



Turing Machine for Distributed Computing Model

Acc. No. (PMS) T-1408



Developed by:

Saeed ur Rahman Khan

Supervised by:

Dr. M. Sikander Hayat Khiyal

**Department of Computer Science
Faculty of Applied Sciences
International Islamic University, Islamabad
2006**



بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

**In the name of ALMIGHTY ALLAH,
The most Beneficent, the most
Merciful.**

**Department of Computer Science,
International Islamic University, Islamabad.**

15-February 2006

Final Approval

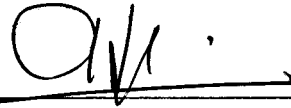
It is certified that we have read the thesis, titled "Turing Machine for Distributed Computing Model" submitted by Saeed ur Rahman Khan under University Reg. No. 234-CS/MSCS/F04. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad, for the Degree of MS in Computer Science.

Committee

External Examiner

Dr. Abdus Sattar

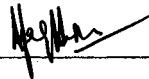
Ex- Director General
Pakistan Computer Burro.



Internal Examiner

Dr. M. Afaq

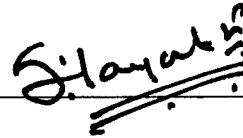
Head, Department Electronics Engineering,
International Islamic University, Islamabad.



Supervisor

Dr. M. Sikandar Hayat Khiyal

Head, Department of Computer Science,
International Islamic University, Islamabad.



Dedication

**Dedicated to my Family who supported me in all
aspects throughout my life.**

A dissertation submitted to the
Department of Computer Science,
International Islamic University, Islamabad
as a partial fulfillment of the requirements
for the award of the degree of
MS in Computer Science

Declaration

I hereby declare that this software, neither as a whole nor as a part thereof has been copied out from any source. It is further declared that I have developed this software entirely on the basis of my personal efforts made under the sincere guidance of our teachers. No portion of the work presented in this thesis has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

Saeed ur Rahman Khan
234-CS/MSCS/F04

Acknowledgements

All praise to the Almighty Allah, the most Merciful, the most Gracious, without whose help and blessings, I would have been entirely unable to complete the project.

Thanks to my Parents who helped me during my most difficult times and it is due to their unexplainable care and love that I am at this position today.

Thanks to my project supervisor Dr. M. Sikandar Hayat Khiyal, whose sincere efforts helped me to complete my project successfully.

Acknowledgement is also due to my teachers and friends specially Khaldoon and Naeem for their help in this dissertation. Also due to my lovely Sohail who acted as my younger brother for the whole time he was with me.

Saeed ur Rahman Khan
234-CS/MSCS/F04

Project in Brief

| | |
|------------------------|--|
| Project Title: | Turing Machine for Distributed Computing Model |
| Objective: | To prove the famous theory "Equivalence of Turing Machine and Computer" more efficiently. |
| Undertaken By: | Saeed ur Rahman Khan Reg. 234-CS/MSCS/F04 |
| Supervised By: | Dr. M. Sikandar Hayat Khiyal Head, Department of Computer Science International Islamic University, Islamabad. |
| Technologies Used: | MS Visual C++6 |
| System Used: | Pentium® IV |
| Operating System Used: | Microsoft Windows XP |
| Date Started: | 11 st May, 2003 |
| Date Completed: | 15 th February, 2006 |

Abstract

The aim of this project is to prove the famous theory "Equivalence of Turing Machine and Computer" with more sound reasoning. Turing machine is model for general-purpose computing device. There are many implementations of Turing Machine using physical computer as well as many implementations of physical computing device using Turing Machine. This project elaborates the simulation of computing model, using a high-level language as its native code, on Turing machine. Efforts have been made to make it possible to communicate among many of such simulations and finally, a deterministic Turing model is designed that is able to actually simulate distributed processing.

A computing model based on the working of Turing machine is developed using Microsoft Visual C++6

TABLE OF CONTENTS

| Chapter No. | Content | Page No. |
|-------------|---|-----------|
| 1. | INTRODUCTION | 1 |
| 1.1. | TURING MACHINES ACCORDING TO A.M.TURING'S PAPER..... | 4 |
| 2. | BASICS OF TURING MACHINE..... | 9 |
| 2.1. | TURING MACHINES AND COMPLEXITY | 9 |
| 2.1.1. | <i>Undecidable Problems</i> | 10 |
| 2.1.2. | <i>Reducing one problem to another</i> | 13 |
| 2.1.3. | <i>Universal Turing Machine</i> | 15 |
| 2.1.4. | <i>Turing Machines and Lambda Calculus</i> | 15 |
| 2.1.5. | <i>Computability theory</i> | 17 |
| 2.1.6. | <i>Halting Problem/ function</i> | 20 |
| 2.1.7. | <i>Encoding of Turing Machine's program</i> | 23 |
| 2.1.8. | <i>Church-Turing Thesis</i> | 23 |
| 2.1.9. | <i>Programming Techniques for Turing Machines</i> | 25 |
| 2.1.10. | <i>Countability and Diagonalization</i> | 26 |
| 2.1.11. | <i>Uncountable Sets</i> | 29 |
| 2.1.12. | <i>The Decision Problem</i> | 31 |
| 2.1.13. | <i>Induction and Recursion</i> | 32 |
| 2.1.14. | <i>The Decision Problem In A.M.Turing's Words or Das Entscheidungsproblem</i> | 34 |
| 2.2. | EQUIVALENCE BETWEEN TURING MACHINE AND COMPUTER | 35 |
| 3. | OUR MODEL | 41 |
| 3.1. | AIMS OF OUR RESEARCH | 41 |
| 3.2. | OVERALL WORKING IN BRIEF OF OUR TURING MACHINE | 43 |
| 3.3. | TURING MACHINE EXECUTING THIRD GENERATION LANGUAGE INSTRUCTIONS..... | 44 |
| 3.4. | VARIABLE DEFINITION | 44 |
| 3.5. | VARIABLE READING:..... | 45 |
| 3.6. | VARIABLE UPDATING:..... | 46 |
| 3.7. | STATEMENTS: | 48 |
| 3.8. | ASSIGNMENT STATEMENT..... | 48 |
| 3.9. | SEND / RECEIVE EXPRESSION TO ANOTHER TURING MACHINE..... | 48 |
| 3.9.1. | <i>Send</i> | 49 |
| 3.9.2. | <i>Receive</i> | 50 |
| 3.10. | READ INPUT / WRITE OUTPUT: | 51 |
| 3.11. | EXPRESSIONS: | 51 |
| 3.12. | EVALUATING POSTFIX EQUATION: | 53 |
| 3.13. | CONDITIONAL BLOCKS:..... | 55 |
| 3.14. | LOOPS:..... | 56 |
| 3.15. | FUNCTION DEFINITION/CALLING: | 56 |
| 3.16. | INTERFACE..... | 59 |
| 3.17. | SYNTAX OF LANGUAGE | 61 |
| 4. | RESULTS..... | 64 |
| 4.1. | PROGRAM TO DISPLAY: | 64 |
| 4.2. | PROGRAM TO ASSIGN: | 64 |
| 4.3. | PROGRAM TO MULTIPLY TWO NUMBERS: | 64 |
| 4.4. | PROGRAM TO RECEIVE AND DISPLAY:..... | 65 |
| 4.5. | PROGRAM TO RECEIVE, MANIPULATE AND DISPLAY: | 65 |
| 4.6. | PROGRAM TO RECEIVE, IF ELSE: | 65 |
| 4.7. | PROGRAM TO RECEIVE, NESTED IF ELSE (FIND SMALLEST): | 66 |

| | | |
|-----------|--|-----------|
| 4.8. | PROGRAM TO RECEIVE, LOOP (CALCULATE FACTORIAL):..... | 66 |
| 4.9. | NESTED LOOP | 66 |
| 4.10. | PROGRAM TO SEQUENCE:..... | 67 |
| 4.11. | PROGRAM TO FUNCTION RECEIVE, LOOP: | 68 |
| 5. | CONCLUSION AND FUTURE WORK | 70 |
| 5.1. | CONCLUSION..... | 70 |
| 5.2. | FUTURE WORK | 70 |

Chapter 1

Introduction

1. Introduction

The project is about to implement a computing system on a Turing machine. Let's first describe the Turing machine. Turing machines are not physical objects but mathematical ones that express the functional properties of a system. The architecture is simply described, and the actions that may be carried out by the machine are simple and unambiguously specified. It is not necessary to mention how the machine carries out its actions, but the point is to specify that the machine can carry out the specified actions, and that those actions may be uniquely described.

A system can be a physical or a virtual-system on any other physical system. In the context of Turing Machine, we may classify systems into three categories.

- A Computing System: It is a system that understands a language, read source code written in that language and executes the code. We may say that a 'language' is a set of rules to express instructions to a computing system.
- A Compiler: It is a system that read a source code written in one language and translates into another language. It does not execute the code.
- A Special Purpose System: It is to perform special user required functionality.

We may use the term system, algorithm and Turing machine interchangeably because it is shown in different literature that for every problem that can be solved by a computer (algorithm) there exist a Turing machine. Moreover we believe that, the purpose of building a system is to implement an algorithm.

It can be said that compilers or computing systems separately are subset of computing system, because if we have a close look to a compiler, it reads an algorithm written in one language and translate it into another language but, it itself is an algorithm. Same is true for a computing system. The hardware abstraction layer, the virtual machines and operating systems are typical examples of logical computing system. Different processors having their own instruction set architecture with which they accept code to execute are examples of physical computing system. Virtual or logical systems are algorithms implemented in languages understandable by the systems on which they are running.

All the modern devices whether they are logical or physical (electronic or mechanical or combinations of both) are designed to implement algorithms and we call them systems. All the physical, chemical or microelectronic theories are to implement the algorithms. The efficiency of the whole system depends on the efficiency of the algorithm. Physical or microelectronic optimization has less impact on the overall system efficiency than optimization of the algorithm. So Turing machine can solve all and only those problems solvable by a computer system (algorithm).

There are four classes of machine those are finite automata, push down automata, linear bounded automata and Turing machine where according to capabilities the finite automata is subset of push down automata, pushdown automata is subset of linear bounded automata, and linear bounded automata is subset of Turing machine. Turing machine full flag simulates real computer. Each class of machine accepts specific type of languages having specific grammars.

A Turing machine is a kind of state machine. At any time the machine is in any one of a finite number of states. Instructions for a Turing machine consist in specified conditions under which the machine will transition between one state and another.

A Turing machine has an infinite one-dimensional tape divided into cells. Traditionally we think of the tape as being horizontal with the cells arranged in a left-right orientation. The tape has one end, at the left, and stretches infinitely far to the right. Each cell is able to contain one symbol, either '0' or '1'.

The machine has a read-write head, which at any time scanning a single cell on the tape. This read-write head can move left and right along the tape to scan successive cells.

The action of a Turing machine is determined completely by the current state of the machine, the symbol in the cell currently being scanned by the head and a table of transition rules, which serve as the "program" for the machine.

The actions available to a Turing machine are either to write a symbol on the tape in the current cell, or to move the head one cell to the left or right. If the machine reaches a situation in which there is not exactly one transition rule specified, that is, none or more than one, then the machine halts.

In modern terms, the tape serves as the memory of the machine, while the read-write head is the memory bus through which data is accessed and updated by the machine. There are two important things to notice about the definition. The first is that the machine's tape is infinite in length, corresponding to an assumption that the memory of the machine is infinite. The second is similar in nature, but not explicit in the definition of the machine, namely that a function will be Turing-computable if there exists a set of instructions that will result in the machine computing the function regardless of the amount of time it takes. One can think of this as assuming the availability of infinite time to complete the computation.

These two assumptions are intended to ensure that the definition of computation that results is not too narrow. This is to ensure that no computable function will fail to be Turing-computable solely because there is insufficient time or memory to complete the computation. If a function is not Turing-computable it is because Turing machines lack the computational machinery to carry it out.

This project defines a Turing machine as a computing system model that has own set of rules, symbols. A language similar to the high level language computer language has been made as the understandable code of the Turing machine. When the Turing machine is given source code using this high level language it execute the code. In this Turing machine each sub Turing machine is a computing system and each has an identifier with which these communicate among themselves.

A Distributed system is a collection of processors those do not share memory or a clock. Each processor has its own local memory and clock. These processors communicate with one another through various communication lines. Processors in a distributed system vary in size and functionality. In a distributed system users servers and resources are distributed over the whole system. A distributed system provides user with access to the resources that the system provides. Service activity has also to be carried out across the network but whole these things are transparent to the users. A client interface should not distinguish between a local and a remote resource.

So when implemented, these Turing machines act as virtual distributed machines across the system. On the other hand the algorithm written in its code finds itself in a complete

computing system. This Turing machine is actually composed of many smaller Turing machines each independently a standard computing system. The smaller components communicate among themselves to simulate the communications of a distributed system.

But let's first give a brief introduction about Turing Machine.

Turing's Question was, how do we compute?

His answer: read, write, look, control.

read: input to Turing Machine

write: output from Turing Machine

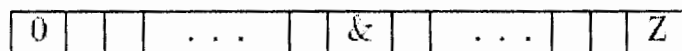
look: move (Turing Machine) tape head

1.1. Turing Machines According to A.M.Turing's Paper

During the International Congress of Mathematicians in Paris in 1900 David Hilbert (1862–1943), one of the leading mathematicians of the last century, proposed a list of problems for the following generations to ponder. On the list was whether the axioms of arithmetic are consistent, a question which would have profound consequences for the foundations of mathematics. Continuing in this direction, in 1928 Hilbert proposed the decision problem (das Entscheidungsproblem), which asked whether there was a standard procedure that can be applied to decide whether a given mathematical statement is true. In his revolutionary paper^[1] Alan Turing (1912–1954) proved that the decision problem had no solution, and in doing so he outlined the rudimentary ideas which form the basis for the modern programmable computer. Today his construction is known as a Turing machine.

| Configuration | | Behavior | |
|---------------|--------|-----------|-----------------|
| m-config. | symbol | operation | final m-config. |
| α | none | RP1 | β |
| α | 1 | RP0 | β |
| α | 0 | HALT | (none) |
| β | 1 | RP1 | α |
| β | 0 | RP0 | α |

Given finite, non-empty, sets A and B , design a Turing machine which tests whether $A \subseteq B$. Suppose that the first character on the tape is a 0, simply to indicate the beginning of the tape. To the right of 0 follow the (distinct, non-blank) elements of A , listed in consecutive positions, followed by the symbol $\&$. To the right of $\&$ follow the (distinct, non-blank) elements of B , listed in consecutive positions, followed by the symbol Z to indicate the end of the tape:



The symbols 0, $\&$, Z are neither elements of A nor B . The machine starts reading the tape in the right most position, at Z . If $A \subseteq B$, have the machine erase all the elements of A and return a tape with blanks for every square which originally contained an element of A . You may use the following operations for the behavior of the machine:

- R: Move one position to the right.
- L: Move one position to the left.
- S: Store the scanned character in memory. Only one character can be stored at a time.
- C: Compare the currently scanned character with the character in memory. The only operation of C is to change the final configuration depending on whether the scanned square matches what is in memory.
 - E: Erase the currently scanned square
 - P(): Print whatever is in parentheses in the current square.

You may use multiple operations for the machine in response to a given configuration. Also, for a configuration q_n , you may use the word “other” to denote all symbols $S(r)$ not specifically identified for the given q_n . Be sure that your machine halts.

⇒ Computing Machines:

We have said that the computable numbers are those whose decimals are calculable by finite means. This requires more explicit definition. No real attempt will be made to justify the definitions given until we reach §9. For present I shall only say that the justification lies in the fact that the human memory is necessarily limited. We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions q_1, q_2, \dots, q_R , which will be called the “m-

configurations". The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". At any moment there is just one square, say the r -th, bearing the symbol $S(r)$ which is "in the machine". We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its m -configuration the machine can effectively remember some of the symbols it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the m -configuration q_n and the scanned symbol $S(r)$. This pair $q_n, S(r)$ will be called the "configuration"; thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (that bears no symbol) the machine may write down a new symbol on the scanned square; in other configurations it may erase the scanned symbol or overwrite. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m -configuration may be changed. Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to "assist the memory". It will only be these rough notes which will be liable to erasure. It is my contention that these operations include all those which are used in the computation of a number. The defense of this contention will be easier when the theory of the machines is familiar to the reader. In the next section I therefore proceed with the development of the theory and assume that it is understood what is meant by "machine", "tape", "scanned", etc.

⇒ Automatic machines:

If at each stage the motion of a machine (in the sense of §1) is completely determined by the configuration, we shall call the machine an "automatic machine" (or a-machine). For some purposes we might use machines (choice machines or c-machines) whose motion is only partially determined by the configuration (hence the use of the word "possible" in §1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic

systems. In this paper I deal only with automatic machines, and will therefore often omit the prefix a-.

⇒ Computing machines.

If an a-machine prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine. If the machine is supplied with a blank tape and set in motion, starting from the correct initial m-configuration the subsequence of the symbols printed by it which are of the first kind will be called the sequence computed by the machine. The real number whose expression as a binary decimal is obtained by prefacing this sequence by a decimal point is called the number printed by the machine.

At any stage of the motion of the machine, the number of the scanned square, the complete sequence of all symbols on the tape, and the m-configuration will be said to describe the complete configuration at that stage. The changes of the machine and tape between successive complete configurations will be called the moves of the machine.

Examples of computing machines:

I. A machine can be constructed to compute the sequence 010101 The machine is to have the four m-configurations "b", "c", "f", "e" and is capable of printing "0" and "1". The behaviour of the machine is described in the following table in which "R" means "the machine moves so that it scans the square immediately on the right of the one it was scanning previously". Similarly for "L". "E" means "the scanned symbol is erased" and "P" stands for "prints". This table (and all succeeding tables of the same kind) is to be understood to mean that for a configuration described in the first two columns the operations in the third column are carried out successively, and the machine then goes over into the m-configuration described in the last column. When the second column is blank, it is understood that the behaviour of the third and fourth columns applies for any symbol and for no symbol. The machine starts in the m-configuration b with a blank tape. (Example 1).

| Configuration | | Behaviour | |
|---------------|--------|-----------|-----------------|
| m-config. | symbol | operation | final m-config. |
| <i>b</i> | none | P0, R | <i>c</i> |
| <i>c</i> | none | R | <i>e</i> |
| <i>e</i> | none | P1, R | <i>f</i> |
| <i>f</i> | none | R | <i>b</i> |

If (contrary to the description §1) we allow the letters L, R to appear more than once in the operations column we can simplify the table considerably.

| Configuration | | Behaviour | |
|---------------|--------|-----------|-----------------|
| m-config. | symbol | operation | final m-config. |
| <i>b</i> | none | P0 | <i>b</i> |
| <i>b</i> | 0 | R, R, P1 | <i>b</i> |
| <i>b</i> | 1 | R, R, P0 | <i>b</i> |

A Turing Machine may have multiple tape or multiple head or multi dimensional tape but all the following attempts have been proven to be equivalent to a ordinary Turing Machine.

- Composition of Multiple Turing Machine
- Multiple Tape
- Multiple read/Write head
- Multi dimensional tape
- Non determinism.

That is we can construct a Turing Machine which is equivalent to any of the above of these.

Chapter 2

Basics of Turing Machine

2. Basics of Turing Machine

This chapter provides a comprehensive knowledge about some advance issues of Turing Machine, their capabilities and limitations.

2.1. Turing Machines and Complexity

There is numerous literature presenting different models of computing devices. Finite automata are good for devices with a small amount of memory and relatively simple control. Pushdown automata are good for devices with unlimited memory with a stack. However it have been shown limitations of these models for even simple tasks. This makes them too restrictive for general purpose computers.

In contrast, the Turing Machine, first proposed by Alan Turing in 1936^[1], is a much more powerful model. It is essentially a finite automaton but with an unlimited and unrestricted memory and is a more accurate model of a general purpose computer since it can do everything a general purpose computer can do (although slower). Nevertheless, there are problems that Turing machines can't solve; therefore a real computer can't solve them either.

This is where we make a change in direction. We will start to examine problems that are at the threshold and beyond the theoretical limits of what is possible to compute using computers today. We will examine the following issues with the help of Turing Machine's:

- ⇒ We use the simplicity of the Turing Machine model to prove formally that there are specific problems (that. languages) that the Turing Machine cannot solve.
 - Three classes: —Recursive“ = Turing Machine can accept the strings in the language and determine if a string is not in the language. Sometimes these are called decidable“.
 - recursively enumerable" = Turing Machine can accept the strings in the language but cannot tell for certain that a string is not in the language. Sometimes these are called —partially decidable“.

- non-RE" = no Turing Machine can even recognize the members of the language. These are —non decidable.“

We then look at problems (languages) that do have Turing Machine's that accept them and always halt; that., they not only recognize the strings in the language, but they tell us when they are sure the string is not in the language.

The classes P and NP are those languages recognizable by deterministic and nondeterministic Turing Machine's, respectively, that halt within a time that is some polynomial in the input. Polynomial is as close as we can get, because real computers and different models of (deterministic) Turing Machine's can differ in their running time by a polynomial function, for example., a problem might take $O(n^2)$ time on a real computer and $O(n^6)$ time on a Turing Machine.

⇒ NP-complete problems: These are in a sense the —hardest“ problems in NP. These problems correspond to languages that are recognizable by a nondeterministic Turing Machine. However, we will also be able to show that in polynomial time we can reduce any NP-complete problem to any other problem in NP. This means that if we could prove an NP Complete problem to be solvable in polynomial time, then $P = NP$.

⇒ Some specific problems that are NP-complete: satisfiability of boolean (propositional logic) formulas, traveling salesman, etc.

2.1.1. Undecidable Problems

Given a C program (or a program in any programming language, really) that prints "hello, world" is there another program that can test if a program given as input prints "hello, world"?

This is tougher than it may sound at first glance. For some programs it is easy to determine if it prints hello world. Here is perhaps the simplest:

```
#include "stdio.h"
void main()
{
printf("hello, world\n");
}
```

It would be fairly easy to write a program to test to see if another program consisting solely of printf statements will output "hello, world". But what we want is a program that can take **any arbitrary program** and determine if it prints "hello, world". This is much more difficult. Consider the following program:

```
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==a)
long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };
main(m1,s) char *s; {
int a,b,c,d,o[k],n=(int)s;
if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
else switch(m1-=h){
case f: a=(b=(c=(d=g)<<g)<<g)<<g;
return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
case h:
for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
case g:
if(n<h)return(g);if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
```



```

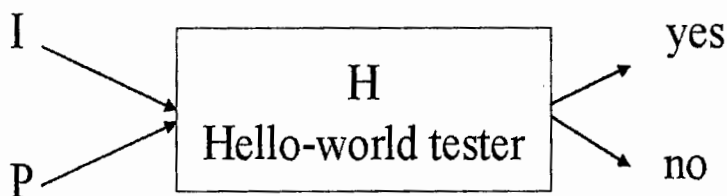
else{c='\r'\b';n-=j-g;o[f]=o[g]=g;}
if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
return(o[b-g]%n+k-h);
default: if(m1==e) main(m1-g+e+h,s+g); else *(s+g)=f;
for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
}
}

```

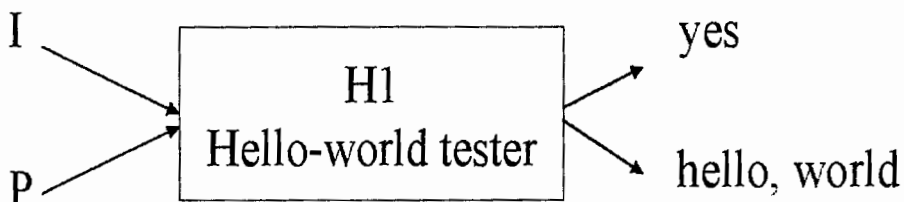
The program above, when compiled with a C compiler will actually print “hello, world”.

When we start to look at the problem of creating a program that can determine if any arbitrary program prints “hello, world” we see this program can be very difficult to create indeed. In fact, we can prove there is no C program to solve that problem (called **undecidable**) by supposing that there were such a program H, the “hello, world tester”

H takes as input a program P and an input file I for that program, and tells whether P, with input I, prints “hello, world” (by which we mean it does so as the first 13 characters) by outputting "yes" if it does, and "no" if it does not.

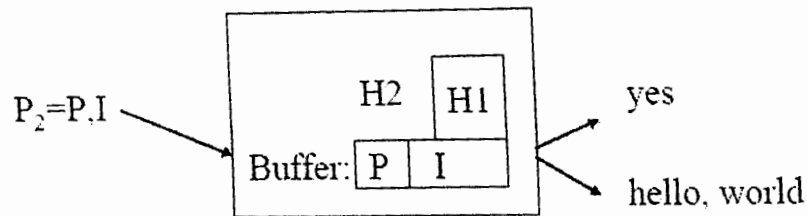


Next we modify H to a new program H1 that acts like H, but when H prints no, H1 prints “hello, world”. To do this, we need to find where "no" is printed and change the printf statement to “hello, world” instead:

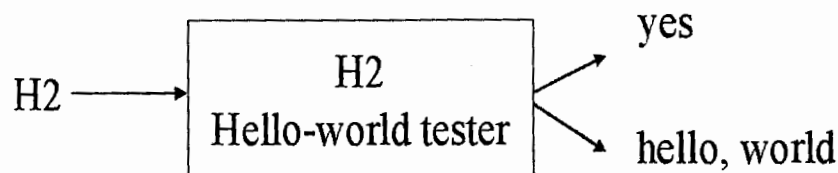


Next modify H1 to H2 . The program H2 takes only one input, P2, instead of both P and I. To do this, the new input P2 must include the data input I and the program P. The program P and data input I are all stored in a buffer in program H2. H2 then simulates

H1, but whenever H1 reads input, H2 feeds the input from the buffered copy. H2 can maintain two index pointers into the buffered data to know what current data and code should be read next:



However, H2 cannot exist. If it did, what would H2(H2) do? That is, we give H2 as input to itself:



If $H2(H2) = \text{yes}$, then H2 given H2 as input evidently does not print "hello, world". It is printing "yes". If we go back to the original program H, then $H(H2)$ outputs yes if H2 prints "hello, world". But H2 is not printing "hello, world" it is printing "yes" instead.

We have a similar contradiction if $H2(H2) = \text{"hello, world"}$. But if H2 prints "hello, world" then back with H the output should be "yes". But if the output was "yes" then we would not get the output of "hello, world".

This situation is paradoxical and we conclude that H2 cannot exist. As a result, H1 cannot exist and H cannot exist. Therefore we have contradicted the assumption that H exists and no program H can tell whether or not another arbitrary program P prints "hello world".

2.1.2. Reducing one problem to another

Once we have a single problem known to be undecidable we can determine that other problems are also undecidable by **reducing** a known undecidable problem to the new problem. We will use this same idea later when we talk about proving problems to be NP-Complete.

To use this idea, we must take a problem we know to be undecidable. Call this problem

U. Given a new problem, P, if U can be reduced to P so that P can be used to solve U, then P must also be undecidable.

It is important to show that the new problem P can be used to solve the undecidable problem U, and not vice-versa. If we show that our new problem can be solved by the undecidable problem, then it merely shows that something impossible for computers to do can solve our new problem. But it doesn't say anything about our new problem. We might just be using a really hard solution to an easy problem. But if we can show the other direction, that P can solve U, then P must be at least as hard as U, which we already know to be undecidable.

Consider the problem: Does program Q ever call function foo? We can prove this problem is undecidable.

Just as we saw with the "hello world" problem, it is easy to write a program that can determine if some programs call function foo. For example, a program with no function called foo obviously does not invoke function foo. But we could have a program that contains lots of control logic to determine whether or not function foo is invoked. This general case is much harder, and in fact undecidable. To prove this we use the reduction technique for the "hello world" problem:

1. Rename the function "foo" in program Q and all calls to that function.
2. Add a function "foo" that does nothing and is not called.
3. Modify the program to remember the first 12 characters that it prints, storing them in array A
4. Modify the program so that whenever it executes any output statement, it checks the array A to see if the 12 characters written are "hello, world" and if so, invokes function foo.

If the final program prints "hello, world" then it must also invoke function foo. Similarly, if the program does not print "hello, world" then it does not invoke foo.

Let's say that we have a program F-Test that can determine if a program calls foo. If we run F-Test on the modified program above, not only can it determine if a program calls foo, it can also determine if the program prints "hello, world". Therefore it is also

capable of determining the "hello, world" problem. But we showed that this problem is undecidable, so our program F-Test must be undecidable as well.

We'll re-visit problem reduction when we come to talk about NP-Complete problems.

2.1.3. Universal Turing Machine^[2]

Definition: A universal Turing machine (UTM) is a Turing Machine that can be fed as input a string composed of 2 parts:

1. The first is any encoded Turing Machine P, followed by a marker, say \$.
2. The second part is a string w called the data.

The UTM reads the input, and then simulates P with input w.

A general purpose Turing Machine can simulate any special purpose Turing Machine. Just store the representation of Turing Machine inside any other Turing Machine. The operations of the Turing Machine are the processes that constitute computation. Any particular Turing Machine represents one algorithm

The Universal Turing Machine a Turing Machine which implements the Turing Machine algorithm is supplied with input symbols and a set of instructions (on tape) specifying what a specific Turing Machine would do Since any algorithm can be implemented as a specific Turing Machine the UTM can perform any algorithm.

2.1.4. Turing Machines and Lambda Calculus

One of the surprising results from mathematical analysis of computation (generally credited independently to Alan Turing and Alonzo Church) is that a very simple machine of the von Neuman variety (or of any one of many similar designs, including for example Conway's ingenious Game of Life, a cellular automaton popular nowadays as a computer screen saver) is capable of computing anything which we know any way at all of computing.

For example, it is quite possible to build a machine of this sort with only one or two instructions (which perhaps respectively read two bits and store back the negated OR of them, and conditionally select one of two possible next instructions depending on the value of a given bit) which is quite capable of performing any computation we might

want. Almost all the instructions contained in modern computers are logically unnecessary: They are included only to speed up specific common operations, such as addition.

Turing's proof that a given machine is a **Universal Turing Machine**, capable of computing anything which can be computed at all, is quite simple in concept, reducing to showing that such a machine can be programmed to emulate perfectly any other machine which you can describe precisely, and hence can compute whatever the described machine could compute.

Alonzo Church^[3] worked in a more mathematical setting, developing a simple abstract **lambda calculus**, and then similarly showing that any other computational scheme could be described in terms of it.

Turing's proof has an intuitively pleasing nuts-and-bolts quality to it that made it more immediately appealing and popular: To this day we speak of "Universal Turing Machines" rather than (say) "Lambda Calculus Isomorphism". (Turing's name also produces better puns about Universal Turing Machines...)

Church's lambda calculus has however perhaps had a deeper and more significant impact:

- It provided the conceptual substrate for **denotational semantics** (perhaps the most promising, comprehensive and rigorous technique for describing what programming languages "mean").
- It inspired the field of pure-functional programming, probably the currently the promising line of research in programming language design. ("Haskell" appears likely to be to functional programming what Smalltalk was to object-oriented programming: The proof-of-concept implementation that moves the idea from the lab into mainstream consciousness.)
- It influenced the creation of Lisp, as witness the "lambda" syntax used to this day in Lisp -- although John McCarthy^[4], inventor of Lisp, strenuously denies that Lisp was ever intended to be an implementation of the lambda calculus.

Since Muq MUF is in turn based heavily on Lisp, it can be reasonably argued that anyone programming in Muq MUF owes a considerable intellectual debt to both Alan Turing's

proof, with his abstract "Turing Machines" which led to the underlying computer architecture, and to Church's proof, which led to the underlying software architecture.

Both proofs were of course at the time considered to be exercises in "pure mathematics", devoid of any practical application *grin*.

2.1.5. Computability theory

Computability theory is that part of the theory of computation dealing with which problems are solvable by algorithms (equivalently, by Turing machines), with various restrictions and extensions. Computability theory addresses four main questions:

- What problems can Turing machines solve?
 - What other systems are equivalent to Turing machines?
 - What problems require more powerful machines?
 - What problems can be solved by less powerful machines?
- ⇒ What problems can Turing machines solve?

Not all problems can be solved. An **undecidable problem** is one that cannot be solved by any algorithm, even given unbounded time and memory. Many undecidable problems are known. For example, the Entscheidungsproblem (German for "decision problem") is this: given a statement in first-order predicate calculus, decide whether it is universally valid. Church and Turing independently proved this is undecidable. The halting problem is: given a program and inputs for it, decide whether it will run forever or will eventually halt. Turing proved this is also undecidable. A computable number is a real number which can be approximated to any arbitrary degree of accuracy by an algorithm. Turing proved that almost all numbers are uncomputable. Chaitin's constant is an uncomputable number, even though it is well defined.

⇒ What other systems are equivalent to Turing machines?

The languages that are accepted by a Turing machine are exactly those that are generated by formal grammars. The lambda calculus is a way of defining functions. The functions that can be computed in the lambda calculus are exactly those that can be computed by a Turing machine. These three formulations, Turing machines, formal grammars, and the lambda calculus all look very different, and were all developed by different people. Yet

they are all equivalent, and have the same problem-solving power. This is generally taken as evidence for the Church-Turing thesis, which is the claim that our intuitive notion of an *algorithm* or an *effective procedure* is captured by the mathematical definition of a Turing machine.

Electronic computers, and even quantum computers, are exactly equivalent to Turing machines, if they have access to an unbounded supply of memory. As a corollary, all implementable programming languages are at best equivalent in power to a Turing machine (in practice, very few are less powerful). Such languages are said to be Turing-complete. Systems equivalent to a Turing machine include:

- Turing machine with several tapes
- Turing machine with a 2-dimensional "tape" (an infinite number of linear tapes)
- Turing machine with a limited number of states and tape symbols
- States \times symbols can be any of 2×18 , 3×10 , 4×6 , 5×5 , 7×4 , 10×3 , 22×2
- Finite state machine with 2 stacks
- Finite state machine with 2 counters
- Formal grammar
- Post system
- Lambda calculus
- Partial recursive functions
- Almost any modern programming language (when given unlimited memory), including:
 - A language with 1 instruction, 1 parameter (see OISC and URISC here)
 - A language with 8 instructions, no parameters (see BrainFuck)
 - Wang tiles
 - Recurrent neural network (finite-precision inputs/outputs/weights, infinite-precision signals initialized to zero)

- Cellular automaton, including:
- Conway's Game of Life
- Cellular automaton with just 1 dimension, 2 states, 3 cells per neighborhood (for example. rule 110)
- Non-deterministic Turing machine
- Probabilistic Turing Machine
- Quantum computer

The last three examples use a slightly different definition of *accepting* a language. They are said to accept a string if *any* computation accepts (for non-deterministic), or *most* computations accept (for probabilistic and quantum). Given these definitions, those machines have the same power as a Turing machine for accepting languages.

⇒ What problems require more powerful machines?

Sometimes machines are considered that have more power than a Turing machine. For example, an oracle machine uses a black box that can compute some particular function that might not be possible for an ordinary Turing machine. The theory of real computation deals with machines using infinite-precision real numbers. Within this theory, it is possible to prove interesting statements such as "the complement of the Mandelbrot set is only partially decidable". For other such powerful machines, see super-Turing computation and hypercomputation.

⇒ What problems can be solved by less powerful machines?

The Chomsky hierarchy defines those languages that can be accepted by four classes of algorithms. They all assume a machine consisting of a non-deterministic finite state machine combined with some form of memory. If the memory is an infinite tape, then it has the full power of a Turing machine, and can accept exactly those languages that are generated by unrestricted grammars. If it is given only an amount of memory proportional to the size of the input, then it can recognize exactly those languages generated by context-sensitive grammars. If it is given only a stack as its memory, then it can recognize exactly those languages generated by context-free grammars. If it is given

no additional memory at all then it can accept exactly those languages generated by regular grammars.

Other restrictions on memory, or time, or other resources have often been considered. The results for those restrictions are usually considered part of complexity theory rather than computability theory.

2.1.6. Halting Problem/ function

Turing used the Universal Turing Machine idea to prove that in any general algorithm there were propositions which could not be decided. Basic method was to assume Decidability was true and look for a paradox, for example. The Liar's Paradox "This statement is false."

- If the statement is true, its content makes it false
 - If it is false, its content makes it true
- Turing's paradox arose from The Halting A Turing Machine is just a formal model of what is an algorithm and a Universal Turing Machine is an algorithm for executing algorithms. Any Turing Machine (including the UTM) is said to halt if the algorithm terminates. They only stop when the computation is finished.

Suppose we define a Universal Turing Machine which halts if The Turing Machine it is executing does not halt

What happens when we ask the UTM to execute itself, its own algorithm?

- If the UTM halts, then the algorithm does not halt, that. the UTM does not halt.
- If the UTM does not halt, then the algorithm does halt, that. the UTM does halt.

The above two statements contradict with each other as we have shown in the part 'undecidable problems'. This problem is known as halting problem. So Turing Machine cannot solve the Halting Problem.

So using Turing's definition of a "definite method" there exists a problem which cannot be solved^[5]. There is no Turing Machine that can accept any its own encoded Turing Machine. Let us give an elaborated explanation.

Basic Idea:

- Define halting function $H(P,w)$, where P is encoding of program (that., encoded Turing machine) w is intended input for P .
- Let $H(P,w) = \text{yes}$ if P halts on input w .
- Let $H(P,w) = \text{no}$ if P does not halt on input w .
- Assume that a program computing $H(P,w)$ exists.
- Construct a program $Q(P)$ with input P :
- $x = H(P, P)$
- While $x = \text{yes}$, goto step 2.
- Now run program Q with input $P = Q$.
- Suppose $Q(Q)$ halts. Then $H(Q,Q) = \text{yes}$, but Q is stuck in infinite loop and so it doesn't halt.
- Suppose that $Q(Q)$ doesn't halt. Then $H(Q,Q) = \text{no}$, while in fact $Q(Q)$ halts.
- Therefore $H(P,w)$ cannot exist.

Alan Turing solved a fundamental problem in mathematics. Defined what it is to be an algorithm and explored the limits of computability provided an existence proof for the general purpose digital computer and he didn't stop there. The Halting Problem and its role in answering Hilbert's question is a key event in the history of computing.

Programming languages provide a framework for expressing algorithms. They provide a means for giving ordered instructions to a machine capable of executing a program. Let's put it quite simply: they are the languages for writing programs. The actual details of how modern programming systems work are not so simple, however, so let's elevate them for a brief moment: programming languages and the systems that support them mediate between you and the machine.

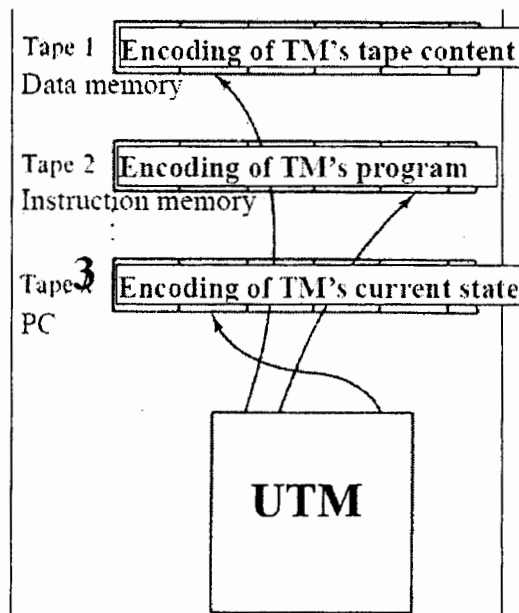
As such, it seems difficult to consider programming without having a particular machine or mechanism in mind. It is certainly the case that a programming language is closely married with a programming system, some sort of computing system that you as its

programmer must reason about in order to direct the execution of your programs. As a computing entity, the programming language system does present some notion of computation. However, it turns out that the programming system's model of computation need not be the hardware itself. Instead, modern programming languages provide a layer of abstraction between a programmer's algorithmic concerns and the physical constraints of their hardware and operating system.

Despite being an abstraction of the physical machine, programming languages do allow you to program with some precisely defined mechanism in mind. After all, a programmer needs to have a clear understanding of his program's effect. Let's consider what makes this layer of abstraction possible. First, we are armed with the notion, first introduced by Alan Turing, that there exist abstract computing models that provide universal computation, for example, the Turing machine model. Combine this fact with the knowledge that the hardware and operating system form a physical computing machine that closely emulates a universal computing device. Finally, let's make the conceptual leap that a computing device really is just another algorithm. The practical effect of all this is that the machine model of a programming language can be an abstract one. The abstract machine model that a programming language system presents for your programs does not have to be the (concrete) computer on your desk.

- A programming system presents a programmer with a universal computing device for executing any algorithm expressed in its programming language,
- A universal computing device, like that presented by a programming system, has an (expressible) algorithm (expressible in, say, a machine language) recursively enumerable language: if any language is accepted only by a specific turing machine that is called recursively enumerable language. These are closed under union concatenation and kleen closure.

2.1.7. Encoding of Turing Machine's program



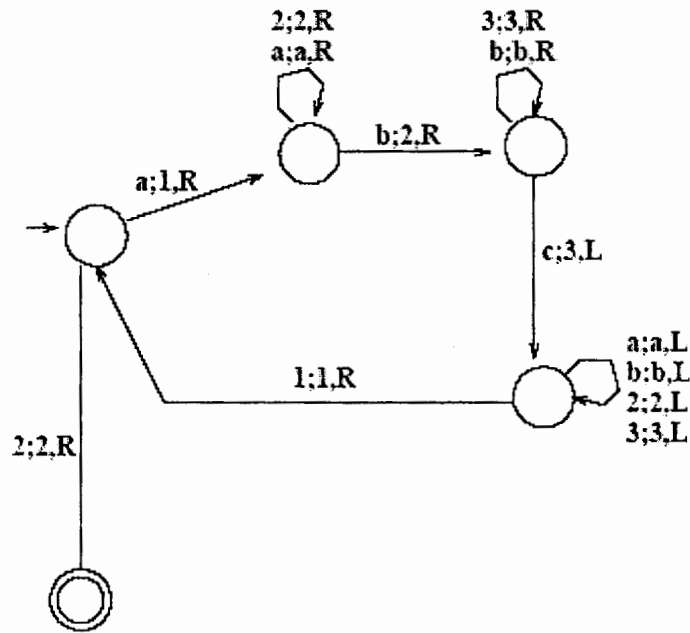
Encode the three ingredients of Turing Machine using three tapes of a UTM and follow the steps to run a Turing Machine in UTM.

- UTM (simulates Turing Machine)
- read tape 1
- read tape 3
- consult tape 2 for what to do
- write tape 1 if necessary
- move head 1
- write tape 3

2.1.8. Church-Turing Thesis

The Turing machine models a human being solving a problem in an algorithmic way. The Turing machine is a general model of computation.

⇒ Church-Turing Thesis: A language can be solved by an algorithm if and only if it can be accepted by a Turing machine that terminates on every input.



Finite and pushdown automata are too restricted to serve as models of general-purpose computers.

A Turing machine is similar to a finite automaton but with an unlimited and unrestricted memory an infinite tape. It has a tape head that can read and write symbols and move around on the tape.

A Turing machine can do everything that a real computer (as we know it) can do. Nonetheless, there are problems that no Turing machines, and hence no real computers, can solve.

⇒ Recursively Enumerable Languages

Definition: A language L over an alphabet Σ is called recursively enumerable if there is a Turing Machine that accepts every word in L and either rejects (crashes) or loops forever for every word in L^c ; that is,

$\text{accept}(T) = L$,

$\text{reject}(T) + \text{loop}(T) = L^c$.

In other words, the class of languages that are accepted by a Turing Machine is exactly those languages that are recursively enumerable.

⇒ Definition: A language L over an alphabet Σ is called recursive if there is a Turing Machine that accepts every word in L and rejects every word in L^c ; that is,

$\text{accept}(T) = L$,

$\text{reject}(T) = L^c$,

$\text{loop}(T) = \emptyset$;

2.1.9. Programming Techniques for Turing Machines

Sometimes, it helps to see a Turing Machine as something more complex than it actually is. Those tricks do not change Turing machines as such; it is only the interpretation that changes.

We can store data elements in states. This is implemented as complex state names. The original state holding state q , and data elements A and B is named $[q; A; B]$. This technique can be used for example, to remember some elements of the read data.

We can divide the tape into tracks. The symbol on the original tape holding symbols A , B , and C from three tracks of tape is named $[A; B; C]$. This technique can be used for example, for marking elements of data, that, track 1 can be used for actual data, and track 2 for marks.

We can extract parts of a Turing machine and treat them as subroutines. If a subroutine is called from more than one place, its code needs to be copied to a new set of states.

Nature of Turing machines,

- Can match power of any sophisticated automata
- Can match power of any real special purpose computer
- Can be made general to simulate any special purpose Turing Machine
- Therefore can match power of any real general purpose computer
- In fact, it can match the power of any computation methods

T-168

Here an example is presented of what can be done using Turing Machine. No CFG generates $L = \{anbncn\}$

But there is a UG for $\{anbncn\}$

$S \rightarrow W D Z \mid \square$ 1. start

$D \rightarrow A B C D \mid A B C$ 2. n of each

$C A \rightarrow A C$ 3. sorting

$C B \rightarrow B C$ is

$B A \rightarrow A B$ permitted

$W A \rightarrow a W$ 4. traverse,

$W B \rightarrow b X$ downcase,

$X B \rightarrow b X$ and

$X C \rightarrow c Y$ check

$Y C \rightarrow c Y$ the sort

$Y Z \rightarrow \square$ 5. check step 4

A recursive (decidable) language L is a language for which a Turing machine that accepts it and halts on all inputs exists: If $w \in L$ then the Turing machine halts in state q accept; if $w \notin L$ then the Turing machine halts in state q reject.

- A language L is a recursively enumerable (r.e.) language if a Turing machine that accepts it, exists: For all $w \in L$, the Turing machine halts in q accept; for all $w \notin L$ the Turing machine either halts in state q reject or never halts.

2.1.10. Countability and Diagonalization

The notion of countability was described by Georg Cantor^[6] in 1873. If we have two infinite sets, how can we tell whether one is larger than the other? Obviously we can't start counting with each element or we will be counting forever Cantor's solution is to make a **correspondence** to the set of natural numbers.

A correspondence is a function $f: A \rightarrow B$ that is one-to-one from A to B . Every element of A maps to a unique element of B , and each element of B has a unique element of A mapping to it.

Example: The set of natural numbers, $N = \{ 1, 2, 3, 4, \dots \}$.

The set of even numbers, $E = \{ 2, 4, 6, \dots \}$

It might seem that N is bigger than E , since we have values in N that are not in E (by one measure, we have twice as many.) However, using Cantor's definition of size both sets have the same size:

$$N \quad 1 \quad E = F(N) \quad 2$$

$$2 \quad 4$$

$$3 \quad 6$$

etc.

For every number in N , there is a corresponding value in E .

Definition: A set is **countable** if it is finite or if it has the same size as the natural numbers.

For example, as we saw above, E is countable.

Example: The set of positive rational numbers, Q , is countable. That is,

$$Q = \{ m/n \mid m, n \in N \}.$$

To show that this is countable, we need to make a 1:1 correspondence between the rational numbers and the natural numbers. We must make sure that each rational number is paired with one and only one natural number. Consider the mapping as shown in the matrix below:

| | | | | | |
|------------|------------|------------|------------|------------|------------|
| 1/1 | 1/2 | 1/3 | 1/4 | 1/5 | 1/6 |
| 2/1 | 2/2 | 2/3 | 2/4 | 2/5 | 2/6 |
| 3/1 | 3/2 | 3/3 | 3/4 | 3/5 | 3/6 |
| 4/1 | 4/2 | 4/3 | 4/4 | 4/5 | 4/6 |
| 5/1 | 5/2 | 5/3 | 5/4 | 5/5 | 5/6 |
| 6/1 | 6/2 | 6/3 | 6/4 | 6/5 | 6/6 |

If we started in the first row and just worked our way to the right, we could assign each rational number to a natural number:

$$1/1 \rightarrow 1$$

$$1/2 \rightarrow 2$$

$$1/3 \rightarrow 3$$

etc.

However, we would never make an assignment for values like $2/1$.

The solution is to traverse the diagonals of the matrix. We assign values as:

$$1/1 \rightarrow 1$$

$$2/1 \rightarrow 2$$

$$1/2 \rightarrow 3$$

$$3/1 \rightarrow 4$$

$$2/2 \rightarrow \text{skipped}$$

$$1/3 \rightarrow 5$$

etc.

Notice how we had to skip any elements that would cause a repeat. Continuing in this way, we can obtain a list for all elements of Q and therefore Q is countable.

2.1.11. Uncountable Sets

Since we have seen infinite sets that are countable, it might seem like any infinite set is countable. However, this is not the case.

Example: The set of real numbers, R , is not countable.

Suppose that R is countable. Then there is a correspondence between members of R and members of N . The following table shows some hypothetical correspondences:

| N | R |
|---|----------|
| 1 | 3.14159. |
| 2 | 55.555. |
| 3 | 0.12345. |
| 4 | 0.50000. |

Given such a table, we can construct a value x that is in R but that has no pairing with a member in N .

To construct x , we ensure it has a digit that is different from all values listed in the table. We can do this by starting with the first fractional digit of 3.14159. This is the digit 1. So we pick something different, say we pick 4. Then we move to the second value. The second fractional digit of 55.555 is 5. $\underline{3}45$ is $\bar{3}$. So we pick something different, say 6. The third fractional digit of 0.12345 is 3. So we pick something different, say 1. We can continue in this way, to construct $x = 0.4612 \dots$

The value x is in R . However, we know that x has no corresponding value in N because it differs from n in N by the n th fractional digit. Since x has no corresponding value in N , the set of real numbers is not countable.

This result is important because it tells us something our Turing Machine's and computers cannot compute. It is impossible to exactly compute the real numbers \mathbb{R} we must settle for something else, for example, a less precise answer or computation.

Corollary: There exist languages that are not recognizable by a Turing Machine.

First, the set of all Turing machines is countable. We can show this by first observing that the set of all strings Σ^* is countable, for a finite alphabet Σ . With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, all strings of length 1, all strings of length 2, etc.

The set of all Turing Machine is countable because each Turing Machine may be encoded by a string s . This string encodes the finite control of the Turing Machine. If we omit those strings that are not valid Turing Machine, then we can obtain a list of all Turing Machine.

To show that the set of languages is not countable, observe that the set of all infinite binary sequences is uncountable. The proof for this is identical to the proof we used to show that the set of real numbers is uncountable.

The set of all languages has a correspondence to the set of all infinite binary sequences. (For the alphabet of $\{0,1\}$ they are the same. For languages with more than two symbols, we use multiple bits to represent the symbols). Therefore, the set of all languages is also uncountable and we conclude there are languages that are not recognized by any Turing machine.

In fact, there are many more languages that are not recognized by Turing Machine's than languages that are recognized by Turing Machine's. Fortunately, most of the time we don't care about these other languages, but are only interested in ones that Turing Machine's can recognize.

The above property of enumerability is one reason why we call languages recognizable by Turing Machine's to be *recursively enumerable*. The "—recursion" part is historical, from using recursion to implement many of these problems (and therefore meaning that this problem is decidable, or recognizable by a Turing Machine).

2.1.12. The Decision Problem

David Hilbert^[7] presented a list of mathematical problems in 1900^[8]. Among the problems were the foundational questions of whether the axioms of arithmetic are consistent, and whether there is a number system in cardinality between the rational numbers and the continuum of real numbers. In^[9] Hilbert posed another group of problems. This time dealing with the consistency, completeness and independence of the axioms of a logical system in general, as well as the problem of deciding whether a given statement is valid within a logical system. The solutions to these problems, in particular Kurt Gödel's^{[10][11]} demonstration of the incompleteness of arithmetic with the existence of statements that are not provable (as true or false), had profound consequences for mathematics, and brought to the fore mathematical logic as a separate field of study. In this article, however, we deal primarily with the decision problem, which, to quote Hilbert and Ackermann^[12], can be stated as:

. . . there emerges the fundamental importance of determining whether or not a given formula of the predicate calculus is universally valid. . . . A formula . . . is called *satisfiable* if the sentential variables can be replaced with the values truth and falsehood . . . in such a way that the formula [becomes] a true sentence. . . . It is customary to refer to the equivalent problems of *universal validity* and *satisfiability* by the common name of the *decision problem*.

Following Gödel's results, the decision problem remained, although it must be reinterpreted as meaning whether there is a procedure by which a given proposition can be determined to be either "provable" or "unprovable". In Alonzo Church^[13], formulated this problem as: "The *decision problem* of a logistic system is the problem to find an effective procedure or algorithm, a *decision procedure*, by which for an arbitrary well-formed formula of the system, it is possible to determine whether or not it is a theorem" To be sure, Church^[14] proves that the decision problem has no solution, although it is the algorithmic character of Turing's solution that is pivotal to the logical underpinnings of the programmable computer. Moreover, the simplicity of a Turing machine provides a degree of accessibility to the subject ideal for a first course in logic or discrete mathematics.

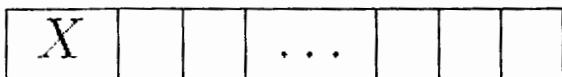
Briefly a Turing machine, M , is a device which prints a sequence of zeroes and ones on a tape based on (i) the figure currently being scanned on the tape, and (ii) a set of instructions. Moreover, Turing describes the logical construction of a universal computing machine, U , which accepts the set of instructions of a given machine M in some standard form, and then outputs the same sequence as M . Applying a machine U to another machine M is denoted as $U(M)$ for this exposition. It follows from Turing's paper (discussed later) that if the decision problem has a solution, then there is a machine D which accepts the set of instructions of another machine M and decides whether M prints a finite or an infinite number of symbols on the tape. Determining whether a machine terminates in a finite number of executable steps is today known as the halting problem in computer science. If the decision problem, and hence the halting problem has a solution, then a new machine T can be defined so that $T(M)$ halts if M does not halt, and $T(M)$ does not halt if M halts. By considering the behavior of $T(T)$, we conclude that T halts and T does not halt, a contradiction, from which it follows that the decision problem has no solution.

2.1.13. Induction and Recursion

The logic behind the modern programmable computer owes much to Turing's "computing machines," discussed in the first project, which the reader should review. Since the state of the machine, or m -configuration as called by Turing, can be altered according to the symbol being scanned, the operation of the machine can be changed depending on what symbols have been written on the tape, and affords the machine a degree of programmability. The program consists of the list of configurations of the machine and its behavior for each configuration. Turing's description of his machine, however, did not include memory in its modern usage for computers, and symbols read on the tape could not be stored in any separate device. Using a brilliant design feature for the tape, Turing achieves a limited type of memory for the machine, which allows it to compute many arithmetic operations. The numbers needed for a calculation are printed on every other square of the tape, while the squares between these are used as "rough notes to 'assist the memory.' It will only be these rough notes which will be liable to erasure".

Turing continues: “The convention of writing the figures only on alternate squares is very useful: I shall always make use of it. I shall call the one sequence of alternate squares F-squares, and the other sequence E-squares. The symbols on E-squares will be liable to erasure. The symbols on F-squares form a continuous sequence. ... There is no need to have more than one E-square between each pair of F-squares: an apparent need of more E-squares can be satisfied by having a sufficiently rich variety of symbols capable of being printed on E-squares”.

Let’s examine the Englishman’s use of these two types of squares. Determine the



output of the following Turing machine, which begins with the tape and the scanner at the far left, reading the symbol X.

| Configuration | | Behavior | |
|---------------|--------|---------------------------|-----------------|
| m-config. | symbol | operation | final m-config. |
| <i>a</i> | X | R | <i>a</i> |
| <i>a</i> | 1 | R, R | <i>a</i> |
| <i>a</i> | blank | P(1),R,R,P(1), R, R, P(0) | <i>b</i> |
| <i>b</i> | X | E, R | <i>c</i> |
| <i>b</i> | other | L | <i>b</i> |
| <i>c</i> | 0 | R, P(X), R | <i>a</i> |
| <i>c</i> | 1 | R, P(X), R | <i>d</i> |
| <i>d</i> | 0 | R, R | <i>e</i> |
| <i>d</i> | other | R, R | <i>d</i> |
| <i>e</i> | blank | P(1) | <i>b</i> |
| <i>e</i> | other | R, R | <i>e</i> |

Recall the meaning of the following symbols used for operations.

- R: Move one position to the right.
- L: Move one position to the left.
- E: Erase the currently scanned square
- P(): Print whatever is in parentheses in the current square.

2.1.14. The Decision Problem In A.M.Turing's Words or Das Entscheidungsproblem

Alan Turing's^[1] proved most influential not only for mathematical logic, but also for the development of the programmable computer, and together with work of Alonzo Church^[3] inaugurated a new field of study, known today as computability. Recall that Turing's original motivation for writing the paper was to answer the decision problem of David Hilbert^[7], which asked whether there is a standard procedure that can be applied to decide whether an arbitrary statement (within some system of logic) is provable. A previous project examined the construction of Turing's "universal computing machine," which accepts the instructions of any other machine M in standard form, and then outputs the same sequence as M . The concept of a universal machine has evolved into what now is known as a compiler or interpreter in computer science, and is indispensable for the processing of any programming language. The question then arises, does the universal computing machine provide a solution to the decision problem? The universal machine is the standard procedure for answering all questions that can in turn be phrased in terms of a computer program.

First, study the following excerpts from Turing's paper :

Automatic machines: If at each stage the motion of a machine is completely determined by the configuration, we shall call the machine an "automatic machine" (or a-machine). ...

⇒ Computing machines:

If an a-machine prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine. If the machine is supplied with a blank tape and set in motion, starting from the correct initial m-configuration, the subsequence of symbols printed by it which are of the first kind will be called the sequence computed by the machine. ...

⇒ Circular and circle-free machines.

If a computing machine never writes down more than a finite number of symbols of the first kind, it will be called circular. Otherwise it is said to be circle-free. ...

A machine will be circular if it reaches a configuration from which there is no possible move, or if it goes on moving and possibly printing symbols of the second kind, but cannot print any more symbols of the first kind.

⇒ Computable sequences and numbers.

A sequence is said to be computable if it can be computed by a circle-free machine. A number is computable if it differs by an integer from the number computed by a circle-free machine.

2.2. Equivalence between Turing Machine and computer

⇒ The computing model on Turing machine described in^[15] presents 5 tape Turing machine mapping some assembly function features.

First tape contains the main memory program. The program is in Assembly language format containing both address and contents. The marker * and # are used as end marker of address and contents. Second tape is instruction counter. The third tape contains memory address. Fourth tape contains computer input file and fifth tape is for temporary use.

The functionality is simple. Fetch for the instruction number in the first tape that match the instruction counter.

When the instruction address is found examine the value.

The value contains instruction in the first two three bits and remaining contains address that are involved in action.

If the instruction requires value of some address then the value will be part of instruction. Copy the instruction in second tape and execute it.

Details of what size of target address area will be instruction and data and how to handle them are not discussed. It cannot handle more than binary operations.

⇒ The computing model on Turing machine described in^[16] Turing Machine implementation involves a set of named states, which are specified in the building stage of the machine. During Execution the machine will always be in one state. Here we discuss some of its portions those are related to us.

This paper discusses the following two as previous work.

1. Cheran Soft Visual Turing

<http://www.cheransoft.com/vturing/index.html>

2. Various applets on the web

For example, <http://www.igs.net/~tril/tm/tm.html>

The aims of the program attempt to keep the solid foundations laid down by previous work while building up on them and improving them. The core aims are discussed below.

- The ability to program Turing Machines as truth tables, state diagrams and simple instructions and to convert between all three formats.
- The ability to convert instructions from pseudo code to Turing Machine instructions
- The ability to simulate Turing Machines in the formats above. The Simulator must have a Graphical User Interface and must give real-time visual feedback to the user, with a possibility to pause on a state for debugging.
- The ability to save Turing Machines and re-use them later.

In this paper states, symbols, instructions are saved in complex data structures (discussed in the paper) moves (rules) are also saved in complex data structures. Following are some of the issues those we have focused on this paper.

➤ Variable storage issues

The issue of variables is more complex than other ones. The machine to emulate a computer would contain 4 tapes. One tape for user defined variables, one tape for system variables, one tape for the call stack and one tape for processing.

First, we will look at the method used to actually store variables on the variable tapes.

Each variable is written on the tape and separated by a unique variable symbol. For example, if three variables a, b and c with values “hello”, “goodbye” and “end” respectively were to be stored on a tape, the tape would look like the one in figure 14.



Figure 2.1 Storing variables on a tape

Notice how the variable symbols are given special markers so the machine does not confuse them to be part of the actual variables. But it is not discussed in the paper what special technique used for variable markers. (Any preamble, special symbol etc.). It is worth to mention that the variable markers are implementation specific.

The storage of private variable would include an extra step to this storage model. For every scope block, there would have to be a start and end marker, and within those two markers, the variables in that scope would be stored.

➤ Variable assignment

Assigning a variable would require the following steps:

- On the relevant tape (system or user defined) find the variable marker if it does not exist yet, create it at the end. When scanning the tape, skip any blocks which are out of our scope.
- Copy the value to be assigned straight after the marker character
- If there isn't enough space between the marker character and the start of the next variable, shift the tape contents using a simple shifting machine.

➤ Variable retrieval

Retrieving a variable would require the following steps:

- On the relevant tape (system or user defined) find the variable marker if it does not exist yet, return an error
- Copy the value between the marker found to the next marker (the variable) from the variable tape to the desired location (usually the processing tape)

➤ Loops

Loops are fairly easy to convert to a tree using only the features specified above.

- First, create a system variable to use as a loop counter (unless an existing variable has already been specified in a For loop declaration). In both cases, initialise it with the initial value needed.
- Run the code within the loop repeatedly, incrementing the counter each time.
- Each time the code is about to run, compare the target value and the loop counter value using the code specified in section 8.5 above. If they match, exit the loop.

➤ **Conditional blocks**

With conditional blocks, each expression of the condition would first need to be split into the seven basic commands described above. That done, the different expressions would be evaluated and equated. The code within the block would then be run depending on the result of the equation.

➤ **Function calling**

Function calling would be fairly simple, assuming the binary encoding of the Turing Machine of the function was stored in a normal system variable. The steps to calling would be:

- Write the current tape configuration of the processing tape to the call stack tape.
- Run a universal Turing Machine on the binary encoding of the Turing Machine of the function for any variables needed by the function, create a new scope block. Make the first variable in the new scope block the return value for the function.
- When the function is done running, read the tape configuration from the call stack tape and restore it. Get the return value from the first variable in the function scope block and then delete the scope block created.

➤ **Arithmetic operations involving brackets and indices**

Arithmetic operations involving brackets are simple all that needs to be done is to split the bracketed operation into the seven basic operations specified above and store the results of each of the brackets in a system variable. All operations on that bracket

can then be done on the system variable it returned.

Supporting indices is also simple all that needs to be done is a series of multiplication on the relevant number or variable.

The paper does not discuss the moves in detail and many of its feature are implementation specific. There is no detail of solving equation. It left discursion just after how to perform add, subtract, multiply and divide etc. No details of scope handling is discussed.

Chapter 3

Our Model

3. Our Model

3.1. Aims of our research.

The aim of our research was:

- Expressing all the functionalities of Turing Machine in completely predefined moves. These moves are specified in the rules portion of our paper.
- Moves (rules) are built in the Turing Machine containing the symbols and states. So there is no need to dig down through all those complex data structures to go to the next configuration.
- The symbols used in the paper to represent states and tape symbols may be changed in implementation. But strictly they should have a one to one mapping those symbols specified in the research paper.
- Each block of Turing Machine contains one token of higher level language. So Turing Machine will not scan through all the individual parts of token. Instead it will read the whole token at once.
- The input (source code) given to Turing Machine must not be converted to any other form for execution.
- In addition to above point, the syntax of source code must be same as pseudo code or syntax of third generation language (higher level language). So there is no need to convert from pseudo code to Turing Machine input code.
- Reference to any variable either to read or update must be done in one travel in tape. One travel means going in reverse direction only once and then coming back only once to the original position. That is the travel will change direction only once in the whole travel, let alone reading some value in the tape and then fetching some another value in another tape for one variable call.
- As the syntax is same to that of higher level language syntax, so the execution will also be sequentially token by token in one direction. Only in variable

referencing, loop, function calling there is a turning point in the moves.

- It is worth to mention that some previous models have used program counter or other jump mechanism in their model. But they have implemented that by initializing some block of tape and then searching that value in another tape one by one in all boxes until the value is found. We have not used this. Loop and function call is implemented using such techniques that jump instruction is not needed.
- Expressions are solved by converting them to postfix form.
- Send receive will be implemented.
- Least possible temporary variable should be used.
- So there is no need to convert from pseudo code to Turing Machine input code.

In modern computing model, different parts of it are again a individual computing model having their own addressable identities. Therefore, we may call them addressable computing unit.

Our proposed model is a compound Turing Machine of many simple Turing Machines. We now describe systematically the characteristics of our model.

There are many kinds of physical computing devices, such as mini, micro, mainframe, super scalar etc. These may be general purpose and special purpose. Computing models may be physical and virtual (as software on other physical model). But the virtual models can also be described as a big algorithm, collection / sequence of small physically implemented algorithms.

In spite of these facts, there is no fundamental difference between data processing, storage and communication between any kinds of computing model mentioned above.[5]

In a fully distributed environment, there can be global addressing systems to address a small component of a native node participating in the whole system.[6] There may be many mechanism adopted for this purpose.

Our model describes using Turing Machine:

A computer with

- Infinity long sequence of words each with an address
- Infinity long address space.
- Program of computer is stored in some word of memory.

Each word or box of tape represents a simple token of Higher-level language (Unlike the models described in existing books that refers instruction of a memory as in the machine or assembly language.)

3.2. Overall working in brief of our Turing Machine.

The main bottom-line idea of this implementation is, for each occurrence of instructions the Turing Machine goes to a specific state and after performing the execution of that instruction it goes to the next.

The structure of the Turing Machine is very simple switch statement nested by a while loop. The switch statement checks the current configuration of the machine (current state and symbols / tokens at current tape box.) and moves the machine to new configuration. While loop is terminated by checking Turing Machine at final state.

While (current state is not final state)

```

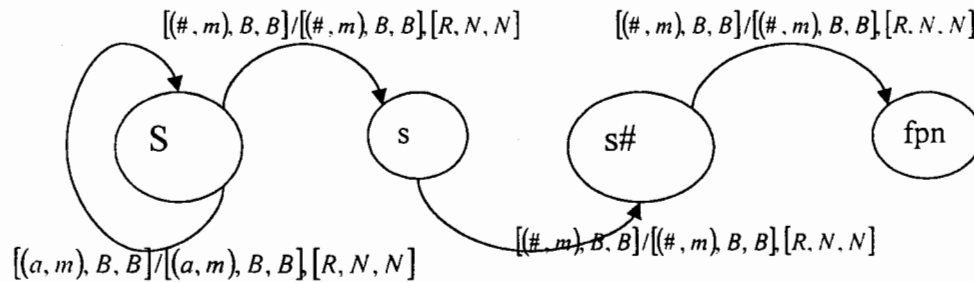
{
  switch( current state)
  {
    Check and move it to next configuration.
  }
}

```

All the instructions are implemented as moves of Turing Machine.

3.3. Turing machine executing third generation language instructions.

The machine, given in figure 3.1 starts at 'S' state reading the symbols at the program track of program tape from left to right and fetches the keyword 'hash'. On reading it go to 's' state and after that goes to their respective states and executes the instructions. Each occupies one block of tape, as all of these are tokens. On reading two consecutive 'hash', program ends. Here we discuss some features of three generation language executed in



this Turing machine.

Figure3.1 Starting of machine, passing comments, starting main program, finishing.

3.4. Variable definition

Variable can be defined before the starting of main program as well as inside the program. The keywords 'VDL', 'VDR', 'VDM', are used for the tape to detect the left, right and middle point of the variable definition. Details of these are discussed before.

As shown in figure 3.2, whenever the tape head sense a 'VDL' in s state, it assumes that there is a variable definition at right so it goes to Variable definition state (V_d) and remains inactive until it finds a 'VDL' indicating the end of the variable definition. On reaching the end, it returns to 's' state.

If any defined variable is used as global variable inside some function it must be defined before the function first call.

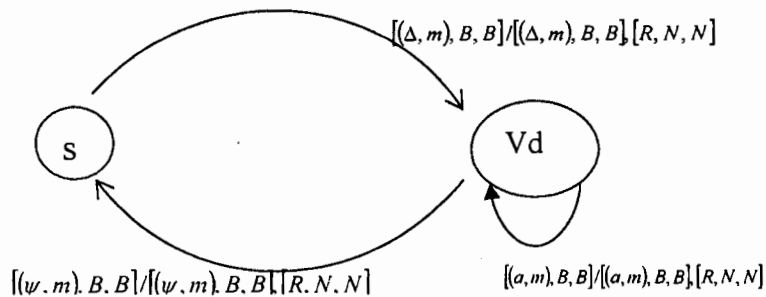


Figure 3.2 *passing a variable definition at 's' state*

3.5. Variable reading:

When TM read a variable, it first

- Push (write in the third tape and move the third tape pointer right one box.) the return point. (The current box number).
- Push the return point.
- Push R for read or W for write (discussed later).
- Push the variable name.

Let us discuss the steps in detail as shown in figure 3.3:

When it reads 'VCL' in 'e' state it assumes that there is a variable call at right so, it switch to 'vr' state and goes to the last point of variable.

'vrpr' state: sensing 'VCR' it switch to 'vrpr'. then it push the next box address as return address then switching to 'vpn' state, it push 'R', indicating that it wants to read a value of the variable and switch to 'vfv' state to find variable.

'vfv' state: in this state it keeps moving the tape 1 head moving left. On every findings of 'VDM' it switch to 'vfn' state. On this state it moves left and mach the variable name with the tape 3 name (previously pushed the required variable name). if not matched it again switch to 'vfv' state. On finding the name it switch to 'vn' state.

'vn' state: In this state it first deletes the pushed name starts moving right two boxes and if the value is other then 'VCL' it is the value. Otherwise it is the reference to another variable so it pushes the new name (the variable name at the reference) and again switch to 'vfv' state and works in the previous way. When it finds it, it pops the variable name

from the third tape. Then it moves the 3rd tape head left. There it checks whether it was to be read or written (whether there was 'R' or 'W').

- If it was to be read, delete the 'R' from third tape, move third tape head left, then it writes the value to the postfix equation and goes to 'vegr' state to go to the previous position. The expressions are solved by converting them to postfix form so the value should be written to postfix equation.
- Otherwise if it was to be written, delete the 'W' from third tape, move third tape head left, then it moves the value from postfix equation to the value part of the definition and erase from the postfix equation and goes to 'vwgr' state to go to the previous position.
- Now in both 'vegr' or 'vwgr' states the third tape value is the memory location of the next execution starting point from where the variable was referenced. The state is also changed to state to go the place from where the variable was called. In case of variable writing back at moves second tape head right one box and go to 'vsgr' state and start moving right to the next execution starting point from where the variable was referenced.

3.6. Variable updating:

In any statement the Lvalue is calculated and updated. In this case no variable is updated at the right hand side of the assignment operator. So when we finish calculating the right hand side of the assignment operator (discussed at evaluation section), we fetch the variable by the previous way and assign the value there at the value part. This will be discussed in the statement part. How to write variable is already discussed in the previous part.

Variable length is each variable will occupy one block. In this TM, address space, variable value length, function names and all other reserve word, keyword length all is assumed to occupy one block as those are tokens of the native code of Turing Machine.

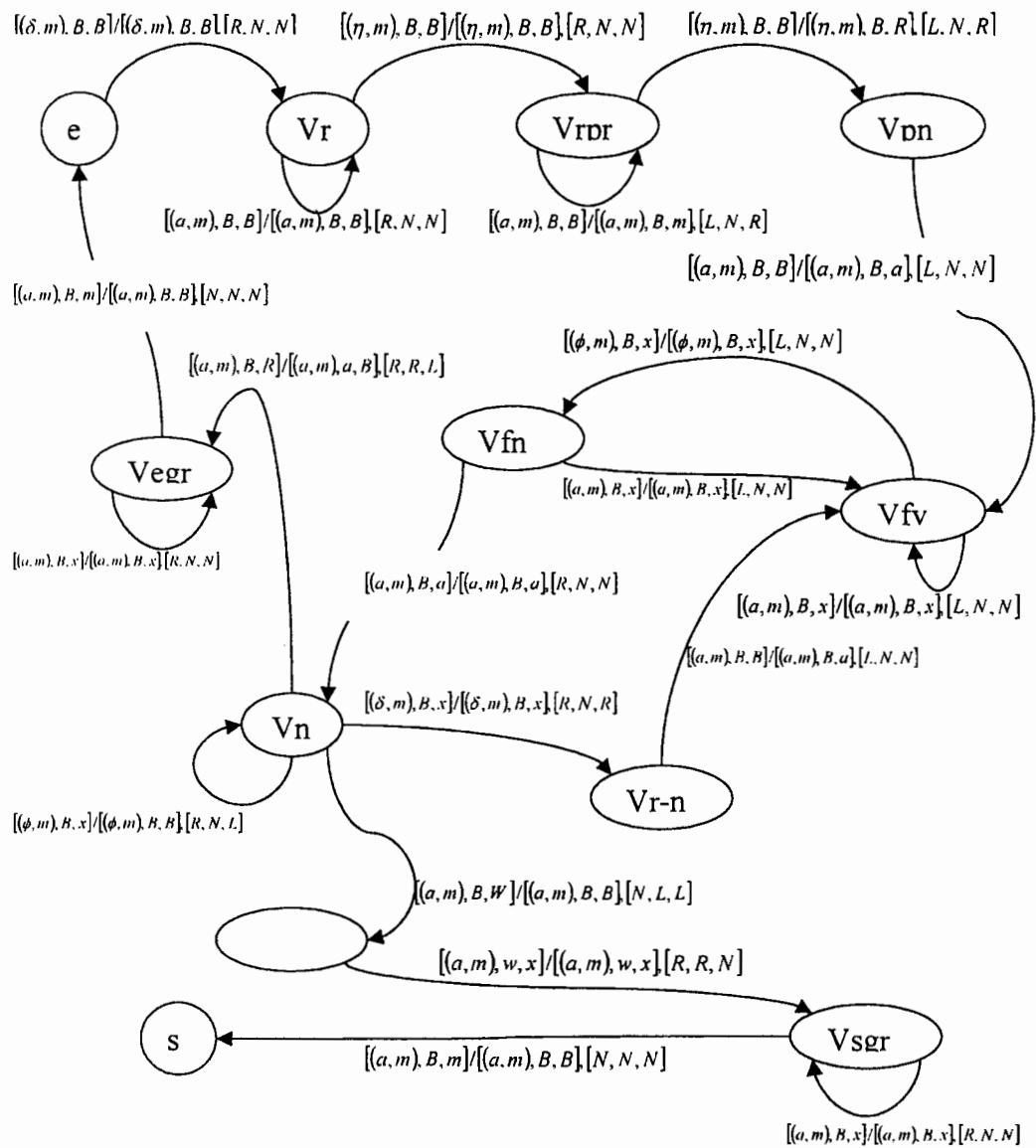


Figure 3.3 Accessing a variable for reading and updating

3.7. Statements:

If we check the source code of any language, it is a series of statements just controlled by some flow control statements. Statements are executed in 's' state. At the starting of program the TM is at 's' state. At the beginning, inside a block (whether a function call or loop or any other class of block it might be) the TM is at normal state (s state) executing statements. Statements are of the following categories:

3.8. Assignment statement.

(Variable := expression.): in 's' state if the TM sense a variable it assumes that it is an Assignment statement. So until it sense a ":", it waits. Then it goes to 'e' state and reads variable as discussed in the variable part. Initial moves up to 'e' state are shown in the figure 3.4.

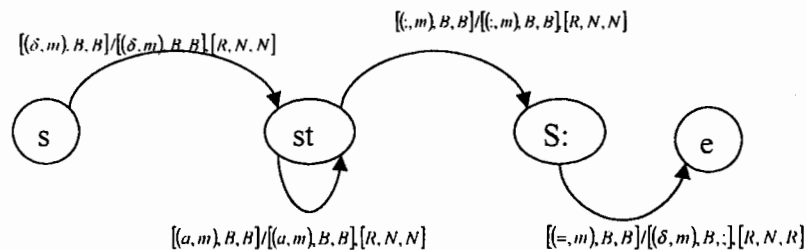


Figure 3.4 initial states for an assignment statement

Further moves are discussed at expression part. Other moves regarding assignment are discussed in evaluation part.

3.9. Send / Receive expression to another Turing Machine.

Send: To make the send operation easy, in this model the data stream to be sent is first gathered into the buffer (second tape) which is empty before and after this. The variable reading and sending to tape 2 is discussed in variable read part. Then it reads the send symbol and goes to send state and start sending until the buffer (second tape) is empty. After it is empty it goes to s state again. The moves are discussed below.

We have stated before the components those are exchanging data among themselves can be ports of a single CPU or different CPUs connected by communication ports. But all of

those should have an address.

We here also add that data sent and received to and from different I/O ports also done by send and receive function.

In 's' state if the TM reads a 'SEND' in it goes to 'e' state and reads variable and writes values to postfix tape as discussed in the variable read section. On reading 'SEND_S' in 'e' state it goes to 'Ξ' state.

'Ξ' state: in this state it reads the next tape value as the destination address (n) and goes to 'Ξn' state. In this state it keeps on sending and erasing from the second tape. When the second tape is empty, it finishes sending and return to 's' state. Diagram for the moves are shown in figure 3.5.

3.9.1. Send

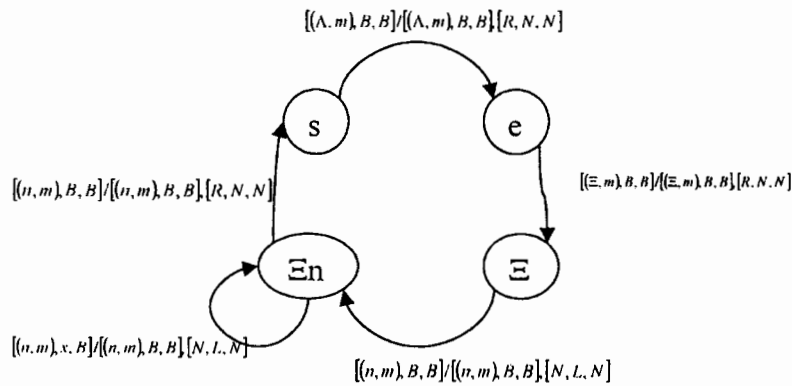


Figure3.5 moves for sending data to another machine

As in figure 3.6, on reading the 'RECV' keyword it goes to ' Θ ' state. In this state it assumes the next tape box value as the receiving port address (n). it reads the receiving port address and goes to ' Θn ' state. In this state it keeps on receiving values from the port and on receiving 'Blank symbol' it goes to 's' state. In 's' state it reads a variable call and fetch the variable and write the value there from postfix (second) tape. Moves of it is discussed below.

On reading a 'VCL' keyword in 's' state it goes to 'st' state.

In 'st' state it cross the variable and read ';'. On reading ';' it moves right and push the box number as return point and goes to 's;' state. In this state it pushes 'W' and goes to 's;-2' state moves after that are discussed in variable read part.

3.9.2. Receive

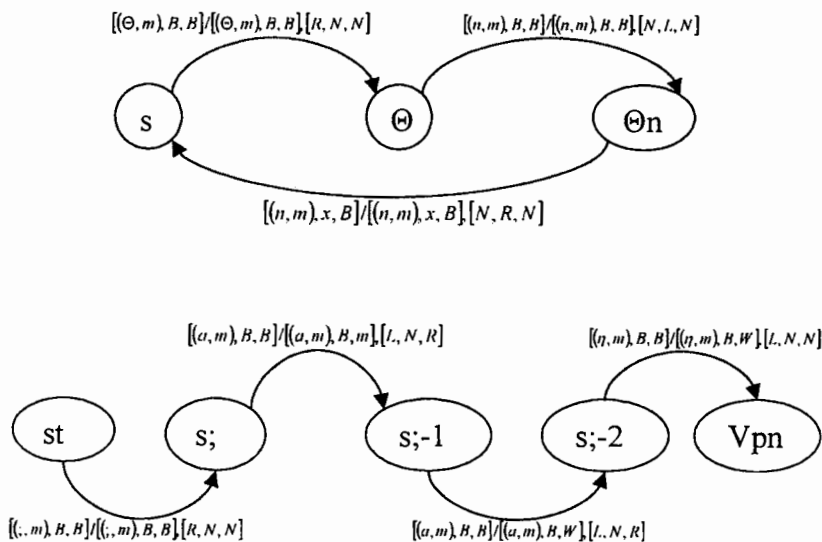


Figure 3.6 Moves for receiving from another machine

3.10. Read input / Write output:

These operations are same as send and receive which is discussed later. The difference is at the send / receive node number. It is the port address of the input and output port.

3.11. Expressions:

These are arithmetic operations with brackets and indices or condition checking. When an expression comes as a variable assignment statement a ';' is pushed (At the end it will be popped) then the expression is converted into a postfix form. Our expression will contain numeric values and operators terminated by ';'. The numeric values will be a value in a box of tape. The operators will be:

- (: open parenthesis
-): closing parenthesis
- *: multiplication
- /: division
- +: addition
- -: subtraction

These operators have four levels of precedence.

- Highest: (,)
- Middle: *, /
- Low: +, -
- Lowest: < > = !

We will add one more operator to help with the processing of the program. That is a semicolon, ';', which will indicate the end of an expression.

We start converting expression into postfix form at 'e' state.

To convert an expression from infix to postfix, we must first determine if the next (valid) item in the equation is a number or an operator.

⇒ If the item is a number:

If the item is a number, it is sent directly to the postfix equation. This is discussed in the variable part how to do it.

⇒ If the item is an operator:

If the item is an operator, we use the stack to do one of the following:

If the operator is: (, *, /, + or - . in 'e' state if the operator is 'o' go to 'eo' state.

In 'eo' state move the operators from tape 3 to tape 2 that is pop the stack until either the stack is empty, the operator '(' at the top of the stack or the top value on the stack has lower precedence than the current operator. Moving from third tape to second tape means each operator is popped from the stack, it is sent directly to the postfix equation. Then we go to 'eoo' state and push the current operator onto the stack and return to 'e' state.

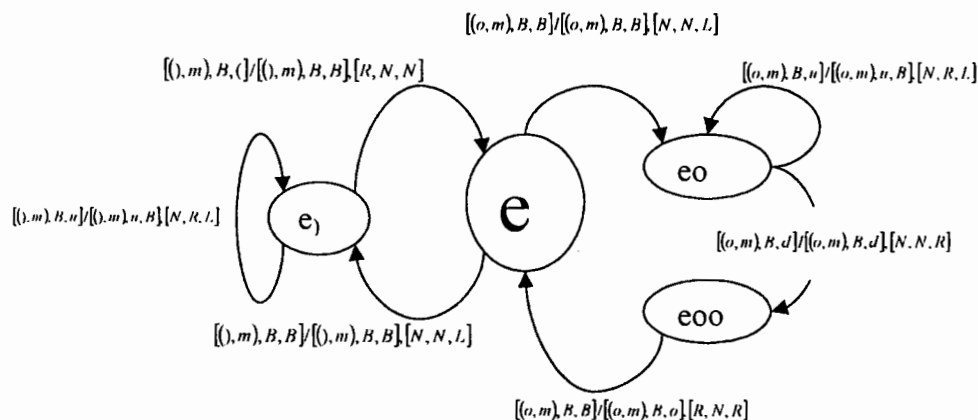


Figure 3.7 Handling bracket in converting to postfix equation

⇒ If the operator is:)

Pop the stack until pop an open parenthesis, (. The closing parenthesis is NOT pushed on the stack. All operators that are popped (except for the open parenthesis) are sent directly to the postfix equation. If there is not an open parenthesis on the stack (that. you empty the stack), the equation has an error of an unmatched closing parenthesis.

If the operator is ';' pop the stack until ';' at the top, sending all popped operators to the postfix equation other than ';'. Evaluate the expression and save the result into variable. Then pop the ';'. In case the expression contains variables the variables are retrieved and the value is sent to postfix equation.

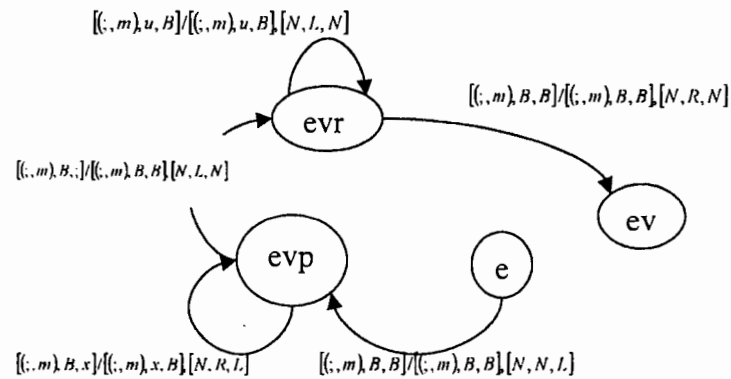


Figure 3.8 moves for sending remaining operators at stack to postfix equation and coming to starting of postfix equation.

3.12. Evaluating postfix equation:

If the machine reads a ';' at 'e' state it comes to know that the whole equation has been read and now it is time to make the stack empty to convert them to postfix form. So in 'e' state if it reads ';' it goes to 'evp' state. In this state it first pops all the remaining operators from the stack and send them to postfix form equation until it reads a ';'. On reading the ';' it assumes the bottom of the stack has been reached and erase the ';' and go to 'evr' state.

In 'evr' state the tape pointer of the second tape is at the end of the postfix equation. So it goes to the beginning of the equation and switch to 'ev' state to evaluate.

To evaluate a postfix expression, we will again need to use the stack. (Third Tape) However, the stack will contain numeric values instead of operators.

In 'ev' state when a numeric value is encounter, the value is pushed on the stack.

When an operator (other than ;) is encountered, two values are popped from the stack the operation is performed on these two values and the result is pushed onto the stack going through 'evo' and 'evoe' states and returning to 'ev' state.

When the no operand is encountered, there should only be one value on the stack above the ';'. This value is the result of the expression. This value is moved to postfix equation because it is to be written in the value part of the left variable of the equation. Also

moving the tape 1 head right one box it goes to 'ev1' state. It push the tape 1 box number as return point and go to 'ev2' state. Then it push 'W' indicating its willingness to write into the value part of variable. Then it goes to 'ev3' state.

In this state it keeps moving left until it reads a : indicating the L Value part has arrived. Now it goes to 'ev4' and 'vpn' states and reads the variable name.

The next moves from here is discussed in the variable part.

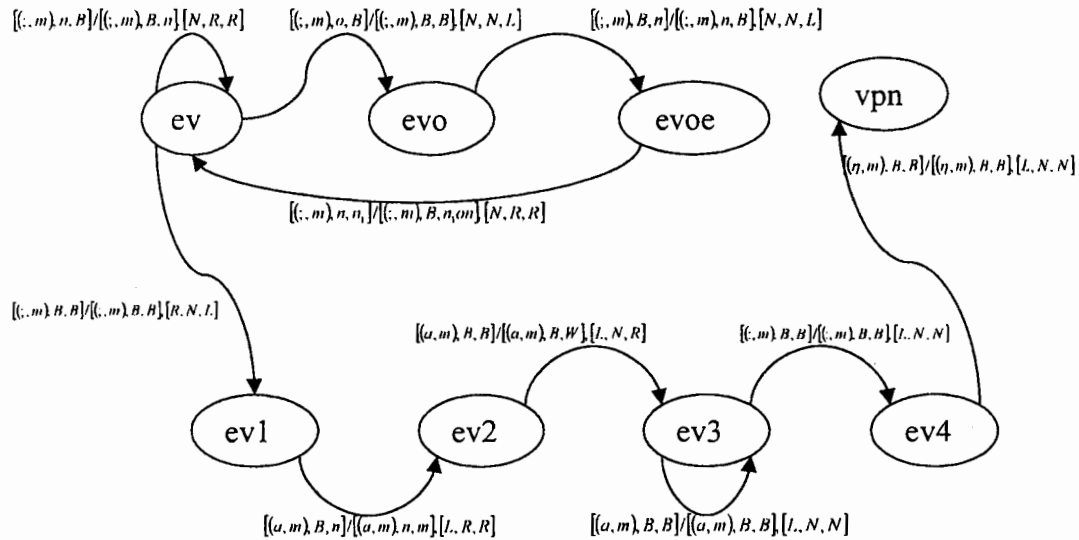


Figure 3.9 moves for evaluating postfix equation

3.13. Conditional blocks:

Conditional blocks are of two types:

Block like if else statements in C: when it reads an 'IF' it goes to expression state. On the right side of IF must be some expression to evaluate and it evaluates it and push either 'T' or 'F' in the third tape. Then it reads a 'THEN' then it evaluates and the boolean result from the third tape (stack) it also goes to the condition state. If the condition is false it goes to false state and do nothing until it goes to a normal state.('s') in false state it only moves right but when reads an 'IF' it just push an 'IF' and when it finds 'ENDIF' it just pops the 'IF' If it finds 'ENDIF' in false state and the stack-top is false then it pops the false and goes to normal state. If it reads 'ELSE' then it goes to else part so if before it was false (there was false at the top) then it goes to true state else it goes to false state.

The moves are as follows:

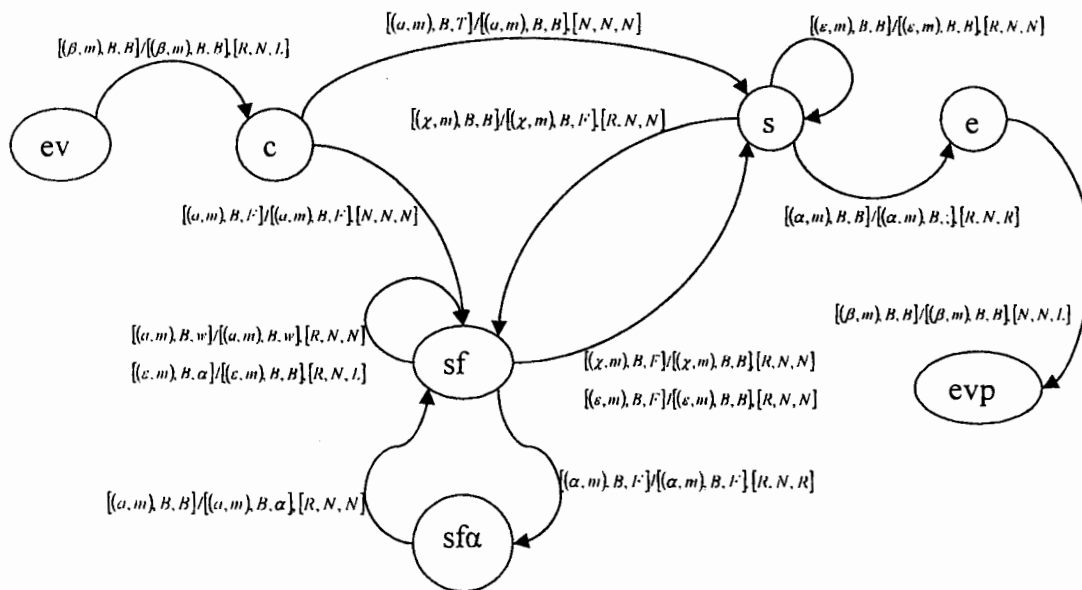


Figure 3.10 moves of conditional blocks.

Block like switch statements in C: For some simplicity reason of this design this part is not covered.

3.14. Loops:

When there is an instruction symbol for loop. It push the program tape number as return point. Then it checks the condition. Depending on the result of the check it goes to either true or false state and remains there till it gets end loop symbol. At end if it gets the end symbol while false state it pops the top (return point) else if it was in true state while reading the end loop symbol it goes back to the old point to check the condition again. The moves are as follows:

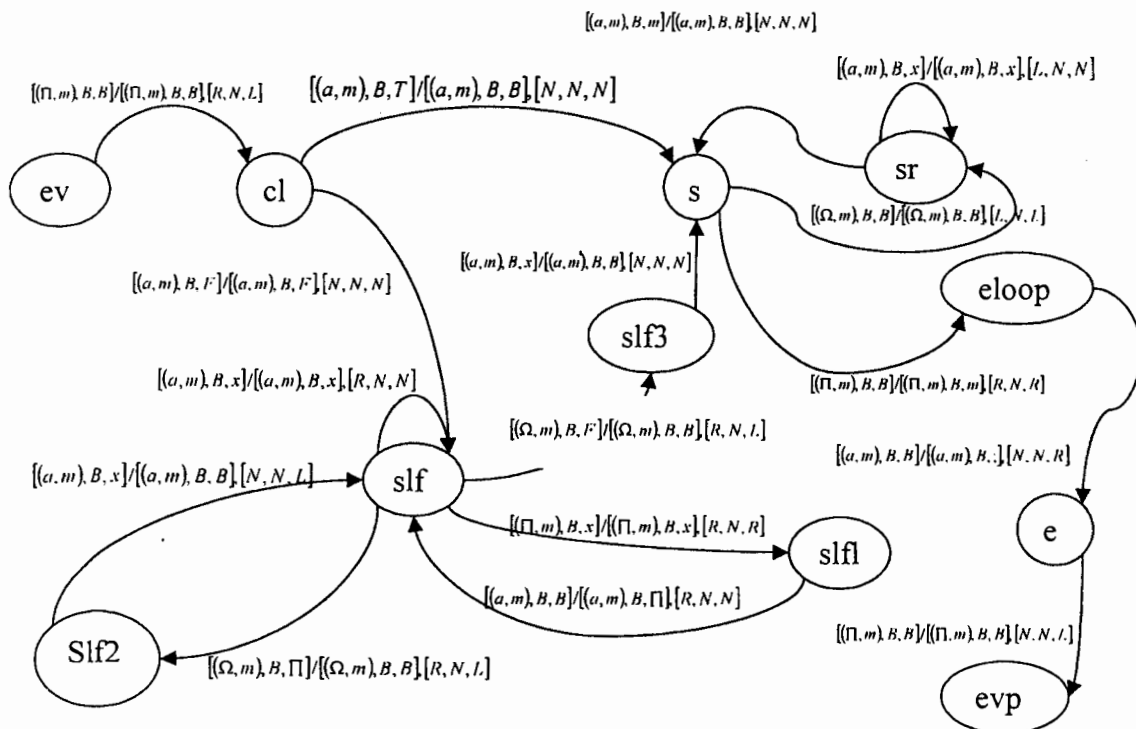


Figure 3.11 moves for loop

3.15. Function definition/calling:

Function is defined before the main program starts. Unlike the variable definition, there is no move for function definition area. It is covered at S state before reading 'hash' and going to s state. Function is defined as follows:

FUN <function-name> PARE <parameter list> BODY <Expressions> BODYEND

Where

- FUN, the left end symbol of definition,
- PARE symbol used as separator between function name and parameters,
- Parameters are defined as reference variable is stated in the variable definition part.
- BODY, the left point of definition before the body.
- Then comes the body of function as sequences of code.
- BODYEND the last point of body.

Each occupies one block of tape, as all of these are tokens.

Function is used as follows: (The moves are discussed below the text.)

Function Calling:

FUNC <function-name> (<parameter list>) <Expressions> ENDCALL

Where

- FUNC the left end symbol of definition,
- ENDCALL the left point of function call.

When TM read a Function call it goes to function call state and then it reach to the end of the call point ENDCALL, then push the current program point (block number of program tape), the box address as return point then it start moving left and push the parameters from the left side, at last push the Function name then it starts moving left going to state “finding function”. When it finds it, (Read a PARE followed by the matched name. When it read PARE it comes to know that there is a Function name at left, so it check the left box and if left box match the top-stack the definition is found.) it start moving right two boxes and then parameters are starting there. It first then push ‘hash’ then goes to the last part of definition and start coming back and pushing the whole function body till the starting of function. Then it goes to the end point of the tape, starts popping the whole function. When it pop ‘hash’ it assumes the next is parameters so it places parameters in appropriate places. Then it goes to the s state and start executing body. At the end (when it read BODYEND) it again goes to the return point from where it was called.

We have told in the variable part that global variables using inside a function body must

be defined before the function definition so there is no problem in global variable retrieval process.

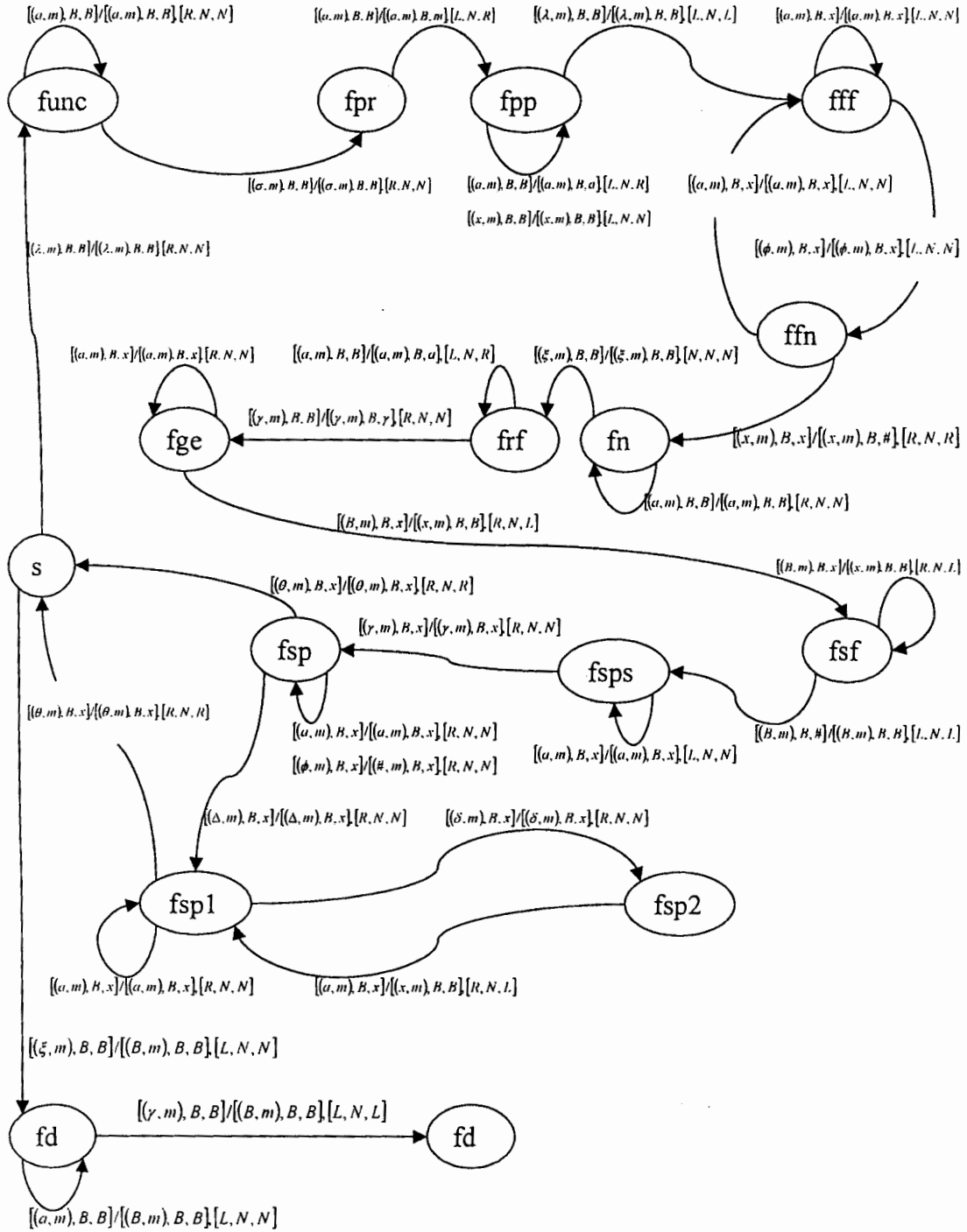


Figure3.12 moves for function handling

3.16. Interface

We have used MS Visual C++ 6 to implement our proposed Turing Machine. Its interface is as shown in the figure. Different parts of it are self descriptive.

After giving source code (input) of this language and pressing 'Run program' button, the Turing Machine starts moving its three heads according to following conditions:

The implemented software takes input (source code of the language of Turing Machine) in one text area.

- The source code is case sensitive.
- The tokens are written separated by single space for the machine to detect them.
- Delay between executions of moves can be set. By default it is 1 milliseconds
- It is also possible to run the machine in steps by checking the check box.
- The three tapes discussed in the research paper are also shown at the dialog Box.
- During execution time the status of the three tapes are displayed.
- Error handling features are partially implemented. That is, if an error occurs in any state (the configuration has no valid move) the Turing machine may not display error on that specific state but it will definitely show error at some later state.

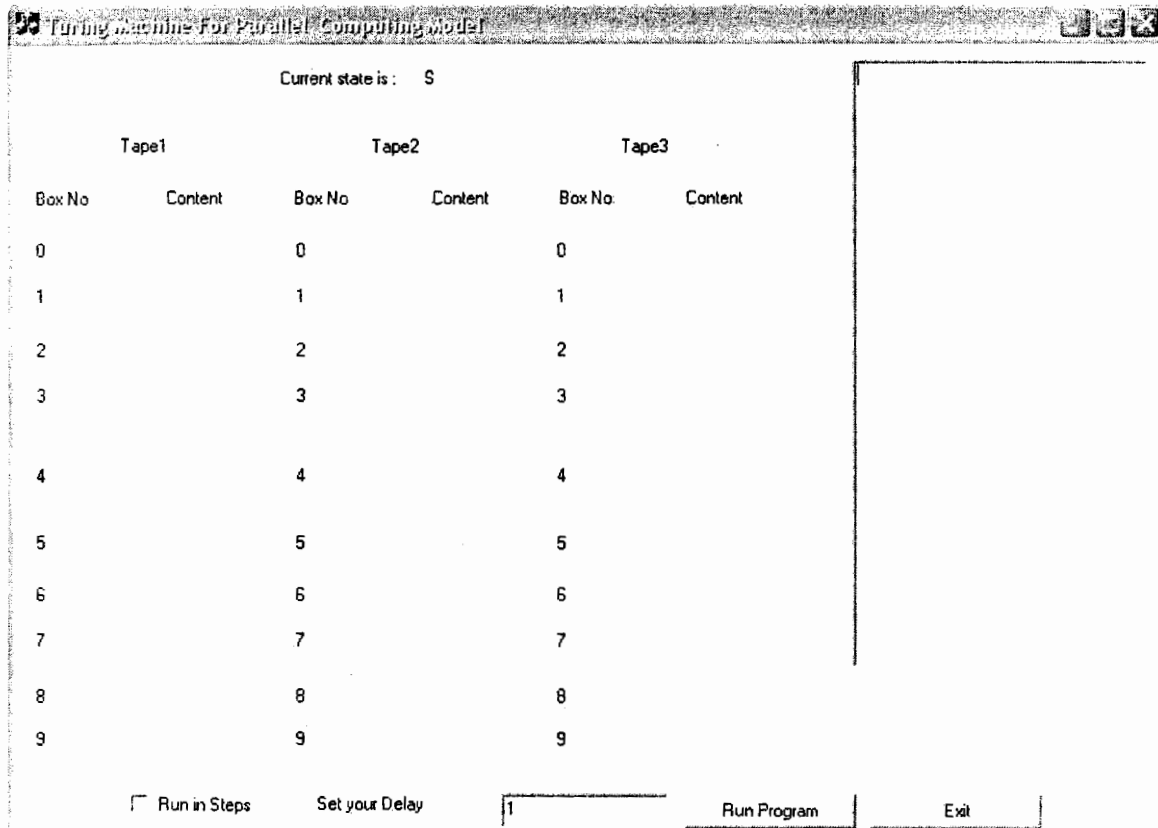


Figure 3.13 Interface of the Turing machine

3.17. Syntax of Language

Syntax to write program in this Turing Machine is discussed below.

- Main body:
 - Before the Main body we may write our comments followed by variable declarations and function declarations. It must be noted that:
 - All variables are global, so duplicate variable declaration will result in error.
 - Variables used in function must be declared before their use.
 - No constant value concept in the program. So all the constant values must be defined as variables.
 - Function must be defined before the program starts.
 - Start point of program is key word 'hash'. On reading a 'hash' in the tape one the program begins execution.
 - One may give source code as input after the first 'hash' only. There should be no 'hash' keyword in the main program till the end.
 - On reading two consecutive 'hash', the program stops. After that there should be no code but may be comment. So comments must be either before the first 'hash' or at the end after the two consecutive 'hash'.
 - We may summarize the whole program syntax as follows:
 <comments> <variable-definitions> <function-definitions> hash <main-body>
 hash hash <comments>
 - Variable Declarations: We have only one type of variable. To declare a variable we have to follow the following format with keywords.
 - 'VDL' the left end symbol of definition,
 - 'VDM' keyword is used as separator between value and name,
 - 'VDR' keyword is used as the left point
 - VDL <variable-name> VDM <initial-value> VDR
 - Variable using: We have to follow the following format to use the variable.
 - 'VCL' the left end symbol of variable call,
 - 'VCR' keyword is used as the left point
 - VCL <variable-name> VCR
- Variable Reference Declarations:
 - VDL <variable-name> VDM VCL <variable-name> VCR VDR
 - It is same as variable definition instead at value area the referring variable is called. It is specially designed to use as function parameters.
- Expression syntax:
 - Expression writing syntax is call the assignment variable and then write '=' and the write syntax as call the variables and put the operators between them.
 - Variable Name : = Variable Name operator Expression ;

- Send to other Turing machine or output device.
 - This Turing machine uses 'send' and 'receive' to send to and receive from other this type of Turing machine.
 - It is worth to be noted that for every Input and Output operations, it receives and sends. This model assumes the I/O ports as other Turing machine like itself..
 - We use IP address to send to other machine and different keywords for different I/O ports.
 - SEND Variable Name Variable Name...SEND_S <send-address>
- Receive from other Turing machine or input device.
 - RECV <receive-address> Variable Name Variable Name...
- If then else syntax is discussed below.
 - IF <condition-like-expression> THEN <Expressions> ELSE <Expressions> ENDIF
 - Here the condition is also like expression.
- Loop syntax is discussed below:
 - LOOP <Condition> LOOP <Expressions> ENDLOOP
 - Here the condition is also like expression.
- Function Definition syntax is discussed below:
 - FUN <function-name> PARE <parameter list> BODY <Expressions> BODYEND
- Function Calling:
 - FUNC <function-name> <parameter list> ENDCALL

Chapter 4

Results

4.Results

In this chapter we present the sample program written using language of our machine. At run time, our implemented software writes all the configurations of the Turing Machine moves in a file named “moves.txt” at the folder where the program resides. Detail of all the moves can be seen there. Summery of the moves are discussed below with their source codes.

4.1. Program to display:

This program defines a variable. It passes through variable definition states and enters sending state and display.

```
hash VDL number1 VDM 5 VDR SEND VCL number1 VCR SEND_S
DISPLAY hash hash
```

This program defines a variable. It passes through variable definition states and enters sending state and sends it to the node having IP address 192.168.2.30.

```
hash VDL number1 VDM 5 VDR SEND VCL number1 VCR SEND_S
192.168.2.30 hash hash
```

4.2. Program to assign:

This program defines two variables and then assigns ones value two another and displays the assigned value.

```
VDL number1 VDM 5 VDR VDL number2 VDM 3 VDR VDL result VDM 0
VDR hash VCL result VCR := VCL number1 VCR ; SEND VCL result VCR
SEND_S DISPLAY hash hash
```

4.3. Program to multiply two numbers:

This program defines three variables and then multiplies ones value with another and saves the result in the third and displays the result..

```
VDL number1 VDM 9 VDR VDL number2 VDM 5 VDR VDL result VDM 0
VDR hash VCL result VCR := VCL number2 VCR * VCL number1 VCR ;
```

```
SEND VCL result VCR SEND_S DISPLAY hash hash
```

4.4. Program to receive and Display:

This program defines a variable. It passes through variable definition states and enters receiving states. Then it receives a value from user and saves it in the variable and then enters sending state and displays the contents of the variable.

```
hash VDL number1 VDM 5 VDR RECV USER VCL number1 VCR ; SEND
VCL number1 VCR SEND_S DISPLAY hash hash
```

This program defines a variable. It passes through variable definition states and enters receiving states. Then it receives a value from node having IP address 192.168.2.30 and saves it in the variable and then enters sending state and displays the contents of the variable.

```
hash VDL number1 VDM 5 VDR RECV 127.0.0.1 VCL number1 VCR ; SEND
VCL number1 VCR SEND_S DISPLAY hash hash
```

4.5. Program to receive, manipulate and Display:

This program defines two variables and receives their values as described above and multiplies their values and display.

```
hash VDL num VDM 5 VDR VDL number VDM VCL num VCR VDR VDL
number1 VDM 5 VDR VDL number2 VDM 5 VDR RECV USER VCL number
VCR ; RECV USER VCL number1 VCR ; RECV USER VCL number2 VCR ;
VCL number VCR := VCL number2 VCR * VCL number1 VCR ; SEND VCL
number VCR VCL number1 VCR VCL number2 VCR SEND_S DISPLAY hash
hash
```

4.6. Program to receive, if else:

This program defines two variables and receives their values as described above and goes to if else state and check the greater value and display it.

```
hash VDL num VDM 5 VDR VDL number VDM VCL num VCR VDR VDL
number1 VDM 5 VDR VDL number2 VDM 5 VDR RECV USER VCL number
VCR ; RECV USER VCL number1 VCR ; RECV USER VCL number2 VCR ;
```

```
VCL number VCR := VCL number2 VCR * VCL number1 VCR ; IF VCL
number VCR < VCL number1 VCR THEN SEND VCL number VCR SEND_S
DISPLAY ELSE SEND VCL number1 VCR SEND_S DISPLAY ENDIF hash
hash
```

4.7. Program to receive, nested if else (Find Smallest):

This program also act as the previous program but it performs nested if else in the if--then else moves.

```
hash VDL num VDM 5 VDR VDL number VDM VCL num VCR VDR VDL
number1 VDM 5 VDR VDL number2 VDM 5 VDR RECV USER VCL number
VCR ; RECV USER VCL number1 VCR ; RECV USER VCL number2 VCR ;
VCL number VCR := VCL number2 VCR * VCL number1 VCR ; IF VCL
number VCR < VCL number1 VCR THEN IF VCL number2 VCR < VCL
number1 VCR THEN SEND VCL number1 VCR SEND_S DISPLAY ELSE
SEND VCL number2 VCR SEND_S DISPLAY ENDIF ELSE IF VCL number2
VCR < VCL number VCR THEN SEND VCL number VCR SEND_S DISPLAY
ELSE SEND VCL number2 VCR SEND_S DISPLAY ENDIF ENDIF hash hash
```

4.8. Program to receive, Loop (Calculate Factorial):

This program defines a variable and receives its values as described above and goes to loop state and find the factorial of the number and display it.

```
hash VDL num VDM 1 VDR VDL number VDM VCL num VCR VDR VDL
number1 VDM 1 VDR VDL number2 VDM 1 VDR RECV USER VCL number2
VCR ; LOOP VCL number VCR < VCL number2 VCR LOOP VCL number1
VCR := VCL number2 VCR * VCL number1 VCR ; VCL number2 VCR :=
VCL number2 VCR - VCL number VCR ; ENDLOOP SEND VCL number1
VCR SEND_S DISPLAY hash hash
```

4.9. Nested Loop

These are examples of nested loop programs.

```
VDL num1 VDM 3 VDR VDL num2 VDM 2 VDR hash VDL num VDM 1
```

```
VDR VDL number VDM VCL num VCR VDR VDL number1 VDM 1 VDR
VDL number2 VDM 1 VDR RECV USER VCL number2 VCR ; LOOP VCL
num2 VCR < VCL num1 VCR LOOP LOOP VCL number VCR < VCL
number2 VCR LOOP VCL number1 VCR := VCL number2 VCR * VCL
number1 VCR ; VCL number2 VCR := VCL number2 VCR - VCL number
VCR ; ENDLOOP VCL num1 VCR := VCL num1 VCR - VCL num2 VCR ;
ENDLOOP SEND VCL number1 VCR SEND_S DISPLAY hash hash
```

```
hash VDL num VDM 1 VDR VDL number VDM VCL num VCR VDR VDL
number1 VDM 3 VDR VDL number2 VDM 4 VDR LOOP VCL number VCR <
VCL number1 VCR LOOP LOOP VCL number VCR < VCL number2 VCR
LOOP SEND VCL number2 VCR SEND_S DISPLAY VCL number2 VCR :=
VCL number2 VCR - VCL number VCR ; ENDLOOP SEND VCL number1
VCR SEND_S DISPLAY VCL number2 VCR := VCL number2 VCR + VCL
number1 VCR ; VCL number1 VCR := VCL number1 VCR - VCL number
VCR ; ENDLOOP hash hash
```

```
hash VDL num VDM 1 VDR VDL number VDM VCL num VCR VDR VDL
number1 VDM 3 VDR VDL number2 VDM 4 VDR LOOP VCL number VCR <
VCL number1 VCR LOOP LOOP VCL number VCR < VCL number2 VCR
LOOP SEND VCL number2 VCR SEND_S DISPLAY VCL number2 VCR :=
VCL number2 VCR - VCL number VCR ; ENDLOOP SEND VCL number1
VCR SEND_S DISPLAY VCL number1 VCR := VCL number1 VCR - VCL
number VCR ; ENDLOOP hash hash
```

4.10. Program to sequence:

The following programs solves equations having more than two variables.

```
VDL number VDM 5 VDR VDL number1 VDM 7 VDR VDL number2 VDM 3
VDR VDL number3 VDM 8 VDR VDL number4 VDM 12 VDR VDL result
VDM VCL number VCR VDR VDL number5 VDM 3 VDR VDL number6
VDM 3 VDR hash VCL number VCR := VCL number1 VCR - VCL number2
VCR * VCL number3 VCR + VCL number4 VCR / VCL number5 VCR ; SEND
VCL result VCR SEND_S DISPLAY hash hash
```



```

VDL number1 VDM 9 VDR VDL number2 VDM 5 VDR VDL number3 VDM 6
VDR VDL result VDM 0 VDR hash VCL result VCR := VCL number1 VCR +
VCL number2 VCR * VCL number3 VCR ; SEND VCL result VCR SEND_S
DISPLAY hash hash

```

```

VDL number1 VDM 9 VDR VDL number2 VDM 5 VDR VDL number3 VDM 6
VDR VDL result VDM 0 VDR hash VCL result VCR := VCL number2 VCR *
VCL number3 VCR * VCL number1 VCR ; SEND VCL result VCR SEND_S
DISPLAY hash hash

```

4.11. Program to Function receive, Loop:

The following programs defines functions and at run time calls the functions.

```

FUN dis PARE BODY VDL number1 VDM 5 VDR SEND VCL number1 VCR
SEND_S DISPLAY SEND VCL number1 VCR SEND_S DISPLAY END hash
FUNC dis ENDCALL hash hash

```

```

VDL num VDM 5 VDR FUN dis PARE VDL number VDM VCL num VCR
VDR BODY VDL number1 VDM 5 VDR SEND VCL number1 VCR SEND_S
DISPLAY SEND VCL number VCR SEND_S DISPLAY END hash VDL a
VDM 9 VDR FUNC dis VCL a VCR ENDCALL hash hash

```

```

VDL num VDM 1 VDR FUN fact PARE VDL number VDM VCL num VCR
VDR BODY VDL number1 VDM 1 VDR VDL number2 VDM 1 VDR RECV
USER VCL number2 VCR ; LOOP VCL number VCR < VCL number2 VCR
LOOP VCL number1 VCR := VCL number2 VCR * VCL number1 VCR ; VCL
number2 VCR := VCL number2 VCR - VCL number VCR ; ENDLOOP SEND
VCL number1 VCR SEND_S DISPLAY END hash VDL a VDM 1 VDR FUNC
fact VCL a VCR ENDCALL hash hash

```

Chapter 5

Conclusion and Future Work

5. Conclusion and future work

5.1. Conclusion

In this project a computing model having a new Idea has been created and implemented. Our new computing model is entirely based on Turing machine theory. It has also the most common features of higher level language. This also simulates communications among different computing models.

Moreover this is mathematical model that can be implemented on various architecture. Even different sub Turing machines of our big Turing machine can be implemented on different architecture.

Two things have been achieved in this project:

- “Equivalence of computer and Turing machine proved.”
- A new way to implement distributed computing found.

5.2. Future Work

As future work, we may also construct Turing machines accepting languages of more complex programming paradigms. Some of them are stated below.

- Object-oriented programming
- Event-driven programming
- Flow-driven programming,
- Message passing programming,
- Class-based programming,
- Prototype-based programming (within the context of Object-oriented programming)
- Logic programming
- Constraint programming,
- Component-oriented programming (as in OLE)
- Aspect-oriented programming (as in AspectJ)
- Rule-based programming (as in Mathematica)
- Table-Oriented Programming (as in Microsoft FoxPro)
- Pipeline Programming (as in the UNIX command line)
- Post-object programming
- Subject-oriented programming
- Reflective programming

- Dataflow programming (as in Spreadsheets)
- Policy-based programming
- Annotative programming

We may also construct Turing Machine that have easier syntax, less complexity in running time.

REFERANCE

REFERENCE

1. Turing, A.M., "On Computable Numbers with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, 42 (1936), pp. 230–265.
2. UTM - Computability, Algorithms, and Complexity, Course 240 page – 57
http://www.doc.ic.ac.uk/~imh/teaching/Turing_machines/240.pdf
3. <http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Church.html>
4. Programming Languages, mathematical theory of computation, artificial intelligence, Stanford University. www-formal.stanford.edu/jmcl/
5. The Universal TuringTheorem 69 -
- http://www.mcs.vuw.ac.nz/courses/MATH435/2005T2/435_notes_all.pdf
6. Georg Cantor: His Mathematics and Philosophy of the Infinite by Joseph Warren Dauben
7. <http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Hilbert.html>
8. <http://www.mathematik.uni-bielefeld.de/~kersten/hilbert/problems.html>
9. Hilbert, D., "Probleme der Grundlegung der Mathematik," *Mathematische Annalen* 102 (1930) 1--9.
10. Gödel, K., "Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme," *Monatshefte für Mathematik und Physik* 38 (1931) 173--198. (English translation in *The Undecidable*, M. Davis, editor, Raven Press, New York, 1965.)
11. Grattan-Guinness, I., *The Search for Mathematical Roots, 1870--1940: Logics, Set Theories and The Foundations of Mathematics from Cantor through Russell to Gödel*, Princeton University Press, Princeton, 2000.
12. Hilbert, D., Ackermann, W., *Principles of Mathematical Logic*, Hammond, L., Leckie, G., Steinhardt, F., translators, Chelsea Publishing Co., New York, 1950. pp. 112--113.
13. Church, A., *Introduction to Mathematical Logic*, Princeton University Press, Princeton, New Jersey, 1996. p 99
14. Church, A., "An Unsolvable Problem of Elementary Number Theory," *American Journal of Math.* 58 (1936) 345--363.
15. Simulating Computer by Turing Machine (Hopcroft, J.E., R.Motwani and J.D. Ullman, 2000, Introduction to Automata Theory Language and computation, 2/E, Addison-Wesley CO, USA)
16. Guetta, D., 2003, "Turing Machine Development Environment", 19 Church Mount, London, N2 0RW, England (Project files can be found at <http://www.guetta.com/visualturing>)

Research paper

Turing Model for Distributed Computing

Saeed ur Rahman Khan and Malik Sikander Hayat Khiyal
Department of Computer Science, International Islamic University, Islamabad, Pakistan

Abstract: Turing machine is model for general-purpose computing device. There are many implementations of Turing Machine using physical computer as well as many implementations of physical computing device using Turing Machine. In this study, we elaborate the simulation of computing model, using a high-level language as its native code, on Turing machine. We also make it possible to communicate among many of such simulations. Finally we present a deterministic Turing model that actually simulates distributed processing.

Key words: Turing machine, physical machine, simulations

INTRODUCTION

Turing Machine is a mathematical model of computing device. Turing Machine^[1], as defined by Alan Turing in his historical study, is a 7 tuple as following:

$$TM = \{Q, \Sigma, \Gamma, \delta, S, B, f\}$$

Where, the symbols are sequentially the set of states, the input symbols, the tape symbols, the set of transition functions, the starting state, the blank symbol and finally the set of final states.

There are many simulations of computer using Turing Machine and simulation of Turing Machine using computer to prove them equivalent^[2-4]. Here, we elaborate them. This study describes a computing model having many addressable identities as basic unit, Turing Machine. Then the complete Turing Machine makes it possible to make communications among these models.

Thus in future, we shall use only TM as the formal representation of what can be computed by any kind of computing device^[4].

Our proposed model is a compound Turing Machine of many simple Turing Machines. We now describe systematically the characteristics of our model.

In modern computing model, different parts of a computing model is again a computing model themselves having their own addressable identities.

There are many kinds of physical computing devices, such as mini, micro, mainframe, super scalar etc. These may be general purpose and special purpose. Computing models may be physical and virtual (as software on other physical model). But the virtual models can also be described as a big algorithm, collection/sequence of small physically implemented algorithms.

In spite of these facts, there is no fundamental difference between data processing, storage and communication between any kinds of computing model mentioned above^[5].

In a fully distributed environment, there can be global addressing systems to address a small component of a native node participating in the whole system^[6]. There may be many mechanism adopted for this purpose.

This study describes using Turing Machine:
A computer with

- Infinity long sequence of words each with an address
- Infinity long address space.
- Program of computer is stored in some word of memory.

Each word or box of tape represents a simple token of Higher-level language (Unlike the models described in existing books that refers instruction of a memory as in the machine or assembly language.)

MODEL IN BRIEF

$$TM = \{Q^n, \Sigma, \Gamma, \delta, S, B, f\}$$

States: Q^n . The states are compound states of the states of its sub Turing Machines. So we may say states are of the following types: $q_1 q_2 q_3 \dots q_n$ where, $q \in Q$.

Σ is set of input alphabets.

Γ is set of tape alphabets.

δ is set of moves. The moves are also compound moves of the following formats:

$$\begin{aligned} &\delta(p_1, [(a_1, m_1), w_1, x_1])(q_1, [(b_1, m_1), y_1, z_1], [D, D, D])_1 \\ &\delta(p_2, [(a_2, m_2), w_2, x_2])(q_2, [(b_2, m_2), y_2, z_2], [D, D, D])_2 \\ &\delta(p_3, [(a_3, m_3), w_3, x_3])(q_3, [(b_3, m_3), y_3, z_3], [D, D, D])_3 \\ &\vdots \\ &\delta(p_n, [(a_n, m_n), w_n, x_n])(q_n, [(b_n, m_n), y_n, z_n], [D, D, D])_n \end{aligned}$$

Where, $a, b, m, w, x, y, z \in \Gamma$, $q_1, q_2 \in Q, D \in \{L, R, N\}$

S is the starting state for all sub Turing Machines of the compound Turing Machine.

B is the blank symbol.

f is the final accepting state. But when all the sub Turing Machines goes to accepting state then the final Turing Machine is assumed to be at the final state. So the final state of the whole Turing Machine is $f_1 f_2 f_3 \dots K_n$

FEATURES

Our aim is to make the Turing machine that having the following features:

- The TM is designed in so that its functionality will be nearest possible of real computer while remaining within the limitation of TM, the input code of TM (which is also the executable code) will be like that of Higher level Language code. Not all the functionality of Higher Level Language are discussed, but some of them to limit the complexity.
- Each token in the Higher Level Language will occupy first tuple of a block in the program tape, which the head will read at once along with the block number (token number).
- The TM is to simulate the computing functions like those are in a distributed environment.
- In literatures, a non-deterministic Turing Machine is known to be a parallel Turing machine. But in some other places it is shown that it is not actually simulation of a parallel processing with which we agree. We disagree with the statement that non deterministic Turing Machine to be a simulator of distributed computing environment, because of the following reasons

In non-deterministic environment where there are several moves for a given configuration, the Turing Machine starts moving in all paths and thus creates branches only at that particular configuration.

The number of branches only depends on the number of moves available from that given configuration, otherwise not. If there come a multiple path configuration at any point, it will

again branch in the number of moves those are available from that configuration.

In actual distributed environment, the distributed processors (nodes) are not created by instructions. (In thread creation, there is actually no physical parallel processing occurs. Rather the created threads share the time of same processing node. Even if they use different node the different nodes are previously created by some other mechanism.) In distributed environment, the nodes are mostly previously statically created. If not then processing nodes are created on availability basis but not because of any instruction.

We produce a deterministic Turing Machine having all the configurations with one move besides acting as distributed model.

- Our Turing Machine has each component as an independent computing model and those perform their tasks independently and while needed communicate with the other model. The main reason to unite the individual components and make the compound Turing Machine is, the components of the compound Turing Machine can communicate among themselves only not out of those. This will simulate the physical interconnected nodes those can communicate and make a distributed system. The components those are exchanging data among themselves can be ports of a single CPU or different CPUs connected by communication ports. But all of those should have an address.
- We have represented the reserve word tokens in this model with the help of Greek alphabets. But after implementation any other suitable token can be used in these places.

FUNCTIONALITIES

Computing machine has the following functions:

Variable definition/storage retrieval: To reduce the complexity, there are only two types of variable declarations here.

- Variables.
- Reference to other variable or reference.

Assignment operations: Basic arithmetic operations add/subtract/multiply divide: There is no explicit sub TM designed here for these.

Arithmetic and logical operations with brackets and indices: This is done by converting equations to postfix form.

Conditional blocks: We have only used 'if then else' type of conditional blocks here.

Loops: In this model loop is started by checking an initial value of a variable and termination is also done by checking that value. Inside the loop body, the variable is updated.

Function calling

- Functions in this model are only 'void' functions that mean those do not return value.
- Return value is only got from function, by parameters sent by reference.
- Function parameters only contain references to other variables with which functions send and receive values.

DESCRIPTION OF OUR TM

Our Turing Machine consists of components as stated before. This portion describes one component in detail. There are three tapes in this TM.

Program tape: This tape contains variable declarations, program and input. When the input instruction is called, the input value is stored in places reserved for input in this tape. This tape has two tracks. One for program and variable and another read only track contains sequence number of boxes of the former track. Point to be noted that the tape has one head with which it read both the track and write only on the first track second one is read only. In another way, we may also describe this tape as only one track having the boxes as tuple, which the TM head read. It writes only on the first part of the tuple, second part of the tuple contains the box number and this part is read only.

Postfix tape/Buffer tape: The arithmetic expressions are converted into postfix form before execution. The postfix equation is stored in this tape. At the beginning, this tape is empty. During the functionality, this tape can also be used for other purpose as well such as buffer while sending a buffer over a network. After finishing work, it is empty.

Stack tape: This tape acts as stack for this TM. At the beginning, this tape is empty. The functionality is same like the other tape but it is called stack tape as it is used as stack. Its way of functionality varies in the following way.

- When a value is pushed the tape head moves right and write the value or sometimes it writes and then moves when the current block is already empty.

- When a value is popped, the head erases the current value and moves left or it moves left and then erases the value if the current box is empty.
- At any time to read the top value just read the current box under the head or sometimes when the current box is empty it moves left and read the value.
- Even then, this is a tape so it can be used for other purpose as tape. Sometimes the push pop is merged with other moves.

As default the second and third tape head is kept over a blank box beside a filled box. When there is a value on either of these tapes that to be read constantly then that tapes head is kept on that box containing value.

- Moves: Moves of a single component are of the following form:

$$\delta(q_1, [(a,m),w,x])(q_2, [(b,m),y,z], [D,D,D])$$

where: $a,b,m,w,x,y,z \in \Gamma, q_1, q_2 \in Q, D \in \{L,R,N\}$

- Tape symbols

There are tape symbols for each instruction and each reserve words and each value with which the program is written on the tape. No defined name such as variable name or function name can be repeated. So at same time a name cannot be used as variable name and function. Let alone overriding/overloading etc. Tape symbols can be categorized in the following categories:

- Value
- Reserve words / key words
- Instruction symbols / state symbols: Note that there is a tape symbol for some of the states if not for all.
- Brackets of different types e.g. {, }, (,).
- Operators: + - * / = > < ! :=

WORKING OF TM AND MOVES

From the left to right the global variables and the functions are written on the program tape, followed by # as the marker for starting point of the main program. When the main program is finished there is a '###' as end marker of main program. After that there is no more program written at the starting of computation. However, at run time when a function is called, the called function's code is copied after the '##'. The next function call will copy the called function after that. When a function returns (finish) it is erased from the place where it was copied. at same time

The machine starts reading the symbols at the program track of program tape from left to right and fetches '#'. When it gets it go to s state and after that goes to their respective states and executes the instructions.

$$\delta(S, [(a, m), B, B]) (S, [(a, m), B, B], [R, N, N])$$

$$\delta(S, [(# , m), B, B]) (s, [(# , m), B, B], [R, N, N])$$

Where, $s \in Q$, $a \in \Gamma - \#$, B= Blank symbol, S=Starting state.

Moves at the end point.

$$\delta(s, [(# , m), B, B]) (s_e, [(# , m), B, B], [R, N, N])$$

$$\delta(s_e, [(# , m), B, B]) (f, [(# , m), B, B], [N, N, N])$$

Variable definition/storage retrieval

Variable is defined as follows:

$$\Delta V_n \Phi V_v \psi$$

Where:

- Δ the left end symbol of definition,
- V_n is a variable name,
- Φ symbol used as separator between value and name,
- V_v is a variable value,
- ψ the left point.

Each occupies one block of tape, as all of these are tokens. Whenever the tape head sense a Δ , it assumes that there is a variable definition at right and same for the other variable symbols for their respective purpose. It goes to Variable definition state and remains in that state until definition finishes.

$$\delta(s, [(\Delta, m), B, B]) (v_e, [(\Delta, m), B, B], [R, N, N])$$

$$\delta(v_e, [(a, m), B, B]) (v_d, [(a, m), B, B], [R, N, N])$$

$$\delta(v_d, [(\psi, m), B, B]) (s, [(\psi, m), B, B], [R, N, N])$$

where $a \in \Gamma - \psi - \Delta$, $v_e, v_d \in Q$.

If any defined variable is used as global variable inside some function it must be defined before the function first call due to some restriction in the variable retrieving process.

Variable definition at function parameter or variable as reference to another variable:

- $\Delta V_n \Phi \delta V_m \eta \psi$

Variable is read as follows: $\delta V_n \eta$

Where:

- δ the left end symbol of definition,
- V_n is a variable name,
- η the left point.

As the rule of the all programming languages variable must be defined before use. So from a call point the definition must be at the right side. So when TM read a variable, it first

- Push the return point. (The current box number)
- Push W or R for whether to read or write.
- Push the variable name.

When it starts reading variable it goes to the last point of variable first then it push the next box address (as return address) first then push whether to read or write, then push the variable name then it starts moving left going to 'finding variable state'. When it finds it, (Read a Φ followed by the matched name. Point to be noted that when it read Φ it know that there is a variable name at left, so it check the left box and if left box match the top-stack the definition of variable is found.) then it first delete the pushed name starts moving right two boxes and if the value is other then δ , it is the value. Otherwise it is the reference to another variable so it pushes the new name (the variable name at the reference) and again searches in the previous way. When it finds it, it pops the variable name from the third tape. Then it moves the 3^d tape head left. There it checks whether it was to be read or written.

- If it was to be read, it writes the value to the postfix equation.
- The expressions are solved by converting them to postfix form so the value should be written to postfix equation.
- Otherwise if it was to be written then it moves the value from postfix equation to the value part of the definition and erase from the postfix equation.
- After that it pop the stack so the next top is the memory location from where the variable was referenced. The state is also changed to state to go the place from where the variable was called.

Variable is read from state e. (Moves)

$$\delta(e, [(\delta, m), B, B]) (v_e, [(\delta, m), B, B], [R, N, N])$$

$$\delta(v_e, [(a, m), B, B]) (v_r, [(a, m), B, B], [R, N, N])$$

$$\delta(v_r, [(\eta, m), B, B]) (v_{pr}, [(\eta, m), B, B], [R, N, N])$$

$a \in \Gamma - \eta, e, v_r, v_{pr} \in Q$

$\delta(v_{pr}, [(a,m), B, B])(v_{pr}, [(a,m) B, m], [L, N, R])$
 $\delta(v_{pr}, [(n,m), B, B])(v_{pr}, [(n,m) B, R], [L, N, R])$
 $\delta(v_{pr}, [(a,m), B, B])(v_{pr}, [(a,m), B, a], [L, N, N])$
 $\delta(v_{pr}, [(a,m), B, x])(v_{pr}, [(a,m), B, x], [L, N, N])$
 $a \neq x, a \in \Gamma - \phi$
 $\delta(v_{pr}, [(a,m), B, x])(v_{pr}, [(a,m) B, x], [L, N, N])$
 $\delta(v_{pr}, [(a,m), B, x])(v_{pr}, [(a,m), B, x], [L, N, N])$
 $\delta(v_{pr}, [(a,m), B, a])(v_{pr}, [(a,m) B, a], [R, N, N])$
 $\delta(v_{pr}, [(a,m), B, x])(v_{pr}, [(a,m) B, B], [R, N, L])$
 $\delta(v_{pr}, [(a,m), B, x])(v_{pr}, [(a,m) B, B], [R, N, R])$
 $\delta(v_{pr}, [(a,m), B, B])(v_{pr}, [(a,m) B, a], [L, N, N])$
 $\delta(v_{pr}, [(a,m), B, R])(v_{pr}, [(a,m) a, B], [R, R, L])$
 $\delta(v_{pr}, [(a,m), B, W])(v_{pr}, [(a,m) B, B], [N, L, L])$
 $\delta(v_{pr}, [(a,m), w, x])(v_{pr}, [(a,m) w, x], [R, R, N])$
 $\delta(v_{pr}, [(a,m), B, x])(v_{pr}, [(a,m) B, x], [R, R, N])$
 $\delta(v_{pr}, [(a,m) B, m])(e, [(a,m), B, B], [N, N, N])$
 $\delta(v_{pr}, [(a,m), B, x])(v_{pr}, [(a,m), B, x], [R, N, N])$
 $\delta(v_{pr}, [(a,m) B, m])(s, [(a,m) B, B], [N, N, N])$

Variable updating: In any statement the Lvalue is calculated and updated. In this case no variable is updated at the right hand side of the assignment operator. So when we finish calculating the right hand side of the assignment operator, we fetch the variable by the previous way and assign the value there at the value part. This will be discussed in the statement part. How to write variable is already discussed in the previous part.

Variable length in each variable will occupy one block. In this TM, address space, variable value length, function names and all other reserve word, keyword length all is assumed to occupy one block as those are tokens of the native code of Turing Machine.

Statements: If we check the source code of any language, it is a series of statements just controlled by some flow control statements. Statements are executed in normal state. At the starting of program the TM is at normal state. At the beginning, inside a block (whether a function call or loop or any other class of block it might be) the TM is at normal state (s state) executing statements. Statements are of the following categories:

Assignment statement: (Variable := expression): in s state if the TM sense a variable it assumes that it is an Assignment statement. So until it sense a ":", it waits.

Then it goes to e state and moves after that are discussed in the variable part.

$\delta(s, [(\delta, m), B, B])(s, [(\delta, m), B, B], [R, N, N])$
 $\delta(s, [(a,m), B, B])(s, [(a,m), B, B], [R, N, N])$
 $a \in \Gamma - \{:, =, \delta\}$

Move to just read a variable and write to postfix tape/ buffer

$\delta(s, [(:, m), B, B])(s, [(:, m), B, B], [R, N, N])$
 $\delta(s, [(\delta, m), B, B])(e, [(\delta, m), B, :], [R, N, R])$

- Further moves are discussed at expression part.
- Other moves regarding assignment are discussed in evaluation part.

Read variable from user./Write expression to output: These operations are same as send and receive which is discussed later. The difference is at the send/receive node number. It is the port address of the input and output port.

Send/Receive expression to another Turing Machine: Send: To make the send operation easy, in this model the data stream to be sent is first gathered into the buffer (second tape) which is empty before and after this. The variable reading and sending to tape 2 is discussed in variable read part. Then it reads the send symbol and goes to send state and start sending until the buffer (second tape) is empty. After it is empty it goes to s state again. The moves are discussed below.

Send

$\delta(s, [(\Lambda, m), B, B])(e, [(\Lambda, m) B, R], [R, N, R])$
 $\delta(e, [(\Xi, m) B, B])(e, [(\Xi, m) B, B], [R, N, N])$
 $\delta(\Xi, [(n,m), B, B])(\Xi, [(n,m), B, B], [N, L, N])$
 $\delta(\Xi, [(n,m) x, B])(\Xi, [(n,m), B, B], [N, L, N])$
 $\delta(\Xi, [(n,m), B, B])(s, [(n,m), B, B], [R, N, N])$

Receive

$\delta(s, [(\Theta, m), B, B])(\Theta, [(\Theta, m) B, B], [R, N, N])$
 $\delta(\Theta, [(n,m), B, B])(\Theta, [(n,m), B, B], [N, L, N])$
 $\delta(\Theta, [(n,m), x, B])(\Theta, [(n,m), x, B], [R, N, N])$
 $\delta(\Theta, [(n,m) B, B])(s, [(n,m), B, B], [R, N, N])$
 $\delta(s, [(:, m), B, B])(s, [(:, m) B, B], [R, N, N])$
 $\delta(s, [(a,m), B, B])(s, [(a,m), B, m], [L, N, R])$
 $\delta(s, [(a,m), B, B])(s, [(a,m) B, w], [L, N, R])$
 $\delta(s, [(n,m) B, B])(v_{pr}, [(n,m) B, w], [L, N, N])$

Both together (only the communicating move)

$$\begin{array}{c} \vdots \\ \vdots \\ \delta(\Theta_{n2}, [(n2,m), x, B]) (\Theta_{n2}, [(n2,m), x, B], [N, R, N])_1 \\ \vdots \\ \vdots \\ \delta(\Xi_{n1}, [(n1,m), x, B]) (\Xi_{n1}, [(n1,m), B, B], [N, L, N])_2 \\ \vdots \\ \vdots \end{array}$$

We have stated before the components those are exchanging data among themselves can be ports of a single CPU or different CPUs connected by communication ports. But all of those should have an address.

Expressions: These are arithmetic operations with brackets and indices or condition checking. When an expression comes as a variable assignment statement a ';' is pushed (At the end it will be popped) then the expression is converted into a postfix form. Our expression will contain numeric values and operators. The numeric values will be a value in a box of tape. The operators will be:

- (: open parenthesis
-): closing parenthesis
- *: multiplication
- /: division
- +: addition
- -: subtraction

These operators have four levels of precedence.

- Highest: (,)
- Middle: *, -
- Low: +, -
- Lowest: < > =!

We will add one more operator to help with the processing of the program. That is a semicolon, ';', which will indicate the end of an expression.

To convert an expression from infix to postfix, we must first determine if the next (valid) item in the equation is a number or an operator.

If the item is a number, it is sent directly to the postfix equation. This is discussed in the variable part how to do it.

If the item is a operator, we use the stack to do one of the following:

- If the operator is: (, *, /, + or -
Pop the stack until either the stack is empty, the operator of (at the top of the stack or the top value on the stack has lower precedence than the current operator. As each operator is popped from the

stack, it is sent directly to the postfix equation. Then we push the current operator onto the stack.

$$\delta(e, [(o,m), B, B]) (e_o, [(o,m), B, B], [N, N, L])$$

$$\delta(e_o, [(o,m), B, u]) (e_o, [(o,m), u, B], [N, R, L])$$

$o \in \{+, -, *, /, <, >, =, !, \}$ $u \in \{+, -, *, /, <, >, =, !\}$ u has equal or higher precedence to o .

$$\delta(e_o, [(o,m) B, d]) (e_{oo}, [(o,m), B, d], [N, N, R])$$

$$\delta(e_{oo}, [(o,m), B, B]) (e, [(o,m), B, o], [R, N, R])$$

$d \in \{+, -, *, /, <, >, =, !, \}$ 'd' has lower precedence than o .

- If the operator is:)
Pop the stack until pop an open parenthesis, (. The closing parenthesis is NOT pushed on the stack. All operators that are popped (except for the open parenthesis) are sent directly to the postfix equation. If there is not an open parenthesis on the stack (i.e. you empty the stack), the equation has an error of an unmatched closing parenthesis.

$$\delta(e, [(), m, B, B]) (e, [(), m, B, B], [N, N, L])$$

$$\delta(e, [(), m, B, u]) (e, [(), m, u, B], [N, R, L])$$

$$\delta(e, [(), m, B, ()]) (e, [(), m, B, B], [R, N, N])$$

- If the operator is ';' pop the stack until ';' at the top, sending all popped operators to the postfix equation other than ';'. Evaluate the expression and save the result into variable. Then pop the ';'. In case the expression contains variables the variables are retrieved and the value is sent to postfix equation.

$$\delta(e, [(;), m, B, B]) (e_{v-p}, [(;), m, B, B], [N, N, L])$$

$$\delta(e_{v-p}, [(;), m, B, x]) (e_{v-p}, [(;), m, x, B], [N, R, L])$$

$$x \in \{+, -, *, /, <, >, =, !\}$$

$$\delta(e_{v-p}, [(;), m, B, ;]) (e_{v-r}, [(;), m, B, ;], [N, L, N])$$

$$\delta(e_{v-r}, [(;), m, u, B]) (e_{v-r}, [(;), m, u, B], [N, L, N])$$

$$\delta(e_{v-r}, [(;), m, B, B]) (e_v, [(;), m, B, B], [N, R, N])$$

Evaluating postfix equation: To evaluate a postfix expression, we will again need to use the stack. However, the stack will contain numeric values instead of operators.

- When a numeric value is encounter, the value is pushed on the stack.

- When an operator (other than ;) is encountered, two values are popped from the stack the operation is performed on these two values and the result is pushed onto the stack.
- When the no operand is encountered, there should only be one value on the stack above the ';'. This value is the result of the expression.

$$\delta(e_v, [(;, m), o, B]) (e_w [(;, m), B, B], [N, N, L])$$

$$o \in \{+, -, *, /, <, >, =, !\}$$

$$\delta(e_{v_o}, [(;, m), B, n]) (e_{v_o-e}, [(;, m), n, B], [N, N, L])$$

n = numeric value

$$\delta(e_{v_{o-e}}, [(;, m), n, n]) (e_v, [(;, m), B, non], [N, N, R])$$

$$\delta(e_v, [(;, m), B, B]) (e_{v_F}, [(;, m), B, B], [R, N, L])$$

$$\delta(e_{v_{-1}}, [(a, m), B, n]) (e_{v_{-2}}, [(a, m), n, m], [L, R, R])$$

$$\delta(e_{v_{-2}}, [(a, m), B, B]) (e_{v_{-3}}, [(a, m), B, W], [L, N, R])$$

$$\delta(e_{v_{-3}}, [(a, m), B, B]) (e_{v_{-4}}, [(a, m), B, B], [L, N, N])$$

$$\delta(e_{v_{-3}}, [(\eta, m), B, B]) (v_{pn}, [(\eta, m), B, B], [L, N, N])$$

The next moves from here is discussed in the variable part. For every expression we first push ';' as we told before, then convert the expression into postfix then evaluate, then pop ';'.

Conditional blocks: Conditional blocks are of two types

Block like if else statements in C when it reads an α it goes to expression state. In expression if it reads a β then it evaluates and the boolean result is in the third tape (stack) it also goes to the condition state. If the condition is false it goes to false state and do nothing until it goes to a normal state. ('s') in false state it only moves right but when reads an α it just push an α and when it finds ϵ it just pops the α . If it finds ϵ in false state and the stack-top is false then it pops the false and goes to normal state. If it reads χ then it goes to else part so if before it was false (there was false at the top) then it goes to true state else it goes to false state. The moves are as follows:

$$\delta(s, [(\alpha, m), B, B]) (e, [(\alpha, m), B, B], [R, N, N])$$

$$\delta(e_v, [(\beta, m), B, B]) (c, [(\beta, m), B, B], [R, N, L])$$

$$\delta(c, [(a, m), B, F]) (s_f, [(a, m), B, F], [R, N, N])$$

$$\delta(s_f, [(a, m), B, w]) (s_f, [(a, m), B, w], [R, N, N])$$

$$w \in \{F, \alpha\}$$

$$\delta(s_f, [(\alpha, m), B, F]) (s_w, [(\alpha, m), B, F], [R, N, R])$$

$$\delta(s_{r\alpha}, [(a, m), B, B]) (s_r, [(a, m), B, \alpha], [R, N, N])$$

$$\delta(s_f, [(\epsilon, m), B, \alpha]) (s_f, [(\epsilon, m), B, B], [R, N, L])$$

$$\delta(s_f, [(\epsilon, m), B, F]) (s, [(\epsilon, m), B, B], [R, N, N])$$

$$\delta(c, [(a, m), B, T]) (s, [(a, m), B, B], [R, N, N])$$

$$\delta(s, [(\chi, m), B, B]) (s_f, [(\chi, m), B, F], [R, N, R])$$

$$\delta(s_f, [(\chi, m), B, F]) (s, [(\chi, m), B, B], [R, N, N])$$

$$\delta(s, [(\epsilon, m), B, B]) (s, [(\epsilon, m), B, B], [R, N, N])$$

Block like switch statements in C: For some simplicity reason of this design this part is not covered.

Loops: When there is an instruction symbol for loop. It push the program tape number as return point. Then it checks the condition. Depending on the result of the check it goes to either true or false state and remains there till it gets end loop symbol. At end if it gets the end symbol while false state it pops the top (return point) else if it was in true state while reading the end loop symbol it goes back to the old point to check the condition again. The moves are as follows:

$$\delta(s, [(\Pi, m), B, B]) (e, [(\Pi, m), B, m], [R, N, R])$$

$$\delta(e_v, [(\Pi, m), B, B]) (c, [(\Pi, m), B, B], [R, N, L])$$

$$\delta(c, [(a, m), B, F]) (s_f, [(a, m), B, B], [R, N, L])$$

$$\delta(s_{if}, [(a, m), B, x]) (s_f, [(a, m), B, x], [R, N, N])$$

$$\delta(s_{if}, [(\Omega, m), B, x]) (s, [(\Omega, m), B, B], [R, N, N])$$

$$\delta(c, [(a, m), B, T]) (s, [(a, m), B, B], [R, N, N])$$

$$\delta(s, [(\Omega, m), B, B]) (s_f, [(\Omega, m), B, B], [L, N, L])$$

$$\delta(s_f, [(a, m), B, x]) (s_f, [(a, m), B, x], [L, N, N])$$

$$\delta(s_f, [(a, m), B, m]) (s, [(a, m), B, B], [N, N, N])$$

Function definition/calling: Function is defined before the main program starts. Unlike the variable definition, there is no move for function definition area. It is covered at S state before reading # and going to s state. Function is defined as follows:

$\gamma F_n \Phi (\Delta V_n \Phi \Delta V_m \eta \psi, \Delta V_n \Phi \Delta V_m \eta \psi, \dots) \Theta \{ \text{Body block again coded as the main program is coded.} \}$

The parameter variables are all by reference whether going in or out. That is there format is " $\Delta V_n \Phi \Delta V_m \eta \psi$ " Where:

- γ the left end symbol of definition,
- F_n is a function name,

- Φ symbol used as separator between function name and parameters,
- $\Delta V_n \Phi \delta V_m \eta \psi$ is a variable referenced by value details of which is already discussed at variable part,
- θ the left point of definition before the body.
- Then comes the body of function as sequences of code.
- Ω the last point of body.
- Each occupies one block of tape, as all of these are tokens.

Function is used as follows: (The moves are discussed below the text.)

$$\lambda F_n (\delta V_n \eta, \delta V_m \eta \dots) \sigma$$

Where:

- λ the left end symbol of definition,
- F_n is a function name,
- $\delta V_n \eta$ is the variable call which is already discussed at variable part.
- σ the left point of function call.

When TM read a Function call it goes to function call state and then it reach to the end of the call point ' σ ', then

push the current program point (block number of program tape), the box address as return point then it start moving left and push the parameters from the left side, at last push the Function name then it starts moving left going to state "finding function". When it finds it, (Read a Φ followed by the matched name. Point to be noted that when it read Φ it comes to know that there is a Function name at left, so it check the left box and if left box match the top-stack the definition is found.) it start moving right two boxes and then parameters are starting there. It first then push '#' then goes to the last part of definition and start coming back and pushing the whole function body until the starting of function. Then it goes to the end point of the tape, starts popping the whole function. When it pop '#' it assumes the next is parameters so it places parameters in appropriate places. Then it goes to the s state and start executing body. At the end (when it read Ω) it again goes to the return point from where it was called. We have told in the variable part that global variables using inside a function body must be defined before the function definition so there is no problem in global variable retrieval process.

Moves:

$$\begin{aligned} & \delta(s, [(\lambda, m), B, B]) (f, [(\lambda, m), B, B], [R, N, N]) \\ & \delta(f, [(a, m), B, B]) (f, [(a, m), B, B], [R, N, N]) \\ & \delta(f, [(\sigma, m) B, B]) (f_{pr}, [(\sigma, m), B, B], [R, N, N]) \\ & \delta(f_{pr}, [(a, m), B, B]) (f_{pr}, [(a, m), B, m], [L, N, R]) \\ & \delta(f_{pr}, [(\sigma, m), B, B]) (f_{pr}, [(\sigma, m), B, B], [L, N, N]) \\ & \delta(f_{pr}, [(\theta, m), B, B]) (f_{pr}, [(\theta, m), B, B], [L, N, N]) \\ & \delta(f_{pr}, [(\delta, m), B, B]) (f_{pr}, [(\delta, m) B, \delta], [L, N, R]) \\ & \delta(f_{pr}, [(\eta, m) B, B]) (f_{pr}, [(\eta, m), B, \eta], [L, N, R]) \\ & \delta(f_{pr}, [(\gamma, m) B, B]) (f_{pr}, [(\gamma, m), B, B], [L, N, N]) \\ & \delta(f_{pr}, [(\lambda, m), B, B]) (f_{if}, [(\lambda, m) B, B], [L, N, L]) \\ & \delta(f_{pr}, [(a, m), B, x]) (f_{pr}, [(a, m), B, x], [L, N, N]) \\ & \delta(f_{pr}, [(\phi, m), B, x]) (f_{pr}, [(\phi, m), B, x], [L, N, N]) \\ & \delta(f_{pr}, [(a, m), B, x]) (f_{pr}, [(a, m), B, x], [L, N, N]) \\ & \delta(f_{pr}, [(a, m), B, a]) (f_{pr}, [(a, m), B, \#], [R, N, R]) \\ & \delta(f_{pr}, [(a, m), B, B]) (f_{pr}, [(a, m), B, B], [R, N, N]) \\ & \delta(f_{pr}, [(\Omega, m), B, B]) (f_{pr}, [(\Omega, m), B, \Omega], [L, N, R]) \\ & \delta(f_{pr}, [(u, m), B, B]) (f_{pr}, [(u, m), B, u], [L, N, R]) \\ & \delta(f_{pr}, [(\gamma, m) B, B]) (f_{pr}, [(\gamma, m), B, \gamma], [R, N, R]) \\ & \delta(f_{pr}, [(x, m), B, B]) (f_{pr}, [(x, m) B, B], [R, N, N]) \\ & \delta(f_{pr}, [(B, m), B, B]) (f_{pr}, [(B, m), B, B], [N, N, L]) \\ & \delta(f_{pr}, [(B, m), B, u]) (f_{pr}, [(u, m), B, B], [R, N, L]) \\ & \delta(f_{pr}, [(B, m), B, \#]) (f_{pr}, [(B, m), B, \#], [L, N, N]) \\ & \delta(f_{pr}, [(u, m), B, \#]) (f_{pr}, [(u, m), B, \#], [L, N, N]) \\ & \delta(f_{pr}, [(\gamma, m), B, \#]) (f_{pr}, [(\gamma, m) B, B], [R, N, L]) \\ & \delta(f_{pr}, [(a, m), B, B]) (f_{pr}, [(a, m) B, B], [R, N, N]) \\ & \delta(f_{pr}, [(\delta, m), B, B]) (f_{pr}, [(\delta, m) B, B], [R, N, L]) \\ & \delta(f_{pr}, [(a, m) B, x]) (f_{pr}, [(a, m) B, B], [N, N, L]) \\ & \delta(f_{pr}, [(a, m) B, \delta]) (f_{pr}, [(a, m) B, B], [N, N, L]) \\ & \delta(f_{pr}, [(a, m), B, x]) (f_{pr}, [(x, m) B, B], [R, N, L]) \\ & \delta(f_{pr}, [(x, m) B, \eta]) (f_{pr}, [(x, m) B, B], [N, N, N]) \\ & \delta(f_{pr}, [(\theta, m), B, B]) (s, [(\theta, m), B, B], [R, N, N]) \\ & \delta(s, [(\Omega, m), B, B]) (s, [(\Omega, m), B, B], [L, N, L]) \\ & \delta(s, [(a, m), B, x]) (s, [(a, m), B, x], [L, N, N]) \\ & \delta(s, [(a, m), B, m]) (s, [(a, m), B, B], [N, N, N]) \end{aligned}$$

CONCLUSIONS

We have described one component of our whole Turing machine such components communicate among themselves by the moves described in the send receive section. This way our whole Turing Machine works as a model of distributed computing.

As future work, we may include more complex tasks such as object-oriented structure.

REFERENCES

1. Turing, A.M., 1936, On computable numbers with an application to the entscheidungsproblem (Descision Problem). Proc. London Math. Soc., 42: 230-265.
2. Hopcroft, J.E., R. Motwani and J.D. Ullman, 2000. Introduction to Automata Theory Language and Computation. 2/E, Addison-Wesley CO, USA.
3. Guetta, D., 2003. Turing Machine Development Environment. 19 Church Mount, London, N2 0RW, England.
4. Khiyal, M.S.H., 2004. Theory of Automata and Computation. National Book Foundation, Islamabad.
5. Stallings, W., 1999. Data and Computer Communications. 6/E, Prentice Hall, Inc, USA.
6. Silberschatz, A., P.B. Galvin and G. Gagne, 2004. Operating System Concepts. 7/E, John Wiley and Sons.