <sup>20</sup>. Then  $S$  *accepts* a word  $w$  if  $w \Rightarrow *1$ . If MAS accepts  $w$ , then  $A_S$  - the corresponding Markov Algorithm - also accepts  $w$ .

**Definition 10** A MAS  $S$  (as well as  $A_S$ ) *accept* a language  $L$  if  $S$  accepts all and only the words in  $L$ . Such an  $L$  is said to be **Markov acceptable** language.

**Definition 11** Let  $S$  be a MAS with input alphabet  $\Sigma$  and work alphabet  $\Gamma$  such that  $0, 1 \in \Gamma \setminus \Sigma$ . Then  $S$  recognises  $L$  over  $\Sigma$  if

1.  $\forall w \in L, w \Rightarrow_S *1$ , (accepting 1)
2.  $\forall w \notin L, w \Rightarrow_S *0$ , (rejecting 0)

If a MAS  $S$  recognises  $L$ , then  $A_S$  is also said to be recognise  $L$ .  $L$  is said to be Markov recognizable. Worked example  shows an MAS that accepts a language  $L = \{(ab)^n | n \in \mathcal{N} \geq 0\}$ . Worked example  shows an MAS that recognises a language  $L = \{(ab)^n | n \in \mathcal{N} \geq 0\}$ .

## 5.3 Number Theoretic Functions and Markov Algorithms

The MAS point of view of computation views computations as symbol transformations guided by production rules. To be applicable to a given domain the semantic entities in that domain must be symbolically represented. In other words, a symbolic representation scheme must be conceived to represent objects of the domain. To express number theoretic functions, we need to fix a scheme to represent a natural number using some symbols.

Let a natural number  $n$  be represented by a string of  $(n + 1)$  1s. Thus  $\Sigma = \{1\}$ . A string  $w \in \Sigma^+$  ( $\Sigma^+$  is the set of non empty strings over  $\Sigma$ ) will be termed as a **numeral**. A pair of natural numbers, say  $\langle 2, 3 \rangle$ , will be represented by the corresponding numerals separated by  $\star$ . Thus  $\langle 2, 3 \rangle \rightarrow 111 \star 1111$ .

Worked example  shows an MAS defines the computation of the *succ* function.

**Definition 12** A MAS  $S$  computes a  $k$ -ary partial number theoretic function  $f$  provided that

1. if  $S$  is applied to input word  $1^{n_1+1} \star 1^{n_2+1} \star \dots \star 1^{n_k+1}$ , where  $f$  is defined for  $\vec{n}$  (i.e.  $f(\vec{n})$  is defined), then  $S$  yields  $1^{f(\vec{n})+1}$ ; and
2. if  $S$  is applied to input word  $1^{n_1+1} \star 1^{n_2+1} \star \dots \star 1^{n_k+1}$ , where  $f$  is **not** defined for  $\vec{n}$ , then
  1. either  $S$  does not halt,
  2. or if  $S$  does halt, then it's output is **not** of the form  $1^m, m \geq 1$ .

Exercises  and  show a few MASs that compute partial number theoretic functions. If an MAS exists that computes a number theoretic function, then the function is said to be *Markov computable*.

## 5.4 A Few Important Theorems

We state without proof, the main theorem.

**Theorem 7** Let  $f$  be a number theoretic function. Then  $f$  is Markov computable if and only if it is *partial recursive*.

## 5.5 Worked Examples

1. Consider the alphabet  $\Sigma = \{a, b, c, d\}$  and the MAS given by:

1.  $a \rightarrow c$
2.  $bc \rightarrow cb$
3.  $b \rightarrow .cd$  (Note the `.`.)

Given the input word  $w = ``baba"$ , we have:

$w \equiv baba$     rule ??  
 $= bcba$     rule ??  
 $= bcbc$     rule ??  
 $= cbbc$     rule ??  
 $= cbc b$     rule ??  
 $= ccbb$     rule ??  
 $= cccdb$

2. Let  $L = \{(ab)^n | n \in \mathcal{N} \geq 0\}$ , where  $\Sigma = \{a, b\}$  and  $\Gamma = \Sigma \cup \{\$\}$  and  $\Pi$  is:

1.  $\$ab \rightarrow \$$
2.  $\$ \rightarrow .1$
3.  $\epsilon \rightarrow \$$

The above MAS accepts  $L$ . Consider an input word  $w = ``abab"$ .

$w \equiv abab$      $\epsilon$  (empty string) can prefix any string  
 $= \epsilon abab$     rule ??  
 $= \$abab$     rule ??  
 $= \$ab$     rule ??  
 $= \$$     rule ??  
 $= 1$

3. Let  $L = \{(ab)^n | n \in \mathcal{N} \geq 0\}$ , where  $\Sigma = \{a, b\}$  and  $\Gamma = \Sigma \cup \{\$, \#\}$  and  $\Pi$  is:

1.  $\$ab \rightarrow \$$
2.  $\$a \rightarrow \#$
3.  $\$b \rightarrow \#$
4.  $\#a \rightarrow \#$
5.  $\#b \rightarrow \#$
6.  $\$ \rightarrow .1$
7.  $\# \rightarrow .0$
8.  $\epsilon \rightarrow \$$

The above MAS recognises  $L$ . Consider an input word  $w = ``aba"$ .

$w \equiv aba$       $\epsilon$  prefixing  
 $= \epsilon abab$      rule ??  
 $= \$aba$      rule ??  
 $= \$a$      rule ??  
 $= \#$      rule ??  
 $= 0$

4. The *succ* function: This is defined by the MAS  $\Sigma = \{1\} = \Gamma$  and  $\Pi_S$  of an  $S$  be (i)  $1 \rightarrow .11$ .

## 5.6 Exercises

---

1. Apply the MAS in worked example  $\square$  to the input words: ``aaab", ``baaa" and ``abcd".
2. Check that worked example  $\square$  cannot *recognise* a word that is not in  $L$ .
3. Check that worked example  $\square$  can *accept* a word in  $L$  as well as *reject* a word that is not in  $L$ .
4. Consider the MAS given by  $\Sigma = \{1\} = \Gamma$ , and  $\Pi : 11 \rightarrow 1$ . What unary partial number theoretic function does this MAS compute ?
5. Let  $\Sigma = \{1, \star\} = \Gamma$ . What partial number theoretic functions do the following MASs compute ?
  1.  $S = \{\Sigma, \Gamma, \Pi\}$  where  $\Pi$  is:
    1.  $\$1 \rightarrow \$$
    2.  $\$\star 1 \rightarrow \$$
    3.  $\$ \rightarrow .1$
    4.  $\epsilon \rightarrow \$$
  2.  $S = \{\Sigma, \Gamma, \Pi\}$  where  $\Pi$  is:

1.  $\$1 \rightarrow 1\$$
2.  $\$* \rightarrow \%$
3.  $\%1 \rightarrow \%$
4.  $\% \rightarrow .\epsilon$
5.  $\epsilon \rightarrow \$$

## 6 Turing Machines

---

This model is the most popular model of computation and is what is normally presented in most undergraduate and graduate texts on Computation theory. It was conceived by Alan M. Turing in the thirties with a view to mathematically define the notion of an algorithm. Turing was working with Church during that time. During this time Gödel had presented his famous Incompleteness theorem and was formulating the partial recursive functions approach to computation. Interestingly, neither Church nor Gödel were motivated to consider computation explicitly. Their interest was more in the foundational problems of Mathematics, as alluded to earlier. Alan Turing also was motivated by the foundational issues. However, his approach makes an explicit use of the idea of *mechanical computation*. He viewed those foundational problems in Mathematics in terms of purely mechanical computation that could be carried out by a machine. He concretely imagined "mechanical computation" being carried out by humans, called "computers", who would perform the steps of the algorithm **exactly** as specified without using any intelligence. His model of computation therefore gives a rather "materially" imaginable view of computation. The Turing model, though a rigorously mathematical model, therefore has a certain "technological" appeal that makes it<sup>21</sup> attractive as the initial candidate for presenting the theory of Computation<sup>22</sup>. We will spend some time in it's study as most development in theoretical Computer Science has occurred with this perspective. On the face of it, this model appears to present a view of computation that does *not* seem anything like computing the value of a function.

Consider the problem of determining if a given word, say *aabbaaabbbaa*, is a palindrome (i.e. reads the same backwards or forwards) or not. The algorithm that can tell us if a given word is a palindrome or not is quite straightforward. Our purpose here is to emphasize that there **appear** to be problems that are not numeric in nature. Notice that the algorithm divides all possible words that can be given as input into **two** sets: the set of palindromes and the set of words that are not palindromes. It appears to "classify" the word at input as belonging to either one of these sets, never both. This view

of computation is called as the **language recognition** perspective of computation. The  $\lambda$  calculus was a **function computation** view, while the Markov Algorithm approach transformed a given input symbol to an output symbol - a (*symbol*) *transduction* view of computation. Note that by a **language** we simply mean the set of words that are generated from some given alphabet. The language recognition view involves determining if an arbitrary word belongs to some language  $L$  or not, and  $L$  is given in advance. An algorithm that successfully does this is said to **recognize**  $L$ . To get an idea of how the function computation paradigm looks like from this perspective consider the problem of determining if a natural number  $n$  is a prime number. We first construct a "string representation" of a natural number:

A natural number  $n$  will be represented by a string of  $n$  1s, i.e. 111... $n$  times, and is written as

$1^n$  for short. The set of all primes is the language  $L_{Prime} = \{11, 111, 11111, \dots\}$ . The

question: "is  $n$  prime?", is equivalent to asking: "is it true that  $1^n \in L_{Prime}$ ?"

### 6.1 On the Path towards Turing Machines

---

We develop the notion of a Turing machine in steps. Along the way, we will meet a number of useful intermediates. The idea is to develop the concept of (abstract) machines starting from "simple" ones, i.e. with a gradual increase in capabilities. We will draw parallels to the other models of computation if possible.

#### 6.1.1 Basic Machines

A **machine** at it's simplest, would simply recognize an input from a set  $I$  and produce an output from the set  $O$ . The sets  $I$  and  $O$  are finite. Thus a simple machine would receive an input and produce

an output. For instance, a logic gate like (say) the AND gate would be a simple machine. A simple machine just **responds** (as opposed to **reacts**) to the current input stimuli. It has no memory to **react** on the basis of past history. In essence, a simple machine looks up a table of finite size; we can simply tabulate the output to be produced for a given input. As a result, a simple machine is unable to do more complicated tasks. For instance, a simple machine cannot algorithmically add two  $n$  bit numbers. To do so would require remembering the necessary carry digits. But a simple machine has no memory to remember the past history! All it can do is look up a table of the size  $2^n \times 2^n$  and emit out the sum for the given input pair of numbers. The basic machine would be represented by a simple function that maps the input to the output. It's signature would be:

$$I \rightarrow O(18)$$

### 6.1.2 Finite State Machines (FSMs)

If we add an **internal memory** to a basic machine then we can build a machine that performs addition algorithmically since now the carry digits can be remembered locally. Let  $S$  be the set of possible configurations of a finite internal memory. "Remembering past history" would mean that the output produced would depend on both - the input set  $I$  and the internal memory state  $S$ . Further, the given input *could* change the internal memory state which, in turn, would be used in a future output. A machine with an internal state is called a **Finite State Machine (FSM)**.

We can pictorially depict the operation of an FSM using a **transition graph** (also called as a **transition diagram** or a **state diagram**). Consider the problem of designing a machine that performs addition of binary numbers - the binary adder.

**Figure:** Transition Graph for a Binary Adder Machine

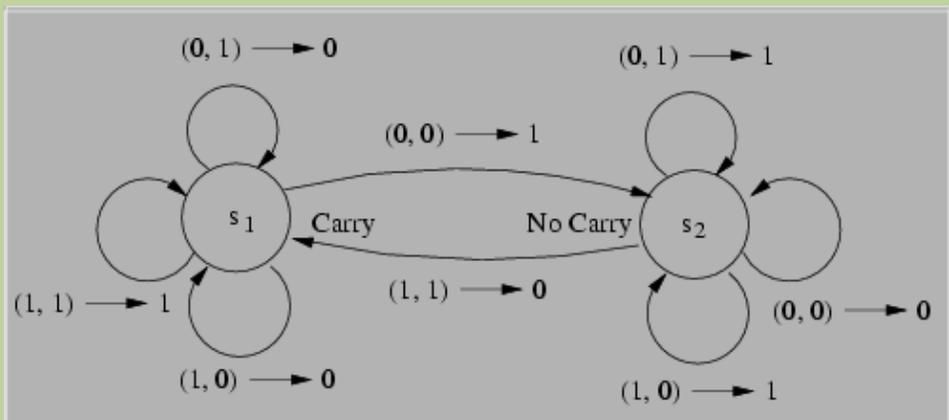


Fig.(1) shows the picture. The labelled circles represent (internal) states and the directed arcs labelled in the form  $i \rightarrow o$  represent that the input  $i \in I$  causes an output  $o \in O$  and shifts the state

$s_i \in S$  to state  $s_j \in S$ . Once the starting state  $s_k$  and the input  $i_0 \in S$  are given, the machine behaviour is defined. While a picture is worth a thousand words, transition graphs become quite unwieldy when the number of states increases. We, therefore, turn to a more formal description of finite state machines.

As pointed out above, a FSM is described by two functions whose signatures are:

$$I \times S \rightarrow O(20)$$

$$I \times S \rightarrow S(21)$$

Eqn.(20) is called as the machine function and Eqn.(21) is the state function. A binary adder machine would be designed as follows:

The sets  $I$ ,  $O$  and  $S$  are:

$$I = \{(0, 0), (0, 1), (1, 0), (1, 1)\} \equiv \{i_1, i_2, i_3, i_4\}$$

$$(21)$$

$$O = \{0, 1\} \equiv \{o_1, o_2\} (22)$$

$$S = \{\text{carry, no-carry}\} \equiv \{s_1, s_2\} (23)$$

The machine function is:

**Table:**Machine function for the Binary Adder machine. The entries in the table are the values from the output for various combinations of  $I \times S$ .

$I \downarrow S \rightarrow$	Carry	No Carry
(0, 0)	1	0
(0, 1)	0	1
(1, 0)	0	1
(1, 1)	1	0

The state function is:

**Table:**State function for the Binary Adder machine. The entries in the table are the values from the set  $S$  for various combinations of  $I \times S$ .

$I \downarrow S \rightarrow$	Carry	No Carry
(0, 0)	No Carry	No Carry
(0, 1)	Carry	No Carry
(1, 0)	Carry	No Carry
(1, 1)	Carry	Carry

Note that the finiteness of the sets  $I$ ,  $O$  and  $S$  will limit our abilities, and we will overcome this aspect when we consider Turing machines. Also, since the machines are designed using finite sets, all the behaviour is completely deterministic - i.e. the machine behaviour can still be tabulated for every possible configuration. However, the existence of a memory has permitted us to **react** rather than just **respond**.

As another illustration, consider designing a machine that can check if the natural number at its input is divisible by three. The sets, the machine function and the state functions are:

$$I = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$S = \{\sigma, s_0, s_1, s_2\}$$

$$O = \{0, 1\} \quad (22)$$

The machine function is given in Table (□) and the state function is given in Table (□).

**Table:**Machine function for the Divisibility-by-Three-Tester machine. The entries in the table are the values from the set  $O$  for various combinations of  $I \times S$ .

$I \downarrow   S \rightarrow$	$\{0, 3, 6, 9\}$	$\{1, 4, 7\}$	$\{2, 5, 8\}$
$\sigma$	1	0	0
$s_0$	1	0	0
$s_1$	0	0	1
$s_2$	0	1	0

**Table:**State function for the Divisibility-by-Three-Tester machine. The entries in the table are the values from the output for various combinations of  $I \times S$ .

$I \downarrow   S \rightarrow$	$\{0, 3, 6, 9\}$	$\{1, 4, 7\}$	$\{2, 5, 8\}$
$\sigma$	$s_0$	$s_1$	$s_2$
$s_0$	$s_0$	$s_1$	$s_2$
$s_1$	$s_1$	$s_2$	$s_0$
$s_2$	$s_2$	$s_0$	$s_1$

The Divisibility-by-Three-Tester machine will be in state  $s_0$  if the number at its input is divisible by three, not otherwise. The machine **recognizes** a number that is divisible by three as whenever the machine is finally in state  $s_0$  for a given input, we can surely say that the input is divisible. We

also say that the machine **decides** if its input is divisible by three. Notice that the output is 1 if the number is divisible, 0 otherwise. We need not actually examine the machine state! The machine is said to **accept** numbers that are divisible by three, and **reject** others. Finally, we observe that a number at the input is a sequence of symbols from the set  $I$ . We have been calling the set  $I$  as the alphabet set  $\Sigma$  and the numbers that are generated as finite sequences of alphabets as words. With  $L$  being the set of all words generated from  $\Sigma$ , in the present case  $L$  is just the set of natural numbers  $\mathcal{N}$ .

FSMs cannot be used to recognize palindromes of any size. That is obvious because FSMs are finite and impose an upper limit on the length of the word that can be given at the input, while the algorithm to recognize a palindrome has to deal with a countably infinitely long sequence of symbols. In contrast, it is possible to construct a MAS that recognizes palindromes. That is because the MAS is **not** restricted to an upper limit of the word length. Arbitrarily long sequences of symbols can be remembered using an external countably infinite memory. Adding such a memory to an FSM gives us the Turing machine. We will take up Turing machines in a short while.

### 6.1.3 Regular Sets and Regular Expressions

As we have seen in the previous section, FSMs can recognize some kinds of sets, but cannot recognize others. We face the question: What kinds of sets are recognised by FSMs ?

A special class of sets of words over  $\Sigma$ , called **regular sets**, is recognized by FSMs<sup>24</sup> and is defined recursively as follows:

#### Definition 13

1. Every finite set of words over  $\Sigma$  (including  $\Phi$ , the empty set) is a regular set.
2. If  $U$  and  $V$  are regular sets over  $\Sigma$ , then  $U \cup V$  and  $UV$  - the concatenation - are also regular.

The *concatenation* of two subsets  $U$  and  $V$  of  $\Sigma^*$  is defined by:

$$UV = \{x \mid x = uv, u \text{ is in } U \text{ and } v \text{ is in } V\}$$

3. If  $S$  is a regular set over  $\Sigma$ , then so is its closure  $S^*$ .

The *closure* operation defines on a set  $S$ , a derived set having the empty word and all words formed by concatenating a finite number of words in  $S$ . i.e.

$$S^* = S^0 \cup S^1 \cup S^2 \dots, \text{ where } S^0 = \Lambda \text{ (}\Lambda \text{ is the empty word symbol) and } S^i = S^{i-1}S \text{ for } i > 0.$$

4. No set is regular unless obtained by a finite number of applications of the above three definitions.

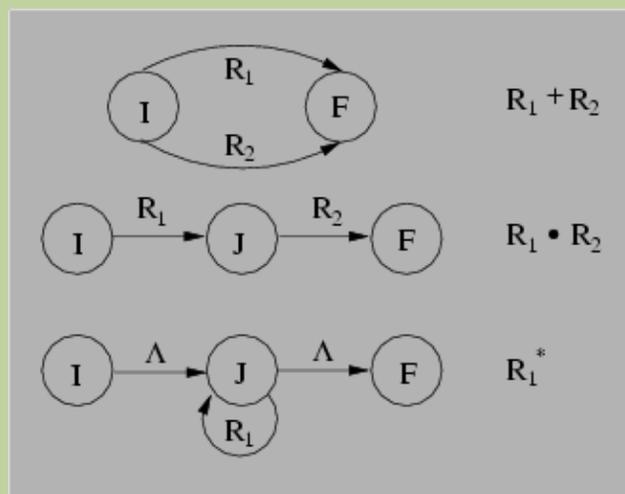
The above definition of regular sets captures the notion of regular behaviour of objects. This can also be cast as a (behavioural) language that is used to express regular sets and the FSM. We take a set of alphabets,  $\Sigma$ , and define regular behaviour recursively as follows:

#### Definition 14

- The alphabets,  $\Lambda$  and  $\Phi$  are regular expressions over  $\Sigma$ ,
- Every alphabet  $\sigma \in \Sigma$  is a regular expression over  $\Sigma$ ,
- If  $R_1$  and  $R_2$  are regular expressions over  $\Sigma$ , then so are  $(R_1 + R_2)$ ,  $(R_1 \cdot R_2)$ , and  $(R_1)^*$ , where  $+$  indicates alternation (parallel path, either  $R_1$  or  $R_2$ , corresponds to the set union operation),  $\cdot$  denotes concatenation (series,  $R_1$  followed by  $R_2$ ), and  $*$  denotes iteration (closure, zero or more of  $R_1$ ).
- The regular expressions are only those that are obtained by using the above three rules.

The FSMs that correspond to the basic operations of alternation, concatenation and closure are shown in Fig. ( ). Notice that these can serve as the "building blocks" to convert a regular expression to its transition graph and vice versa. In other words, given a regular expression, we can create its FSM transition graph and given a FSM transition graph, we can write the regular expression. This conversion forms the basis of programs that manipulate regular expressions, for instance the Unix shells *sh/csh/bash*, *grep*, *sed*, scanners in compilers etc.

**Figure:**The basic operations and their FSMs.



The correspondence between regular expressions and regular sets is as follows: Every regular expression  $R$  over  $\Sigma$  describes a set  $R^+$  of words over  $\Sigma$  (i.e.  $R^+ \subseteq \Sigma^*$ ) as follows:

- If  $R = \Lambda$ , then  $R^+ = \{\Lambda\}$ , a set consisting of the empty word,

2. If  $R = \emptyset$ , then  $R^+ = \emptyset$ , the empty set,
3. If  $R = \sigma$ , then  $R^+ = \{\sigma\}$ ,
4. If  $R_1$  and  $R_2$  are regular expressions over  $\Sigma$  which describe the set of words  $R_1^+$  and  $R_2^+$ , then
  1. if  $R = R_1 + R_2$ , then  $R^+ = R_1^+ \cup R_2^+ = \{x \mid x \in R_1^+ \vee x \in R_2^+\}$ ,
  2. if  $R = R_1 \cdot R_2$ , then  $R^+ = R_1^+ R_2^+ = \{x \mid x \in R_1^+ \wedge x \in R_2^+\}$ ,
  3. if  $R = \{R_1\}^*$ , then  $R_1^+ = \{\Lambda\} \cup \{x \mid x \in (R_1^+)^*\}$

**Remark 1** It is possible to define an enlarged class of regular expressions that have intersection and complementation over and above the operations of union, concatenation and closure.

#### 6.1.4 Properties of FSMs

##### 6.1.4.1 Periodicity

: FSMs lack of capacity to remember *arbitrarily* large amounts of information as there are only a finite number of states. This finiteness defines a limit to the length of the sequence that can be remembered. Also, an FSM will eventually repeat a state, or produce a sequence of states. This periodicity results in a useful characterisation of FSMs - the Pumping Lemma. The lemma simply states that given any sufficiently long string accepted by an FSM, we can find a substring near the beginning that may be *repeated* (pumped) as many times as we like and the resulting string will still be accepted by the FSM.

##### 6.1.4.2 Equivalence Class of Sequences

: Two input sequences  $I_1$  and  $I_2$  are equivalent if the FSM is in the same state after executing them. Because the number of states are finite in an FSM, every input sequence will end up in some state. We can classify input sequences in terms of the states they reach. All input sequences that reach the same state are grouped as one. An FSM thus induces an equivalence class partitioning over the set of input sequences.

##### 6.1.4.3 Impossibility of Multiplication

: It is impossible to carry out multiplication of arbitrarily long sequences of numbers using an FSM. This is because a FSM is finite and hence a limit is imposed on the length of the product. In contrast, addition - binary addition - is possible! Note that multiplication requires remembering the intermediate "products" for later addition.

##### 6.1.4.4 Impossibility of Palindrome Recognition

: A palindrome is a string that is symmetric - it reads the same both ways. To recognise an arbitrarily long palindrome, a machine would need an infinite memory to remember all the alphabets read for later comparison of symmetry. The finiteness of FSM does not permit such a memory and hence a FSM cannot recognise a palindrome

##### 6.1.4.5 Impossibility of Checking Well-formed Parentheses

: The periodicity of an FSM implies that input sequences whose periodicity is greater or sequences that are not periodic cannot be accepted by a FSM since the FSM just does not have those many states. Hence well formed parentheses - balanced brackets - cannot be checked.

## 6.2 The Pushdown Stack Memory Machine

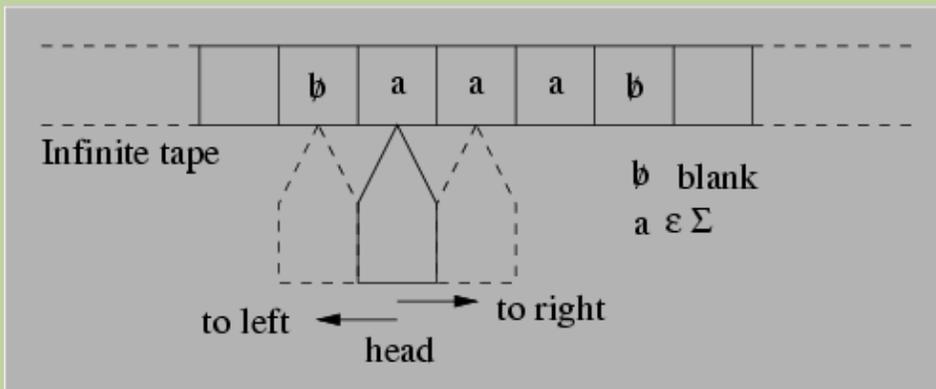
FSMs are limited due to their finite memories. A way of overcoming the limitations of FSMs is to introduce infinite memories. However, mere "inifiniteness" of the memory is **not** sufficient; a certain "freedom" to use the memory is also required. To illustrate this, we introduce a machine called as the *Push Down Stack Memory Machine* (PDM) that overcomes the limitations of FSMs by using an infinite memory. However, the use of memory is restricted in that the memory must be used as a

*stack*. A stack is a structure where the removal of objects from the memory is performed in the reverse order of their arrival into the memory. This behaviour is described as *Last In First Out* and abbreviated as *LIFO*. These machines are routinely applied to recognise context free grammars<sup>26</sup> of programming languages and are not as powerful as TMs.

Our purpose in mentioning them here is two fold: on the one hand, we wish to indicate that some interesting and useful intermediates after FSMs but before TMs are possible. And secondly, we would invite you to explore and find out the properties of PDMs. For instance, FSMs were known to be unable to solve certain problems (balanced parentheses e.g.). Could a PDM solve it? What problems would not be solved even by a PDM? Naturally, the next questions could be like: what other kinds of "restrictions" on memory use are possible and what are the properties of the machines they give rise to?

## 6.3 The Turing Machine

**Figure:**The Basic Turing Machine. Textually, the above figure would be represented as: BHaaaB, where **H** is the head.



The Turing machine is pictured as being composed of a *head* that reads and writes symbols from a set of alphabet,  $\Sigma$ , onto a cell of an external tape (Fig. (□)). The tape is an infinite sequence of cells; each cell can contain one and only one symbol from  $\Sigma$ . The Turing machine operates as follows: the head reads the symbol from the cell currently under it and responds to that symbol by either writing another symbol at the cell or not writing anything at all. It **may** then move one cell either to the left or to the right<sup>27</sup>. We are required to specify, in advance, the response of - i.e. **what** to write and **where**, if at all, to move - the head to every symbol that it reads. The external tape is initially blank. The symbols are first laid out on the tape starting from the left. If the head encounters a blank cell, the machine halts.

To actually work with the machine scheme in the previous paragraph, we need to evolve notation to describe the details like the symbols - alphabet - that can appear on the tape, the starting position of the head, a detailed tabulation of the responses etc. By *state* of the machine, we wish to describe the instantaneous configuration of the machine which is made up of the head position on the tape and the alphabets on the tape. Note that the starting position is merely one of the states. As a result of reading the alphabet on the tape, the machine may write back or move it's head or do both. The machine response thus takes the machine into the **next** state which could be identical to the earlier one

or a **new** one. Given a set  $\Sigma$  of  $k$  alphabets,  $\Sigma = \{a_1, a_2, \dots, a_k\}$  we can have  $k + 3$

responses as follows:

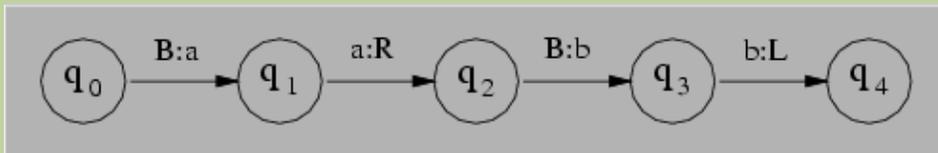
**Table:**  $k + 3$  responses of a Turing machine for an alphabet with  $k$  symbols.

Action	Description
L	Move head one cell to the left
R	Move head one cell to the right
B	Erase whatever is on the cell currently being scanned
$a_1$	Erase whatever is on the cell and replace it by symbol $a_1$
$a_2$	Erase whatever is on the cell and replace it by symbol $a_2$
...	...
$a_k$	Erase whatever is on the cell and replace it by symbol $a_k$

The operation of a Turing machine  $M$  can be described by a quadruple:  $\langle q_i, \sigma; action, q_j \rangle$ . This is read as: If state  $q_i$  reading symbol  $\sigma$  then perform action  $action$  and enter state  $q_j$ . For instance  $\langle q_2, a; R, q_1 \rangle$  would mean that if in state  $q_2$  reading symbol  $a$ , then move head one cell to the right and enter state  $q_1$ . This quadruple is called as an **instruction** (of  $M$ ). Alternately, we can view an instruction as a function - denoted by  $\delta_M$  - that takes in the state and the alphabet being scanned and returns the action and the new state. Thus we also write:  $\delta_M(q_2, a) = (R, q_1)$  for the above quadruple.  $\delta_M$  is not a number theoretic function since it takes state/symbol pair as an argument. It's domain is therefore,  $Dom(\delta_M) : Q \times (\Sigma \cup \{B\})$ , where  $Q = \{q_1, q_2, \dots, q_n\}$  is the set of states, and at least one of those states is a distinguished state - the start state - denoted by  $q_0$  or  $q_{init}$ . Since  $\delta_M$  captures the "motion" from one state to another it is referred to as the **transition function**. Note that the initial tape contents and the initial head position are specified by  $q_0$ .

Consider the following figures that describe a Turing machine.

**Figure:** Transition diagram of a Turing machine  $M$ .



The "arcs" of the state diagram (look like straight lines here) are labelled by the **instructions** of the machine. Each circle represents a **state** of the machine. The machine goes to the state pointed to by the arc when it receives the instruction labelled by the arc when in state from which the arc originates. The starting state is denoted by  $q_0$  and is drawn to the far left, usually. The first arc in Fig. (

Fig. 1) is labelled  $B : a$  and points to state  $q_1$ . This is to be interpreted as: the machine must **write** a blank  $B$  it is currently scanning with the symbol  $a$  and then enter state  $q_1$ . The machine state  $q_0$  has been **specified in advance** to be a head over some cell of an entirely blank tape. Figs. (Fig. 1) pictorially depict the machine in each of the above states.

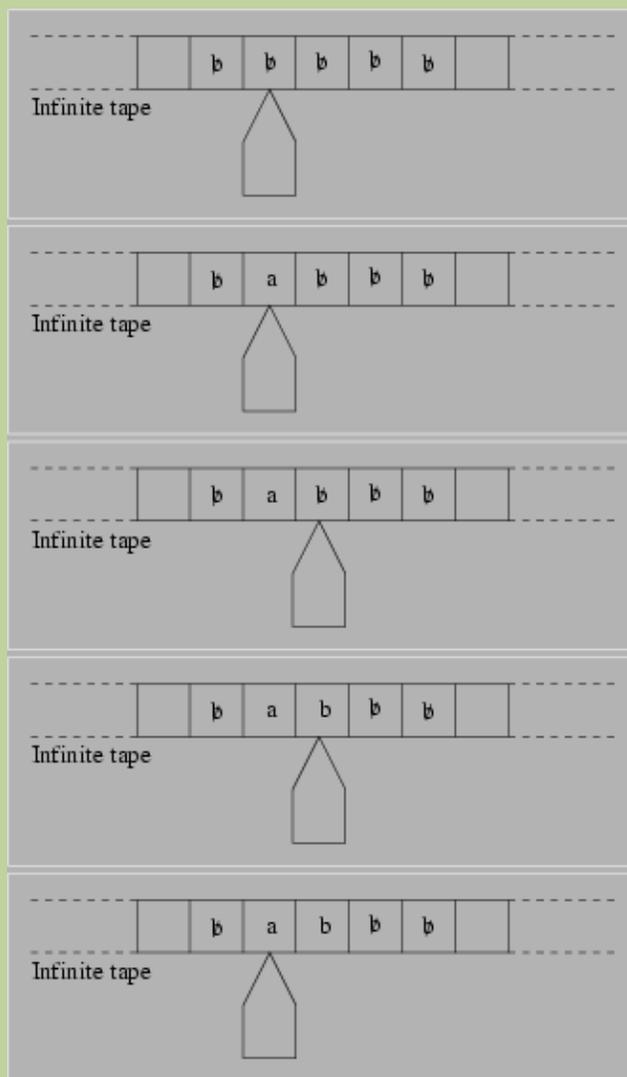
**Figure:** Turing machine in state  $q_0$ .

**Figure:** Turing machine in state  $q_1$ .

**Figure:** Turing machine in state  $q_2$ .

**Figure:** Turing machine in state  $q_3$ .

**Figure:** Turing machine in state  $q_4$ .



The working of the machine can now be described. It starts from a state  $q_0$  where the head is positioned on some cell of an entirely blank tape. It has been instructed that if in this state it **scans** a blank, then it is to **write** an  $a$  on that cell and enter state  $q_1$ . Since it does scan a blank, it writes an  $a$  and enters state  $q_1$ . Now in this state, when it scans and **reads** an  $a$ , it **moves** one cell to the right and enters state  $q_2$ . Here if it **reads** (i.e. scans) a blank (it does), then it **writes** a  $b$  and enters state  $q_3$ . In this state, and the current head position, if it **reads** a  $b$  - which it does - then it **moves** one cell to the left and enters state  $q_4$ . There are no further instructions. Hence the machine **halts** in state  $q_4$ .

In summary, the machine behavior is: When started scanning a cell on a completely blank tape, it writes the word "ab" and halts scanning the symbol "a". We say that the tape configuration is to be completely blank initially. In general, the contents of the tape together with the location of the head for a given Turing machine is referred to as the **tape configuration**. The tape configuration and the current machine state is referred to as the **machine configuration**. Note that if the tape configuration at  $q_0$  is *not* completely blank, and the machine starts scanning a non-black symbol, then it must

halt immediately in state  $q_0$  since there are no instructions! Also, as suggested in the caption to Fig. (

), we will use a textual notation to describe the machine configuration. Finally, the state  $q_i$  is often abbreviated to just  $i$  in state diagrams.

### 6.3.1 Formal Definitions

#### 6.3.1.1 Basic Definition:

**Definition 15** A **deterministic Turing machine**  $M$  is anything of the form

$$\langle Q, \Sigma, \Gamma, q_0, \delta \rangle$$

where  $Q$  is a non empty finite set,  $\Sigma$  and  $\Gamma$  are non empty finite alphabets with  $\Sigma \subset \Gamma$ ,  $q_0 \in Q$ , and  $\delta$  is a (partial) function with domain and codomain given by

$$\delta : Q \times (\Gamma \cup \{B\}) \rightarrow (\Gamma \cup \{B\} \cup \{L, R\}) \times Q$$

$Q$  is the set of states of  $M$ ,  $q_0$  is the initial state of  $M$ ,  $\Sigma$  is the input alphabet of  $M$  and  $\Gamma$  is its tape alphabet (or auxiliary alphabet) and  $\delta$  is the transition function.

### 6.3.1.2 Turing machines as language recognizers:

We define the notion of language **acceptance** and **rejection** to introduce the language recognition paradigm for Turing machines.

**Definition 16** A Deterministic Turing machine  $M$  **accepts** a nonempty word  $w$  if, when started scanning the leftmost symbol of  $w$  on an input tape that contains  $w$  and is otherwise blank,  $M$  ultimately halts scanning a 1 on an otherwise blank tape. It accepts an empty word  $\epsilon$  if, when started scanning a blank on a completely blank tape,  $M$  ultimately halts scanning a 1 on an otherwise blank tape.

**Definition 17** A Deterministic Turing machine  $M$  **accepts** a language  $L$ , if  $M$  accepts all, and only the words  $w$  in  $L$ . A language  $L$  is said to be Turing acceptable if there exists some deterministic Turing machine  $M$  that accepts  $L$ .

**Definition 18** Let  $L$  be a language over alphabet  $\Sigma$ . A Deterministic Turing machine  $M$  **recognizes** language  $L$  if:

1. If  $w$  is an arbitrary nonempty word over  $\Sigma$  and  $M$  is started scanning the leftmost symbol of  $w$  on a tape that contains  $w$  and is otherwise blank, then  $M$  ultimately halts scanning a 1 on an otherwise blank tape if  $w \in L$ , but halts scanning a 0 on an otherwise blank tape if  $w \notin L$ .
2. If  $M$  is started scanning a cell on a completely blank tape, then  $M$  ultimately halts scanning a 1 on an otherwise blank tape if  $\epsilon \in L$ , but halts scanning a 0 on an otherwise blank tape if  $\epsilon \notin L$ .

A language  $L$  is termed **Turing recognizable** if there exists a Turing machine  $M$  that recognizes  $L$ .

Worked examples  $\square$ ,  $\square$  and  $\square$  illustrate the construction of Turing machines for this paradigm.

### 6.3.1.3 Turing machines as function computers:

We define computation of partial functions to introduce the function computation paradigm for Turing machines. As discussed in section ( $\square$ ), we need to decide a scheme to represent natural numbers of a tape. We represent a natural number  $n$  by a string of  $(n + 1)$  1s. When we need to design a machine for functions that take more than one argument - e.g. addition - we will separate the arguments by a single blank.

**Definition 19** A Deterministic Turing machine  $M$  computes a  $k$ -ary partial number theoretic function  $f$  with  $k \geq 1$  provided that:

1. If  $M$  started scanning the leftmost 1 of an unbroken string of  $n_1 + 1$  1s followed by a single blank followed by an unbroken string of  $n_2 + 1$  1s followed by a single blank . . . followed by an unbroken string of  $n_k + 1$  1s followed by a single blank on an otherwise blank tape, where  $f$  happens to be defined for arguments  $n_1, n_2, \dots, n_k$ , then  $M$  halts scanning the leftmost 1 of

an unbroken string of  $f(n_1, n_2, \dots, n_k) + 1$  1s on an otherwise blank tape.

- If  $M$  started scanning the leftmost 1 of an unbroken string of  $n_1 + 1$  1s followed by a single blank followed by an unbroken string of  $n_2 + 1$  1s followed by a single blank . . . followed by an unbroken string of  $n_k + 1$  1s followed by a single blank on an otherwise blank tape, where  $f$  happens to be **not** defined for arguments  $n_1, n_2, \dots, n_k$ , then  $M$  does **not** halt scanning the leftmost 1 of an unbroken string of 1s on an otherwise blank tape.

**Definition 20** A partial number theoretic function  $f$  is said to be **Turing computable** if there exists a Turing machine  $M$  that computes  $f$ .

Worked example  illustrates the construction of Turing machines that compute functions. Note that:

**Remark 2** Every Turing machine with input alphabet  $\Sigma = \{1\}$  computes some unary partial number theoretic function.

#### 6.3.1.4 Turing machines as transducers:

Worked example  illustrates the construction of a Turing machine that works according to the transduction view of computation.

## 6.4 A Few Important Theorems

---

**Theorem 8** *The Blank Tape Theorem: There exists a Turing machine which when started on a blank tape, can write its own description. It is a self-reproducing machine.*

**Theorem 9** *The Halting problem: There does not exist a Turing machine that can decide if the computation process generated by a Turing machine  $T$  will halt.*

**Theorem 10** *It is **not** possible to decide - i.e. have an algorithm - whether a Turing machine will ever print a given symbol  $a \in \Sigma$ .*

**Theorem 11** *It is **not** possible to decide if two Turing machines with the same alphabet are equivalent or not. In particular, this means that we cannot decide the equivalence of two arbitrary programs. This connection with program equivalence is possible because of the notion of an UTM discussed below.*

Since we have already mentioned that the other perspectives of computation are equivalent to the Turing view, we do not state any theorem of equivalence at this point. Instead, we present the notion of a Universal Turing machine. Observe that the formal definition of a Turing machine is simply a **scheme** of describing machines. The question is: is it possible to have a Turing machine describe other Turing machines? The following theorem - due to Alan Turing - asserts this to be true.

#### 6.4.0.1 The Universal Turing Machine (UTM)

**Theorem 12** *There exists a Universal Turing machine.*

An UTM would behave **exactly** as the Turing machine  $M$  that has been described to it. In other words, we can have a single machine to which we merely provide descriptions of the Turing machines we want it to behave like. **This is why we can have a computer - a universal machine - to which we describe our desired machine as a program.**

## 6.5 Chomsky Hierarchy and Markov Algorithms

The Markov Algorithm view of Computation was in terms of production rules that specify the output symbol to replace an input symbol. If we place gradual restrictions on the way production rules are to be "designed" we have a correspondence between the grammar and the machine that recognises the grammar. There are four classes of grammars:

- [Type 0] No restrictions on the production rules. This is the kind of grammar that we have studied in Markov Algorithms. TMs are required to recognise sentences generated by these grammars - in other words, Markov Algorithms are equivalent to Turing Machines!
- [Type 1] Two restrictions are imposed: (i) production rules are of the form  $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ , and (ii) the start symbol  $S$  does *not* appear on the right hand side of a production. Because the replacement of  $A$  by  $\beta$  is dependent on the occurrence of  $\alpha_1$  as prefix and  $\alpha_2$  as suffix, the grammar is said to be *Context Sensitive Grammars* (CSGs). A class of machines called Linearly Bounded Machines (LBMs) are required to recognise sentences generated by these grammars.
- [Type 2] The left hand side of each production rule is a non terminal symbol; the production rules are of the form  $A \rightarrow \alpha$ . Start symbol is allowed to appear in the RHS of the rule. Such grammars are called as *Context Free Grammars* (CFGs) and PDMs recognise the sentences generated by such grammars. The grammars of most programming languages that we use are CFGs and hence their parsers are PDMs.
- [Type 3] The productions of this class of grammars have a non terminal on the LHS and at most one non terminal on the RHS. These grammars are identical to regular expression languages and FSMs recognise the sentences they generate. They are called *Regular Grammars*.

These classes of grammars was first proposed by Noam Chomsky and is called as the Chomsky hierarchy. From the description above, we also see that the various machines that we have encountered while developing the Turing machine view of Computation correspond to certain restrictions on the production rules of Markov Algorithm Schema. It is known that Panini, the Sanskrit grammarian has defined the Sanskrit language in terms of formal production rules in the same spirit as the above formal language theory[3].

## 6.6 Worked Examples

### 1. Same number of *as* and *bs*:

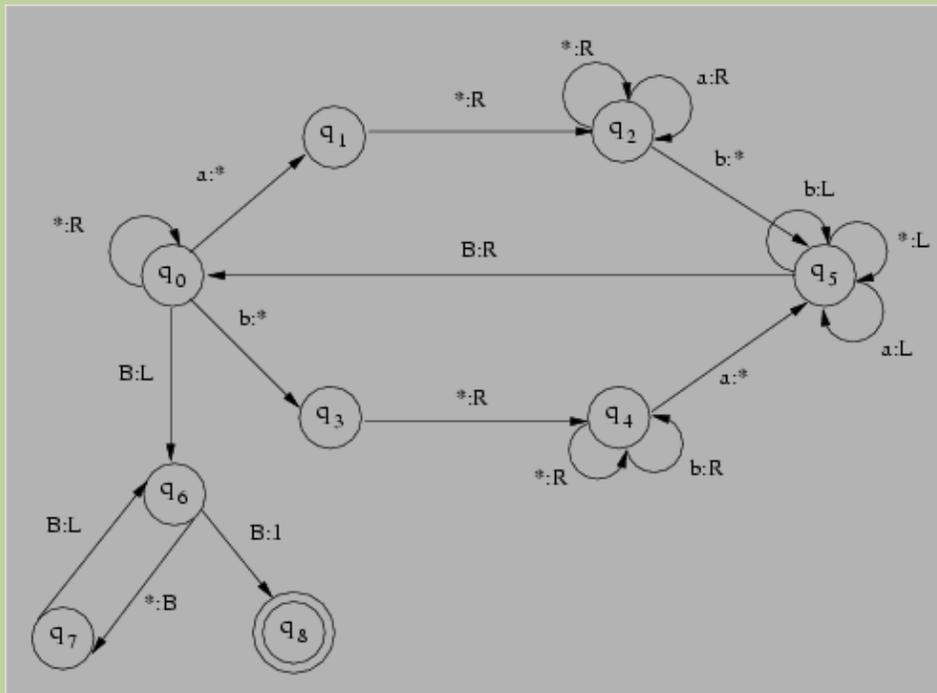
This machine behaves as: When starting scanning the leftmost symbol of word  $w$  consisting of an unbroken string of *as* and *bs* in any order, the Turing machine halts scanning a single  $\perp$  on an otherwise blank tape **if and only if** the number of *as* in  $w$  is equal to the number of *bs* in  $w$ .

For instance, for an initial machine configuration  $BHabbbaaB$ , the machine is required to halt in the configuration  $BH\perp B$ .

**Solution:** To design the machine, we first "think" of an algorithm. The basic idea is to eliminate *as* and *bs* in pairs until either: (i) every symbol has been eliminated, i.e. number of *as* in  $w$  = number of *bs* in  $w$ , or (ii) some symbol is eliminated for which no companion is found, i.e. number of *as* in  $w$   $\neq$  number of *bs* in  $w$ . To find a pair of *a* and *b*, we must have the machine scan a symbol, *mark* it and search for the companion symbol. If the companion symbol is found, then it must again be replaced by the same marker. This process must be repeated over every pair<sup>30</sup>. If no more pairs exist **and** no more symbols exist, then our machine has told us that the given input had the same number of *as* and *bs*. Otherwise, it tells us that the number of *as* was different from the number of *bs*. Notice that the tape of the machine will contain only the marker symbol and blanks if and only if the number of *as* and *bs* are same! Let us use the  $\star$  as the marker. Then for a word  $w = abbbbaa$ , we want the machine to halt with  $BH\perp B$ . The machine (Fig.(□)) would be required to operate as follows:

$BHabbbaaB$	$\rightarrow$	$BH*bbbaaB$	$\rightarrow$	$B*HbbbaaB$	$\rightarrow$
$B*H*bbbaaB$	$\rightarrow$	$BH**bbbaaB$	$\rightarrow$	$HB**bbbaaB$	$\rightarrow$
$BH**bbbaaB$	$\rightarrow$	$B*H*bbbaaB$	$\rightarrow$	$B**HbbbaaB$	$\rightarrow$
$B**H*bbbaaB$	$\rightarrow$	$B***HbbbaaB$	$\rightarrow$	$B***bHaaB$	$\rightarrow$
$B***bH*aaB$	$\rightarrow$	$B***Hb*aaB$	$\rightarrow$	$B*H**b*aaB$	$\rightarrow$
$BH***b*aaB$	$\rightarrow$	$HB***b*aaB$	$\rightarrow$	$BH***b*aaB$	$\rightarrow$
$B*H**b*aaB$	$\rightarrow$	$B**H*b*aaB$	$\rightarrow$	$B***Hb*aaB$	$\rightarrow$
$B***H**aaB$	$\rightarrow$	$B****H**aaB$	$\rightarrow$	$B*****HaB$	$\rightarrow$
$B*****H*B$	$\rightarrow$	$B*****H**B$	$\rightarrow$	$B***H***B$	$\rightarrow$
$B**H****B$	$\rightarrow$	$B*H*****B$	$\rightarrow$	$BH*****B$	$\rightarrow$
$HB*****B$	$\rightarrow$	$BH*****B$	$\rightarrow$	$B*H*****B$	$\rightarrow$
$B**H****B$	$\rightarrow$	$B***H***B$	$\rightarrow$	$B*****H*B$	$\rightarrow$
$B*****H*B$	$\rightarrow$	$B*****HB$	$\rightarrow$	$B*****H*B$	$\rightarrow$
$B*****HBB$	$\rightarrow$	$B*****H*BB$	$\rightarrow$	$B*****HBBB$	$\rightarrow$
$B***H*BBB$	$\rightarrow$	$B***HBBBB$	$\rightarrow$	$B**H*BBBB$	$\rightarrow$
$B**HBBBBB$	$\rightarrow$	$B*H*BBBBB$	$\rightarrow$	$B*HBBBBBB$	$\rightarrow$
$BH*BBBBBB$	$\rightarrow$	$BHBBBBBB$	$\rightarrow$	$BH1BBBBBB$	$\bullet$

**Figure:** Turing machine that detects same number of *a*s and *b*s in an input word. A state with a pair of concentric circles denotes the **endstate**; in this example  $q_8$  is the end state.

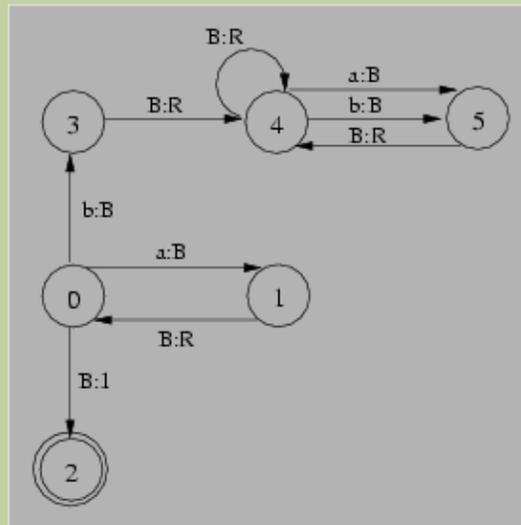


2. Machine that accepts  $L = \{a^n | n \geq 0\}$ :

The problem is to design a Turing machine that *accepts* words from the language  $L = \{a^n | n \geq 0\}$ . These are strings of the form *a*, *aa*, *aaa* etc. Strings like *aab* should not be accepted.

**Solution:** As long as we are reading an *a*, we replace it by a blank and move right. If we have read an *a*, replaced it by a blank and moved right, then we must have the machine in the same state. If on the other hand, the machine reads anything but an *a*, then we must take the machine state into a completely different "cycle". This other "cycle" may or may not halt the machine; we do not care as it is just an acceptor machine. The states 0, 1 and 2 accept the word if it is in  $L$ , else the machine operates in the other states, i.e. 3, 4 and 5. That is what is essentially done in Fig.(□).

**Figure:** Turing machine that accepts  
 $L = \{a^n | n \geq 0\}$ .

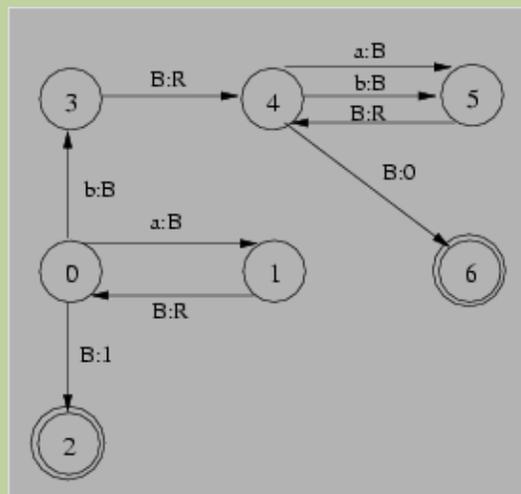


3. **Machine that recognizes**  $L = \{a^n | n \geq 0\}$ :

The problem is to design a Turing machine that *recognizes* words from the language  $L = \{a^n | n \geq 0\}$ . These are strings of the form  $a$ ,  $aa$ ,  $aaa$  etc. Strings like  $aab$  should be *rejected*.

**Solution:** As long as we are reading an  $a$ , we replace it by a blank and move right. If we have read an  $a$ , replaced it by a blank and moved right, then we must have the machine in the same state. If on the other hand, the machine reads anything but an  $a$ , then we must take the machine state into a completely different "cycle". For a recognizer machine, however, we must have this other "cycle" terminate in a rejecting state, where the machine ends up writing a 0 on the tape. The states 0, 1 and 2 accept a valid word in  $L$ , while the others reject it if otherwise. That is what is essentially done in Fig. (□).

**Figure:** Turing machine that recognizes  
 $L = \{a^n | n \geq 0\}$ .



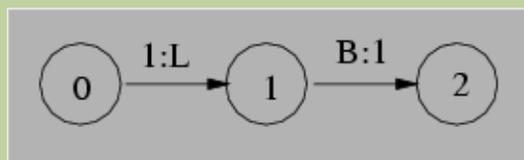
4. **The succ function computer:**

Design a Turing machine that computes the successor of a natural number given on its tape. A natural number  $n$  is represented as an unbroken sequence of  $(n + 1)$  1s (as in an MAS).

**Solution:** Starting from the leftmost 1 of the number, we keep moving to the right until we scan a blank.

At this point, we simply write a 1 in place of the blank and stop. The state diagram of the machine is shown in Fig.(□). By the way, the diagram is incomplete. Please complete it.

**Figure:**Turing machine that computes the *succ*function.

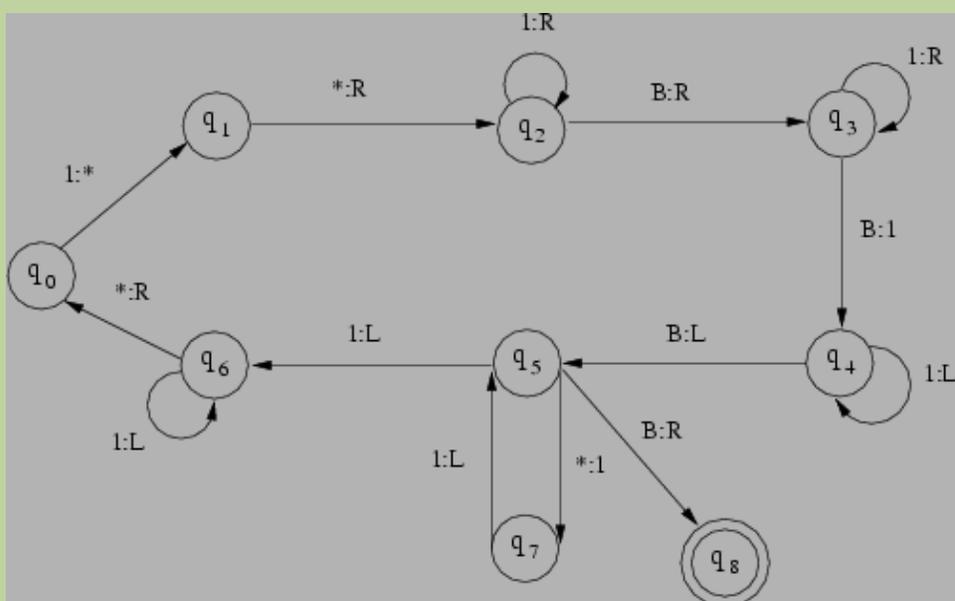


### 5. The Copying machine:

The Copying machine problem is to design a Turing machine that given a string on the tape, writes a copy of the string after the string separated by a blank. Thus if the copying machine receives *BHabaB* as input, it outputs *BHabaBabaB*.

**Solution:** See Fig.(□).

**Figure:**Turing machine that copies it's input word.



## 6.7 Exercises

1. Design a Turing machine that adds two natural numbers.
2. Design a Turing machine that multiplies two natural numbers. (*Hint: Use the copying machine and the representation scheme.*)

## 7 An Overview of Related Topics

A number of other models of computation exist, for example, Logic with Horn clauses, Register machines and the Abacus. Some models have been used to construct programming languages, for instance Prolog which is based on Logic with Horn clauses, and others are used to develop hardware concepts, for instance the Register machine. Some have even been ancient, like the Abacus, although I know of no evidence that the theory of computation was known to the ancients. Attempts at understanding the nature of algorithms has a long history too. The Russian mathematician, Leibnitz - a contemporary of Newton, attempted to understand it too. Given the variety of different models, we have a variety of approaches to solve programming problems. In fact, we may consider the act of writing programs as designing specific machines.

## 7.1 Computation Models and Programming Paradigms

---

The  $\lambda$  calculus approach views computation through an explicit consideration of the behavioural aspects of objects that can undergo computation. This view gives us the *functional* paradigm of programming. On the other hand, the Markov algorithm perspective views computation as symbol transformations explicitly, a view that gives us the *transduction* (of input symbols to output symbols) paradigm. Finally, the explicit consideration of the actions involved in computation - the Turing approach, gives us the *procedural* view of programming. We know that all these approaches are equivalent and there should be no reason to prefer one over the other. Indeed, we ought to select the paradigm that best suits the problem at hand. However, practice deviates from such "ideal" considerations and we find the procedural style dominantly used in software development.

## 7.2 Complexity Theory

---

## 8 Concluding Remarks

---

### Bibliography

---

- 1 *Models of Computation and Formal Languages*, R. Gregory Taylor, Oxford University Press, 1998.
- 2 *Introductory Theory of Computer Science*, E. V. Krishnamurthy, East-West Press, 1983.
- 3 <http://www-groups.dcs.st-and.ac.uk/history/Mathematicians/Panini.html>.

---

[Next Group](#) [Up](#) [Previous](#)

Abhijat Vichare 2005-11-09

---

Email: [amv\[at\]cfdvs.iitb.ac.in](mailto:amv[at]cfdvs.iitb.ac.in)

© Abhijat Vichare  
[Disclaimer](#)

Last Site Update: Nov. 09,  
2005