The South African Institute of Computer Scientists
and
Information Technologists

Proceedings

of the

1996 National Research and Development Conference

Industry meets Academia

Interaction Conference Centre, University of Natal,
Durban.
26 & 27 September

Edited by
Vevek Ram

ISBN 0-620-20568-7

Cover printed by Natal Printers (Pty) Ltd, Pietermaritzburg
Copying by the Multicopy Centre, University of Natal, Pietermaritzburg
Binding by Library Technical Services, University of Natal, Pietermaritzburg

FOREWORD

This book is a collection of papers presented at the National Research and Development Conference of the Institute of Computer Scientists and Information Technologists, held on 26 & 27 September, at the Interaction Conference Centre, University of Natal, Durban. The Conference was organised by the Department of Computer Science and Information Systems of The University of Natal, Pietermaritzburg.

The papers contained herein range from serious technical research to work-in-progress reports of current research to industry and commercial practice and experience. It has been a difficult task maintaining an adequate and representative spread of interests and a high standard of scholarship at the same time. Nevertheless, the conference boasts a wide range of high quality papers. The program committee decided not only to accept papers that are publishable in their present form, but also papers which reflect this potential in order to encourage young researchers and to involve practitioners from commerce and industry.

The organisers would like to thank IBM South Africa for their generous sponsorship and all the members of the organising and program committees, and the referees for making the conference a success. The organisers are indebted to the Computer Society of South Africa (Natal Chapter) for promoting the conference among its members and also to the staff and management of the Interaction Conference Centre for their contribution to the success of the conference.

On behalf of the Organising Committee
Vevek Ram
Editor and Program Chair
Pietermaritzburg, September 1996

Organising Committee

Conference General Chairs
Mr Rob Dempster and Prof Peter Warren (UNP)

Organising Chair
Dr Don Petkov (UNP)

Secretariat
Mrs Jenny Wilson

Program Chair
Prof Vevek Ram (UNP)

Program Committee

Prof Peter Wentworth, Rhodes
Dr Milan Hajek, UDW
Prof Derek Smith, UCT
Prof Anthony Krzesinski, Stellenbosch
Dr Don Petkov, UNP
Mr Rob Dempster, UNP
Prof Peter Warren, UNP

Table of Contents

Real-Time Systems

Database Systems

Education

Practice and Experience

Information Technology and Organizational Issues

Abstracts

List of Contributors

Vladimir B Bajic
Centre for Engineering Research,
Technikon Natal,
P O Box 953
Durban 4000

C N Blewett
Department of Accounting
University of Natal
King George V Avenue
Durban 4001

Justin Cansfield
Department of Accounting
University of Natal
King George V Avenue
Durban 4001

Tom Considine
Apron Services (Pty) Ltd
P O Johannesburg
International Airport
1600

Eric Cloete
School of Electrical Engineering
Cape Technikon
Box 652
Cape Town

I Cloete
Computer Science Department
University of Stellenbosch
Stellenbosch
7600

O A Daini
Department of Computer Science
University of Botswana
Gaborone
Botswana

Nirvani Devcharan
Umgeni Water
Box 9
Pietermaritzburg
3200

P J A de Villiers
Department of Computer Science
University of Stellenbosch
Stellenbosch
7700

Ruth de Villiers
Department of Computer Science and
Information Systems
UNISA
Box 392, Pretoria, 0001

G J Erwin
Business Information Systems
University of Durban-Westville
Private Bag X54001
Durban 4000

G Ganchev
Computer Science Department
University of Botswana
PBag 0022
Gaberone, Botswana

J Geldenhuys
Department of Computer Science
University of Stellenbosch
Stellenbosch
7700

Louise Gibson
BIS, Dept Accounting & Finance
University of Durban
Pvt Bag X10
Dalbridge 4014

Mike Hart
Department of Information Systems
University of Cape Town
Rondebosch
7700

M. Hajek
Department of Computer Science
University of Durban-Westville
Pvt Bag X54001
Durban 4000

A C Hansen
Dept of Agricultural Engineering
University of Natal
Private Bag X01
Scottsville 3209

J M Hattingh
Department of Computer Science
Potchefstroom University for CHE
Potchefstroom 2520

Boris Jankovic
Centre for Engineering Research
Technikon Natal
P O Box 953,
Durban 4000

Paula Kotze
Department of Computer Science and
Information Systems
UNISA
Box 392
Pretoria, 0001

J W Kruger
Vista University
Soweto Campus
Box 359
Westhoven 2124

A C Leonard
Dept of Informatics
University of Pretoria
Pretoria
2000

Ben Laauwen
Laauwen and Associates
P O Box 13773
Sinoville
0129

Mari Le Roux
Information technology, development: project
leader
Telkom IT 1015
Box 2753
Pretoria 0001

P W L Lyne
Dept of Agricultural Engineering
University of Natal
Private Bag X01
Scottsville 3209

Rose Mazhindu-Shumba
Computer Science Department
University of Zimbabwe
Box MP167
Harare, Zimbabwe

Meredith McLeod
Centre for Engineering Research,
Technikon Natal,
P O Box 953
Durban 4000

D Moodley
Computer Management Systems
Box 451
Umhlanga Rocks
4320

Andrew Morris
P O Box 34200
Rhodes Gift
7707

R C Nienaber
Technikon Pretoria
Dept of Information Technology
Private Bag X680
Pretoria 0001

E Parkinson
Department of Computer Science
University of Port Elizabeth
Box 1600
Port Elizabeth 6000

Don Petkov
Department of Computer Science and
Information Systems
University of Natal
PBag x01
Scottsville 3209

Olga Petkov
Technikon Natal
Box 11078
Dorpspruit 3206
Pietermaritzburg

N Pillay
Technikon Natal
Box 11078
Dorpspruit 3206
Pietermaritzburg

V Ram
Department of Computer Science and
Information Systems
University of Natal
PBag x01
Scottsville 3209

Melinda Redelinghuys
Department of Computer Science and
Information Systems
UNISA
Box 392
Pretoria, 0001

Karen Renaud
Computer Science and Information Systems
UNISA
Box 392
Pretoria, 0001

H N Roux
Department of Computer Science
University of Stellenbosch
Stellenbosch
7700

T G Scott
Department of Computer Science
Potchefstroom University for CHE
Potchefstroom
2520

T Seipone
Department of Computer Science
University of Botswana
Gaborone
Botswana

Derek Smith
Department of Information Systems
University of Cape Town
Rondebosch
7700

Anette L Steenkamp
Department of Computer Science and
Information Systems
UNISA
Box 392
Pretoria, 0001

T Steyn
Department of Computer Science
Potchefstroom University for CHE
Potchefstroom 2520

H. Suleman
Department of Computer Science
University of Durban-Westville
Pvt Bag X54001
Durban 4000

A Vahed
Department of Computer Science
University of Western Cape
Private Bag X17
Bellville 7530

A Van der Merwe
Computer science and Informations Systems
UNISA
P O Box 392
Pretoria,0001

Tjaart J Van Der Walt
Foundation for Research and Development
Box 2600
Pretoria, 0001

K Vavatzandis
Department of Information Systems
University of Cape Town
Rondebosch
7700

Y Velinov
Dept Computer Science
University of Natal
Private Bag X01
Scottsville 3209

H Venter
Department of Computer Science
University of Port Elizabeth
Box 1600
Port Elizabeth 6000

H L Viktor
Computer Science Department
University of Stellenbosch
Stellenbosch
7600

R Von Solms
Department of Information Technology
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth 6000

A J Walker
Software Engineering Applications
Laboratory
Electrical Engineering
University of Witwatersrand
Johannesburg

P Warren
Computer Science Department
University of Natal
P/Bag X01
Scottsville 3209

Max Watzenboeck
University of Botswana
Private Bag 0022
Gaberone
Botswana

TEACHING TURING MACHINES - LUXURY OR NECESSITY?

Y.Velinov
Department of Computer Science and Information Systems
University of Natal, Pietermaritzburg
yuri@cs.unp.ac.za

Abstract

The purpose of this paper is to outline an approach for the development of the Tape Machines model in the Theory of Computation which is bound more closely to the real computer world. The approach is based on a developed software consisting of a Tape Machine Simulator and a Tape Machine Assembler. It gives possibility to achieve the aims of the theory without any loss of mathematical rigor but in more natural, useful and attractive manner.

Introduction

Tape Machines were introduced by A. Turing [1936] as a theoretical model of the concept of computation several years before the real electronic computers appeared in practice. The hypothetical devices originally considered by A. Turing (Tape Machines known now as Turing Machines) are appealing because of their simplicity, transparency and completeness. On one hand, they embrace all aspects of the algorithmic processes (the concept of data, the concept of a control device, the concept of an algorithmic language), and on the other hand they convey a strong feeling of a mechanistic device working by itself and independent of any human intellectual activity.

Nowadays, Tape Machines in their different forms are an indispensable pedagogical vehicle in contemporary courses in the Theory of Computation. However, they and the ways the theory around them is developed suffer some disadvantages. The first thing to note is that the original Turing machines are extremely clumsy when really interesting computations have to be performed. There are explicable reasons for that. From a purely theoretical point of view any model of computation must face two discrepant requirements:

- The model must be as simple as possible in order to be convincing as a mechanical approach and also in order to be easy to simulate it by other models of computation.
- The model must have as much expressive power (not just computational power) in order to deal easily with the real computations and also in order to be able to simulate in an easy way other models of computation.

Every model of computation balances these two requirements. Initially the first requirement was considered more important (mainly for the model to be convincingly mechanistic) and the original Turing machines were accepted as more sound. Typically, the theory developed around them consists of descriptive languages for building Turing machines and series of lengthy (and I dare say, cumbersome) theorems proving step by step what can be computed or modeled, where in the heap of details it is quite easy to lose the essential ideas. Later on, sacrificing simplicity to gain more convenience some other machine models, which move the balance from the first requirement to the second one, were developed [Shepherdson, Sturgis 1963, Minsky, 1967]. Regardless of their attractiveness I still find the models unadequate for a very simple reason: developed historically in the frame of pure mathematics and following its traditions, these models and the theory around them are a successful, but not very comfortable attempt to build the theory of computation outside of the existing nowadays in reality computing phenomena. As a result the courses in the Theory of Computation in the Computer Science syllabi are frequently perceived as stand alone impractical outsiders which are included there only to show some respect to the achievements of the mathematicians before the real computer science came to life. This unsatisfactory state was discussed before (see for example [Brady 1974]) but looking at the nowadays textbooks I still cannot see any significant changes in this respect.

The purpose of this paper is to o of Tape Machines which is bound more closely to the real computer world. I will try to show that without any loss of mathematical rigor the aims of the theory can be achieved in a more acceptable and natural manner. That if we wish we can even rise the level of rigor. Finally, that the theory can outline ideas used in the real computer practice.

Tape Machines

If the first of the requirements described above is regarded as essential the initial machine model must be as simple and as transparent as the original Turing Machines are. This together with the fact that writing programs is considered as the most fundamental task in Computer Science makes the machine proposed by Wang [1957] a natural candidate. In the further considerations I will accept as a groundwork the modification of the Wang's machine which can work on arbitrary symbols.

A *Tape Machine* (fig.1) consists of three components: a data tape, a program tape and a processor.

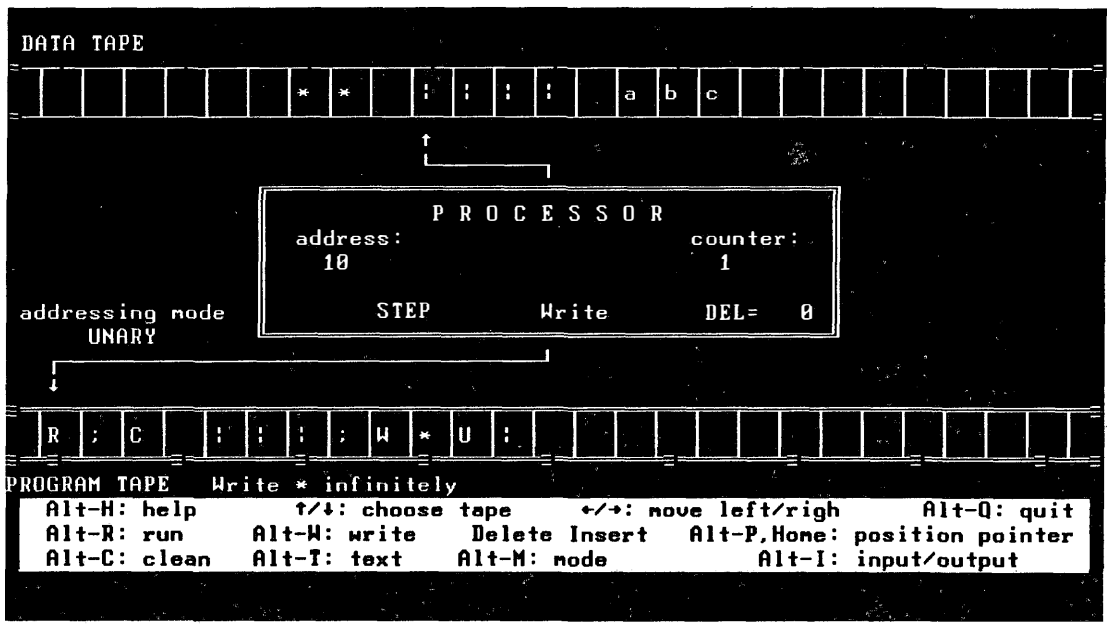


Fig.1.

The *data tape* is infinite in both directions. Each of its cells can be empty (_denotes the empty symbol) or can contain exactly one symbol from a fixed alphabet $\mathcal{A}$. Data, properly organized, are situated on the otherwise empty data tape symbol by symbol. When the machine works the size of the area which contains nonempty cells may change but is always finite.

The *program tape* has a beginning and is infinite in only one direction. Each of the cells of the program tape can contain exactly one of the symbols from the alphabet

$$\{\text{'R'}, \text{'L'}, \text{'W'}, \text{'C'}, \text{'U'}, \text{'|'}\} \cup \mathcal{A} \cup \{\_\}.$$

The purpose of the program tape is to store the instructions for the processor - the *machine commands*. The machine commands are the symbols 'R', 'L' and the strings from the sets $\{\text{'U'}\}\blacklozenge\{\text{'|'}\}^*$, $\{\text{'W'}\}\blacklozenge\mathcal{A}$ or $\{\text{'C'}\}\blacklozenge\mathcal{A}\blacklozenge\{\text{'|'}\}^*$ (here $\blacklozenge$ stands for the concatenation operation). We call the symbols 'R', 'L', 'W', 'C' and 'U' the command codes; U is the code for unconditional jump, C is the code for conditional jump, W is the code for write operation, R is the code for right movement and L is the code for left movement. Only a finite number of cells in the programming tape can contain nonempty symbols.

A continuous sequence of machine commands is called a *machine program.* A machine program must be situated on the program tape symbol by symbol starting from the first cell, with no empty cells between the commands. All unused by the program cells on the program tape must be empty.

The commands in a program are naturally numbered in the order in which they appear with consecutive positive integers starting from 1. The ordinal numbers of the commands in a program are used by the processor when implementing the jump commands.

The *processor* works in discrete time. The time scale is represented by the sequence of the natural numbers. At every moment of its work the processor observes exactly one cell of the data tape and one cell of the program tape. It can move the observation point on the data tape one cell to the left or to the right or it can write in the observed cell any symbol from $\mathcal{A} \cup \{\_\}$. It can also move the observation point on the program tape one cell to the right, or one cell to the left if the observed cell is not the first one. The processor is organized in such a way that it follows the commands written on the program tape. At the beginning the processor observes the first cell of the program tape. If it has just started working or it has just executed a command the observed cell on the program tape contains a command code. The processor analyzes the command moving if necessary the observation point on the program tape to the right cell by cell until next command code is reached. If a syntactically well formed command is recognized the processor interprets it implementing some actions according to the type of the command as follows:

- If the command code R is observed at some moment then at that moment the processor moves the observation point on the data tape one cell to the right and moves the observation point on the program tape one cell to the right to analyze the next command at the next moment.
- If the command cod L is observed at some moment then at that moment the processor moves the observation point on the data tape one cell to the left and moves the observation point on the program tape one cell to the right to analyze the next command at the next moment.
- If the command code W is observed at some moment the processor moves the observation point on the program tape one cell to the right to see the symbol à situated there, then it changes the content of the observed cell on the data tape to à and once more moves the observation point on the program tape one cell to the right to analyze the next command on the tape at the next moment.
- If a command code U is observed the processor analyzes the content of the following cells one after the other until a new command code is encountered. In the event of a syntax error the processor stops. If a syntactically correct command - U with n symbols | after it, has been recognized the general reaction of the processor to the command (we do not specify in details how it reacts at each moment) is to move the observation point on the program tape at the beginning of the n-th command to process it at the next moment.
- If a command code C is observed the processor analyzes the content of the following cells sequentially until a new command code is encountered. In any case of syntactical incorrectness the processor stops. If a syntactically correct command - Cα with n symbols | after that, has been recognized the general reaction of the processor is as follows:
  - in the case when à is observed on the data tape move the observation point on the program tape at the beginning of the n-th command;
  - otherwise move the observation point on the program tape to the beginning of the next command.
- The processor stops working if at the very beginning or just after the implementation of any command a symbol other than a command code is observed.

One more possible step in order to make the Tape Machine more sound and correlated with the other topics in Computer Science could be to describe the processor as a finite automaton. Following this direction it deserves also to consider the relations between the Tape Machine and the original Turing Machines. The "program" part of a Turing Machine is nothing else but a description of a finite automaton, and we usually consider finite automata as hardware structures. It is also possible to show that each program for a Tape Machine can be translated into a program for a Turing Machine and vice versa. This underlines the important fact that the programs (with no recursive calls) are equivalent to the hardware constructions.

The imaginary Tape Machine described above can be brought to life in the form of a simulator on a real computer. Fig.1 shows the appearance of the screen of a simulator implemented for the IBM-PC computers.

Assembler for Tape Machines

Having a simulator at hand it is easy to demonstrate how inconvenient it is to write and change machine programs. This naturally justifies a next step - the introduction of an assembly language and an assembler for the Tape Machine. A simple form of an assembly language for the Tape Machine is described below.

A program written in the assembly language for Tape Machines consists of a sequence of commands each one written on a separate line. The commands have a standard format consisting of 5 fields of fix ed length though some of them can be empty:

LABEL	CODE OF OPERATION	SYMBOL	LABEL/NUMBER	COMMENTS
4 symbols	3 symbols	1 symbol	4 symbols	15 symbols

In order to identify command lines (necessary for the conditional or unconditional jump operations) the assembly language uses strings of symbols called "labels". A label is a string which begins with a character and contains no more than 4 characters. The LABEL FIELD may contain any label but no two different lines may have the same label in it. A label in the label field of a line identifies the command in it. The SYMBOL FIELD may contain any symbol accessible on the keyboard of an ordinary computer. The CODE OF OPERATION FIELD may contain any of the strings 'LFT', 'RGT', 'WRT', 'BEQ', 'BNE', 'JMP', 'HLT' which are the operation codes. The ADDRESS/NUMBER FIELD may contain an label or a four digits positive integer number. The COMMENTS FIELD may contain any sequence of symbols. It is not significant for the operation of the program an can always be left empty.

The content of the different fields of a command line depends on the used operation code. The meaning of the operations together with the format they require is described bellow, where '*nnnn*' denotes a four-digits number, '*mmmm*' and '*llll*' - labels, '*x*' - a symbol, and '_' - an empty space for a symbol:

mmmm LFT _ *nnnn* - Move the observation point on the data tapes *nnnn* cells to the left.

mmmm RGT _ *nnnn* - Move the observation point on the data tapes *nnnn* cells to the right.

mmmm WRT *x* ____ - Write the symbol x on the data tapes.

mmmm BEQ *x* *llll* - If the observed symbol on the data tape is x continue the execution of the program with the command labeled by *llll* or otherwise continue with the next command.

mmmm BNE *x* *llll* - If the observed symbol on the data tape is not x continue the execution of the program with the command labeled by *llll* or otherwise continue with the next command.

mmmm JMP _ *llll* - Continue the execution of the program with the command labeled by *llll*.

mmmm HLT _ ____ - Stop the execution of the program.

The commands of the assembly language are executed one after the other in the natural order (from top to bottom) of the lines they occupy, except when commands with code of operation JMP, BEQ or BNE are encountered. Such commands change the natural order of execution of the commands according to the meaning of the operations as it was described above.

It is necessary to show further that the programs written in the assembly language can be translated into the machine language. Instead of giving a formal proof of this fact it is better to construct an assembler. A two pass assembler seems more simple and convenient for the purpose. The assembler itself should not be constructed on a tape machine. It should be considered only on algorithmic level and of course could be designed in a high level language. The algorithm for its work stands for a proof of the translatability. Moreover, the proof of the translatability can be supported further with a proof of the correctness of the assembler algorithm. Fig.2 shows the appearance of the screen of an implementation of the described assembler for the IBM-PC computers.

Though applied to a very simple assembly language, except for the subroutine calls which cannot be introduced at this stage, the assembler for Tape Machines covers the most significant features of a real assembler. For a more deep understanding of the assemblers structure, if this is regarded as essential, a more rich assembly language with more commands can be introduced. Macrodefinitions and even more - relocatable code together with linkers and loaders may also be considered.

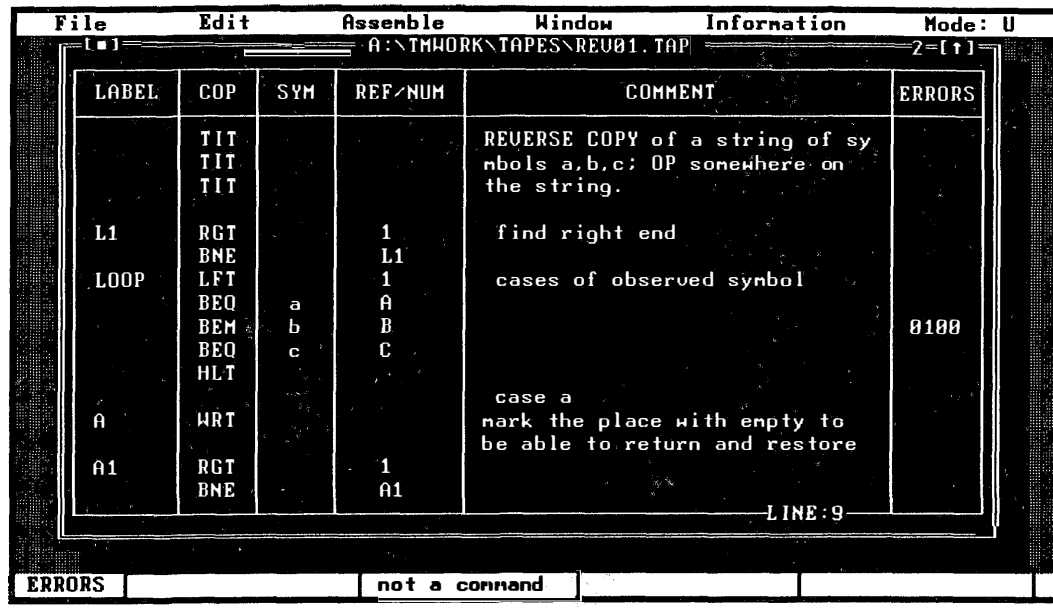


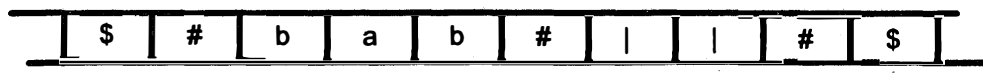
Fig.2.

Assembly language level is not considered in the standard texts concerning Turing Machines. Usually, only remarks that without really increasing the computational power of the machines some additional commands can be introduces are present. The introduction of an assembly language and an assembler not only relieves the exposition of the theoretical results but also combines the needs of the theory with the practical aspects of Computer Science.

Register Tape Machines

The topic can be developed further by introducing a higher level language such as the language of flow diagrams considered by Hermes [1965], but it is better to continue in another direction - using the assembly language to design a virtual Register Machine in order to incorporate the approaches of Shepherdson-Sturgis [1963] or Minsky [1967].

A Register Tape Machine uses unlimited but finite number of registers instead of cells. Each register itself is a tape infinite in one direction, which can contain an infinite chain of symbols. The registers can be simulated on the data tape of the ordinary Tape Machine with the aid of two additional servicing symbols, say '#' and '$'. The symbol # is used to separate the different registers and $ - to indicate the beginning and the end of the field of registers on the data tape. On the data tape the registers are situated attached, one immediately after the other. They can be distinguished by their ordinal numbers or by appropriate names associated with them. The next figure shows a data tape with 2 registers containing some data.



Since each of the the registers must be potentially infinite if a symbol is to be inserted in a register at a certain place a free cell is created there by shifting all the cells till the end of the register field one

cell to the right. If a symbol in a register is to be deleted at a certain place all the cells from that place to the end of the register field are shifted one cell to the left. Therefore, though implemented by a single infinite tape, the virtual Register Machine appears to the user as a list of infinite tapes - registers each one distinguished by a name or a number.

The language for the Register Machine can be constructed directly to be of assembly type. The commands could be of the same format as the format for the commands of the assembly language for Tape Machines. The minimal set of commands must include a command of the type DR *xxxx* (define register) used to introduce a new register on the data tape and associate it with the name '*xxxx*', commands for attaching a symbol at the left (right) side of a register and conditional jump commands depending on the first observed symbol in a register (or distinguishing an empty register). More complex commands can be introduced to compare the content of any two registers. Furthermore, the natural numbers can be represented in unary code as sequences build using a fixed symbol and stored in a register. Then commands for arithmetical operations can be considered or simulated. At this stage all significant commands can be implemented by appropriate sequences of ordinary Tape Machine commands to justify the claim for virtuality of the Register Machine. Further, in order to make the virtual Register Machine more convenient, in analogy with the real computers, some standard resisters can be introduced. For example an accumulator and a stack. The stack gives further possibility to introduce subroutines, a mechanism for recursive calls, mechanisms for passing parameters and so on. A higher level language can also be introduced if necessary.

With the virtual Register Machine at hand the development of the topic can continue to reach the typical goals of the theory of computation (like proving the equivalence of different models of computation, or establishing validity of important theorems like the Theorem of the three indexes or the Theorem of Recursion) but in different stile - by designing programs instead of by proving theorems. And the level of rigor can be raised by proving the programs correct.

Conclusions

Let us summarize what can be achieved if the approach described above is followed.

First of all, on the expense of building several levels of languages and machines, and considering their interrelations the theoretical results can be achieved more easy. The formal troubles in most of the proves are shifted in advance on the constructions of the assembly language or higher level languages and their translation to the machine language. As a result the proves themselves become more clear and communicate better the underlying ideas. This can be expected - the introduction of several levels of languages reflects the popular in the Computer Science methodological principle "divide and conquer" developed there under the pressure of the problem of solving real complicated problems.

Further, the stile of the proofs is unified and consists of construction of programs and proving them correct.

Finally, building an assembly language and the corresponding assembler, constructing a virtual machine and a language to deal with it is beneficial by itself for the computer science education. The ideas of how an assembly language is designed and how an assembler is constructed are developed in a simple way and become more clear and understandable. This gives opportunity to introduce them at the early stage of education.

As a whole this approach brings nearer the theory and the practice in the Computer Science to the benefit of both. But there is also another reason supporting it. The variety of ready available high level languages on one hand and the complexity of the contemporary real assembley languages on the other lead to a decreasing interest in studying the assembley language level. There is a tendency to elude assembley language courses in the software directed syllabi. As a result the students lose the touch with the low level programming. The proposed approach is an alternative which gives possibliliy to incorporate it into the theoretical courses.

The ideas described above were experimented quite successfully, at different stages of their development, with first and second year students. As it could be expected the introduction of simulators made the topic much attractive. The students enjoyed playing with the software supporting the course and prefered it to the blackboard presentations. But playing they succeeded to rich faster more deep understanding of the material.

References

Brady, J.M.,
A Programming Approach to Some Concepts and Results in the Theory of Computation, The Computer Journal, V19, N3, p.234, (1974)

Hermes, H.,
Enumerability, Decidability, Computability., Springer (1965.)

Minsky, M.,
Computation: Finite and Infinite Machines., Prentice-Hall, (1967).

Shepherdson,J.C., Sturgis,H.E.,
Computability of Recursive Functions., JACM,Vol.10, p.217-255 (1963).

Turing, A.,
On Computable Numbers With an Application to the Entscheidungsproblem., Proc. London Math Soc. Ser.2, 42 (1936).

Wang Hao,
A variant to Turing's Theory of Computing Machines., JACM, 4, 1 (1957).