DEVELOPMENT OF A PROGRAMMING LANGUAGE FOR MICRO-COMPUTER BASED LEARNING SYSTEM

by

SHAH MOHAMMAD REZAUL ISLAM

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN ENGINEERING (COMPUTER)

DEPARTMENT OF COMPUTER ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DHAKA, BANGLADESH

CERTIFICATE

This is to certify that this work has been done by me and and it has not been submitted elsewhere for the award of any degree or diploma.

January, 1985. Shah Mohammad Rezaul Islam

DEVELOPMENT OF A PROGRAMMING
LANGUAGE FOR
MICRO-COMPUTER BASED LEARNING SYSTEM

A THESIS

by

Shah Mohammad Rezaul Islam

Accepted as satisfactory for partial fulfilment of the requirements for the degree of M. Sc. Engineering in Computer Engineering.

EXAMINERS

1. _____

Dr. Syed Mahbubur Rahman
Supervisor & Chairman
Assistant Professor
Department of Computer Engineering

2. _____

Dr. A.K.M. Mahfuzur Rahman Khan
Member
Head & Professor of the
Department of Computer Engineering

3. _____ 5/1/85

Dr. Shamsuddin Ahmed
Member
Dean & Professor of the
Faculty of Electrical & Electronic
Engineering

4. _____ 5/1/85

Mr. Hanifuddin Miah
External Member
Chief Scientific Officer &
Director, Computer Services
Bangladesh Atomic Energy Commission

ACKNOWLEDGEMENT

The author expresses his indebtedness and deep sense of gratitude to Dr. Syed Mahbubur Rahman, Assistant Professor of the Department of Computer Engineering, Bangladesh University of Engineering and Technology, Dhaka for his continuous guidance, valuable suggestions, constant encouragement and help all along the course of this work.

The author expresses his greatfulness and thanks to Dr. A.K.M. Mahfuzur Rahman Khan, Professor & Head, Department of Computer Engineering, who was kind enough to provide all the facilities of the department for this work.

The author wishes to give whole hearted thanks to Mr. Kamal for taking the pain and care of typing this thesis.

The author wishes to thank the staff of Computer Centre, Bangladesh University of Engineering & Technology, for their help and co-operation during this work.

A B S T R A C T

Specification for the Programming Language of Microcomputer Based Learning System is presented. Learning system has been analyzed from a systemic point of view. Formal definitions and specifications for a Learning System is also presented.

Specifications for the language are based on hierarchical decomposition of the system into elementary abstract machines, where the machines are defined using by object model concept.

Specification for the language made it highly structured and modular. The essentials of the systax are divided into three parts, i. Execution Atom (EA), which is the smallest execution module of the system, ii. Operative Function, which defines the operation that will be performed when evoked by EA, iii. Domain, which is the set of tuples containing small self contained program unit.

Although defined and designed for microcomputer based learning applications, the language can be used for programming any real time applications as well.

C O N T E N T S

Chapter I	INTRODUCTION	I-1
1.1	GENERAL	I-1
1.2	OBJECTIVE	I-1
1.3	ENVIRONMENT and SYSTEM	I-2
	1.3.1 Authoring Aids	I-2
	1.3.2 Evaluation of CAI Courseware	I-2
	1.3.3 Computational and other Hardware Capability	I-3
1.4	CONCLUSION	I-7
Chapter II	MICROCOMPUTER BASED LEARNING SYSTEM and HUMAN LEARNING	
2.1	INTRODUCTION	II-1
	2.1.1 Microcomputer Aided Instruction	II-1
	2.1.2 Microcomputer Managed Instruction	II-1
	2.1.3 MLS Variation	II-2
2.2	HUMAN LEARNING	II-2
	2.2.1 Individualized Learning System	II-3
2.3	LEARNING SYSTEM	II-3
	2.3.1 Human Learning Process	II-3
	2.3.2 Characteristics of Learner	II-4
2.4	CURRICULUM and INSTRUCTION	II-4
2.5	CONCLUSION	II-5
Chapter III	FORMAL VIEW OF A LEARNING SYSTEM & ITS SPECIFICATION	
3.1	INTRODUCTION	III-1
3.2	COURSEWARE DEVELOPER	III-1
3.3	FORMAL VIEW OF THE SYSTEM	III-2
	3.3.1 Assertion Language	III-3
	3.3.2 Format of Formal Specification	III-5
3.4	FORMAL DESCRIPTION OF THE PHYSICAL SYSTEM	III-10
3.5	CONCLUSION	III-12

Chapter IV FORMAL METHODOLOGY FOR THE DESIGN OF LSL AND ITS SPECIFICATION

4.1 INTRODUCTION IV-1

4.2 METHODOLOGY FOR DESIGN and IMPLEMENTATION IV-2

4.2.1 Stepwise Implementation IV-4

4.3 DESIGN APPROACH OF LSL IV-7

4.4 TYPE MANAGERS IV-8

4.4.1 Type Managers for Courseware IV-9

4.5 CONCLUSION IV-13

Chapter V COMMAND PROCESSOR & USER INTERFACE

5.1 INTRODUCTION IV-1

5.2 USER INTERFACE IV-1

5.2.1 Student Command Set V-1

5.2.2 Course Developer Command Set V-2

5.3 SYNTAX CONVENTION V-3

5.4 CONCLUSION V-6

Chapter VI DESCRIPTION OF LEARNING SYSTEM LANGUAGE

6.1 INTRODUCTION VI-1

6.2 EXECUTION ATOM VI-1

6.2.1 Nested Execution Atom VI-2

6.3 PROGRAM UNIT VI-3

6.4 RUNNING THE SYSTEM VI-4

6.5 CONCLUSION VI-4

Chapter VII DISCUSSION & RESULTS VII-1

CHAPTER - I

INTRODUCTION
1.1 GENERAL

The recent dramatic advances in microcomputer technology have made computers viable for learning applications. Coupled with their drastic price reduction they are available in almost every sphere of education and research.

So far education has been mostly based on traditional class room lectures and print media. Books and periodicals are oritical repositories of knowledge and the basis of most information interaction. With the emergence of computer technology, the information basis has shifted from print to electronic media. Today computer, laser visiondisk, diskettes, tapes, microfilms and fast interactive facilities for accessing information are easily available even for the microcomputers.

1.2 OBJECTIVES

Several research workers have proved[8] microcomputer aided instructions to be very helpful to the teachers to teach and to the students to learn. However most of the authoring programs for the microcomputers are written in BASIC. For a complex subject and complicated graphics, this authoring program itself may be very large and complex. For very large and complex authoring program, we generally have very special types of programmers which we call mCAI programmer.

For a good and efficient authoring program, there must be a very good understanding between the programmer and the teacher who is designing the courseware. To eliminate the need of mCAI programmers, we have developed the formal specification of a language, which we call learning system language (LSL).

The basic design approach of LSL is based on the concept of object model, where the objects are defined in the formal view of the system. The operation of the system may be viewed as manipulation of these objects organized as a hierarchically ordered collections of abstract machines.

With the help of specification language we formally describe the behavior of each of these abstract machines. We also define specification for syntatical part of the system which will be a interface between a command processor and end user.

1.3 ENVIRONMENT and SYSTEM

Before we approach to the system, we define the environment and interrelations between the environment[21] and the system.

The major components of environment are,

i. Courseware developer
ii. Student or learner

Physical specifications of the system includes

i. Authoring aid
ii. Computational capability
iii. Graphic capability
iv. Multisensory Input/output control

1.3.1 Authoring Aids

Authoring aids incorporate courseware for CAI, evaluation of CAI courseware and its verification.

Courseware for CAI

Specification for a particular learning task is designed by the courseware developer. This specification is then coded in LSL for real implementation. It is responsibility of courseware developer to ensure that the courseware attains required pedagogical capability for a particular application.

1.3.2 Evaluation of CAI courseware

Courseware evaluation is very important from instructional viewpoint, normally it requires various kinds of information for proper evaluation.

It is important to find whether the amount and difficulty of the content, the sequence, and the degree of computer interactions are appropriate for the learners. It is also important to know how far it is compatible with the students attitude and competence, and how it can help them attain good performence.

Evaluation is normally divided into two groups,

i. Formative
ii. Summative

Formative evaluation is on going during instructional process and summative evaluation occurs at or after delivery of instruction. The evaluation scheme may also incorporate facility for user's assessment of the courseware for later author review.

Classification of CAI according to learning environment.

According to target user (learner) and learning environment CAI may be classified as follows ,

i. CAI in school environment for school going students.
ii. CAI in adult and non-formal education.
iii. CAI for design & development.
iv. CAI for continuing education in industries.
v. CAI for handicapped persons.

Specification of LSL should be such that it will be able to handle varied target user and learning environment.

1.3.3 Computational and other Hardware capability.

Computational and physical facilities some time determine several important aspect of the implementation of learning system. Hardware capabilities may be viewed from three different angles.

i. Hardware facilities as seen by the student or learner.

ii. Hardware facilities as seen by a particular courseware.

iii. Hardware facilities as seen by the courseware developer.

Learners view of Hardware

Learner accept information from the system and returns some reaction to the system. A learner normally view all hardware around him as a port for information exchange with the system.

Learner receive information from

i. Video Display Terminal

for presentation of

TEXT {size,colour,total characters in full screen, window}

GRAPHICS {colour,resolution,pixels,scroll,window}

ii. Printer

for presentation of hardcopy

MATRIX {matrix size,speed,graphics capability, paper feed,total vertical pins,software control}

GRAPHICS & PLOT {colour,resolution,size,pixel,speed}

iii. Audio Sound Generator

for sound communication

VOICE & TONE {voice synthesizer,tone generator, vocabulary,natural language support,pitch}

iv. User Defined

Exceptional USER DEFINED SERVICE {user defined hardware support}

Learner send his response through

i. Key board

TYPING	{standard or special key board, numeric key pad, function keys, key buffer, overlays}
TOUCH	{standard or special key layout, numeric key pad, key buffer}

ii. Screen

TOUCH	{finger size, screen manipulation, software support}
LIGHTPEN	{select, drawing, colour}

iii. Cards

PUNCH	{nos. of row, nos. of column, character set support}
OPTICAL	{nos of row, nos of column, character set}

iv. User peripherals

JOYSTICK	{horizontal, vertical, down, up}
DIGITIGER	{resolution, maximum grid point}
MOUSE	{resolution, degrees of freedom}
USER DEFINED	{device defined by the courseware developer}

Hardware facilities available at learner level is a subset of those of courseware level. Hardware facilities as seen by the courseware is a subset of the facilities as seen by the courseware developer. Courseware developer has higher access right for the hardwares than the courseware. These access rights are defined in the access vector of the capability module which will be discussed in chapter-IV.

Additional hardware facilities those seen by courseware or courseware developer,

i. Memory

MAIN	{k bytes, speed, addressibility}
BACKUP	{floopy, harddisk, nv-RAM, cassettes, V.C.R}

ii. Processor

SPEED	{clock cycle, clock speed}
INSTRUCTION SET	{instruction set, micro code, length, addressing mode}

iii. Communication

PERIPHERAL	{parallel, serial, LAN}
MULTIUSER	{network architecture, controller}

1.4 CONCLUSION

As has been stated earlier, the specification for the language will be made on the basis of hierarchical decomposition on the system in to level of functional abstractions. We will use object model concept through out our specification as a basis of these abstractions. Besides predicate calculus we will be using a special notation for defining syntatic and semantic aspect of the language.

CHAPTER - II

MICROCOMPUTER BASED LEARNING SYSTEM AND HUMAN LEARNING

2.1 INTRODUCTION

Microcomputer based learning System (MLS) refers to any tutoring and testing with the assistance of microcomputer. The domain of microcomputer based learning may be segmented into two subfields,[5,8,9]

i. Microcomputer Aided Instruction (mCAI)
ii. Microcomputer Managed Instruction (m CMI)

2.1.1 Microcomputer Aided Instruction.

In mCAI microcomputer is viewed as a teaching machine which substitute the teacher. It actually involved teaching the students. The computer presents the new material and assure that student understand it by means of an interactive dialogue or by formal testing. It is defined as teaching process directly inv@lving computer in the presentation of instructional material in an interactive mode to provide and control individualized learning environment for each individual student. These interactive modes are usually subdivided into drill and tutorial, simulation, gaming dialogue, discovery learning and problem solving . Ability to perform complex and repetitive calculations rapidly and accurately, to store volumes of data for subsequent use, to draw and print graphs are some charactaristics instructional purposes in mCAI.

2.1.2 Microcomputer Managed Instruction

In mCMI a microcomputer is used to manage the instructional process, to maintain records on students performance, to control the availability and to time the instructional events of the learning system. It also provide progress reports to instructors and students and help instructors to organize a certain curriculum.

In any traditional course of instruction, much of the valuable time is spent in clerical and administrative activity such as planning, sequence ordering, presentation organizing and grade assignments. Under many circumstances such activities may be delegated to a microcomputer.

2.1.3 MLS Variation

The learning system that we will be discussing will show the characteristics of a Microcomputer Aided Instruction.

MLS may be grouped into four main modes based on the [1,20] facilities required and the possibilities offered. These are,

i. "Pre packed CAI/CMI" to enable the development for factual knowledge and reproductive type skill.

ii. Auto-elaborative MLS to enable learners to build and experiment.

iii. "Personally guided small group MLS" to aid group learning through meeting and discussions.

iv. Gated comperative elaborated MLS, in which the students themselves produce instructional packages to be used by other students thereby adding to the capability of interpersonal experiential learning.

2.2 HUMAN LEARNING

Education is concerned with the transmission of information from one species (an information source or doner) via a reliable communication channel to another species (an information receiver or accepter). The basic unit of education may be conceived as a relatively uninterrupted set of activities involving interplay between a student and an instructional environment.

Conventional teaching is the teacher centred instructional communications which characterized schools and universities almost without challenge until recent years. The typical lecture-and-test approach is essentially an open system with delayed feedback. These feed back may arrive too late to alter instruction for students currently in the system. The process is based on the establishment of a relation of pedagogical communication between a student and an educator and there exist a wide range of expectations, motivations and capacity among the students. The required variety of the instructional system should be large enough to meet these expectations, motivations and capability. With single instructional sequence arranged and presented by one instructor and synchoronized for all student, it is seldom possible to provide enough space for instructional variety.

2.2.1 Individualized Learning System

In an individualized learning system it is possible [4,11] to create and maintain a variety space by development and reorganization of standard and carefully selecting teaching/learning modules considering students ability to study and to generate capability independently. Such de-synchronized instruction, in various formats, has been implemented in many settings and under several names.

2.3 LEARNING SYSTEM

While specifying programming language for Microcomputer based Learning System (MLS), we tried to find how do learners learn when we want them to learn.

2.3.1 Human Learning Process

Although human learning is a continous process, from the standpoint of learner, it may be viewed as a goal oriented activity. From the view point of information processing, learning may arise when a rela[illegible] degree of entropy exists or an priori is open to inform[illegible]. These includes problem solving and decision making a[illegible]ess of learning.

i. Problem solving is concerned with finding the necessary theorems and carrying out actions to achive some task which one did not know when one was faced with the problem initially.

ii. Decision - making involves judgement and thinking in the act of deciding one action against others in the light of some expected value.

2.3.2 Characteristics of Learner

In a task ariented learning system,

a. The learner has to be motivated towards a goal and be adaptive to change.

b. The learner must have certain prerequisites or minimum entry requirement, i.e language or knowledge to enter the learning phase.

2.4 CURRICULUM and INSTRUCTION

Curriculum is defined as intended learning outcomes, an intended capability of a student. Instruction is the set of communications and control procedure designed to regulate the students activity in such a way that he has a high probability of acquiring the intended capability of the curriculum (or a small subset of curriculum).

For each module it is necessary to design instructional activities and message to help the student acquire the intended capability and often, to design assessment procedures to monitor and evaluate the students' progress. In addition control strategies must be chosen, for example a student must show required ability before he can choose another module or sequence.

2.5 CONCLUSION

The organization of automated instruction system is still a typically an adhoc , inventitive operation, ill-understood at a systemic level,despite of sufficient publications concerned with designing instructional communications and materials for such modules.It must be clearly mentioned that the measure of validation of a given instructional module may remain problematical. We need educational engineering of a new degree of sophistication to ensure success in this direction. This can only underscope the need for a highly theorical understanding of the system involved.

CHAPTER - III

FORMAL VIEW OF A LEARNING SYSTEM & ITS SPECIFICATION

3.1 INTRODUCTION

For the whole system host hardware will be a microcomputer with necessary peripherals as described in chapter - I. Normally the system will be a time sharing type where whole resource will be shared by a group of students.

The overall functional concept of the system may be summerized as follows. The course instructor will develop a curriculum, and program it in learning system language, whose formal specification and design methodology is described in chapter - IV. Computer will transmit the information to the students through a media prescribed by the course instructor, evaluate the student's reaction and prepare for the next state depending on the evaluation result at the end of the learning session, detail reporting will be performed by the computer which may be used for next learning session design and group selections.

3.2 COURSEWARE DEVELOPMENT

Courseware development includes the curriculum (course material) along with the instructional set of each procedure for programming the learning system. Total system consists of the following components. [14,15]

i. Type of each state in a presentation
   a. Test state : Those will be termed as the test state during which the computer will wait for feed back from the student or any other input media.
   b. Learning state : Those state for which computer will not wait for any feed back from the students or any other input media.

ii. Duration and control of one state may be any of the following
   a. One test state or learning state will continue till all students have acknowledged but to a maximum limit.

b. One test state or a learning state will continue for a specified duration determined by the curriculam designer.

c. One test state will continue till signalled from certain input channel.

iii. Media of a State

This defines the hardware (processor or input/output) device for a particular setup.

iv. Feed-back information and grade points.

For each test state the possible combinations of feedback and grade point for the specific feedback is supplied for evaluation.

v. Feedback media

Feedback is normally from student key board, but it may be also from variety of input devices.

vi. State sequence

The next state of a presentation is defined through a state space containing the evaluation strategy. The flow and relation between the participants of the system can be blocked as in fig 3.1.

3.3 FORMAL VIEW OF THE SYSTEM

As has been stated earlier, the system is viewed as an interaction between the objects and abstract machine defined on these objects. These abstract machines will be defined interms of abstract machine of preceding level. An abstract program is a program (written in abstract programming language) that can be run in abstract machine. We specify the behaviour of these abstract machine formally in terms of specification language which defines the interconnection between different level within a system.

In the object model, emphasis is placed on crisply characterizing the components of the physical or abstract

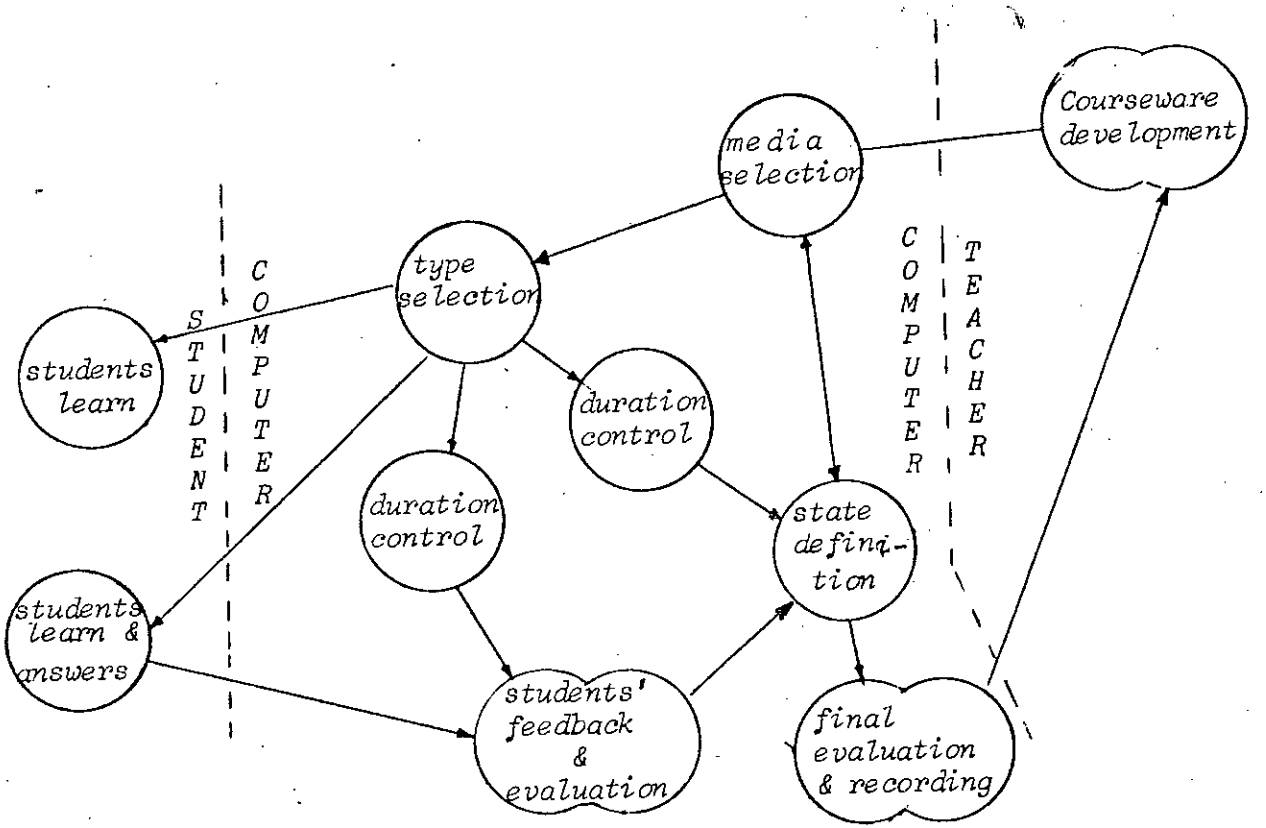


Figure 3.1 Relation between different participants and states of a microcomputer-based learning system.

system to be modeled by a programmed system. The component that are thought of as being " passive " , are called objects. Objects have a certain "integrity " which should not, in fact, can not be violated. An object can only change state, behave, be manipulated, or stand in relation to other objects in ways appropriate to that object . In other words there exist invariant properties that characterize an object and its behavior.

Since many objects essentially have the same behavioral characteristics, it is convenient to define a single set of operations, perhaps parameterized, that are equally applicable to many objects. Two objects are said to be of the same type if they share the same set of operations. In a programmed implementation of a type, the programmed operations are collected together in what is called " type module ". Within a type module definition appears a description of the representation, if any, that is created when an object is instantiated, as well as the procedure that implement the operation of the type.

3.3.1 Assertion Language

Assertion language composed of first-order predicate[18] calculus including primitive functions for sets and tuples are used for defining formal specification of the system.

An assertion is a statement. A proposition is an assertion which is either true or false but not both. Assertions are formed using variables in a "template " which expresses a property of an object or a relation between objects. These templates are called predicates. Assertion made with predicate and variables become true or false when the variables are replaced by specific values. In the expression $P(x_1, x_2, x_3, \ldots, x_n)$, P is a predicate. Variable or constant each xi is an individual variable, and P is said to have n arguments or n - place predicate. If P is an n - place predicate constant and values $c_1, c_2, c_3, \ldots, c_n$ are assigned to each of the individual variables,

the result is a proposition. In order to change predicate into a proposition, each individual variable of predicate must be bound, this may be done in two ways. The first way to bind an individual variable is by assigning a value to it. The second method of binding individual variable is by quantification of the variable. The most common forms of quantifications are universal and existential.

If P (x) is a predicate with the individual variable x as an argument, then the assertion " For all values of x, the assertion P(x) is true " - is a statement in which the variable x is said to be universally quantified. The above statement may be written as

$$\forall x \; P(x)$$

Where the symbol $\forall$ may be read for all , for every , for any , for arbitary or for each . If the assertion P(x) is true for every possible value of x, then $\forall x \; P(x)$ is true ; otherwise $\forall x \; P(x)$ is false. Thus, if the universe of discourse is U, the assertion $\forall x \; P(x)$ is true if and only if the predicate P is valid in U. It follows that for any predicate P and any element c of the universe of discourse, the implication

$$\forall x \; P(x) \Rightarrow P(c) \text{ is true}$$

Another common form of quatification is existential. The individual variable x in the assertion " There exist at least a value for which the assertion P(x) is true" is said to be extentially quantified. Where the symbol $\exists$ may be read "for some " or for at least one ". If the assertion P(x) is true for atleast one element in the universe of discourse, then the proposition $\exists x P(x)$ is true, otherwise, it is false. P(x) is true if and only if P(x) is satisfiable in the universe of discourse. It follows that for any element c of the universe, the implication

$$P(c) \Rightarrow \exists x \; P(x) \text{ is true.}$$

A third form of quantifier is used to assert there exist one and only one element of the universe of discourse which makes a predicate true. This quantifier is denoted by $\exists!$.

While defining specification for different modules of LSL to be discussed in chapter - IV, a variable is declared by two ways, being formal parameters of a function or following any of the symbols $\forall, \exists, \in$ or LET.

The statement "LET x = exp " simply defines a new variable x and associate it with the expression exp. The universally quantified expression defined earlier may have following forms ; $\forall x(R(x))$ or $\forall x(P(x))[Q(x)]$ When the first form in the EFFECTS part, the meaning is same as $\forall xP(x)$ in the definition of $\forall$. That is R(x) is true for every x. When the same form appears in an exception macro, the meaning is : Is it the case that R(x) is true for all x ? the second form is equivalent to $\forall x(P(x) \supset Q(x))$ where P(x) is defined for all values of x and Q (x) need only be defined for values of x for which P (x) is true.

If an expression contains a subexpression " EFFECTS - OF (Z) " where Z is a O - function of the module in question or another module. Here the EFFECTS part of Z is placed in the expression where this subexpression is evoked. It is important that while subexpressions are evoked in this fashion, it must be guaranted that none of the exception conditions of Z are satisfied.

3.3.2 Format for Formal Specification

Normally formal specification consists of following sections.[12,7]

i. DECLARATIONS
ii. PARAMETERS
iii. DEFINITIONS
iv. EXTERNAL FUNCTIONS
v. EXCEPTIONS
vi. FUNCTIONS

i. DECLARATIONS

It gives the type description for all variables that appear in subsequent specifications. All types referenced in the DECLARATIONS are primitive to the programming language that implements the whole system. If there is a type mismatch

during function call in argument, a type mismatch error will be flagged by the system.

ii. PARAMETER

It identifies certain specification variable that will be constant. For instant, PARAMETERS will reflect the maximum real storage in a system, number of floppys available and maximum number of students the system may have to handle. The values of parameters will depend on the implementation of a particular environment, and will remain constant unless the unvironment is changed.

iii. DEFINITIONS

These contains certain macros that are global throughout the system or module of a system, as such reduces the length of specification functions.

iv. EXTERNAL FUNCTIONS

These are external references in the specification of the module of the system . These external references may be resolved in another module of the same system, or it may be a service function of the host computer.

v. EXCEPTIONS

These contains the definition of exceptional conditions within the system or a module.

vi. FUNCTIONS

This is the major part of the system specification.[10] The specification of the functions define the operation done by the function when it is called.

The states and transformations are available to its environment by means of functions that can be called by abstract programs. There are three types of functions available throughout the system

a. V - functions
b. O - functions
c. OV- functions

V - functions

The state information is available via the outputs of V - functions (value returning). Each V-function specification defines its initial value and exception condition space under which it may not be successfully called.

O - functions

O - functions are the members of the set of transformation which causes the system to change its state when it is called. Each O - function specification defines the transformation function and an exceptions condition space similar to V - functions. If an exception condition is satisfied no effects are executed in case of O - function, and control is returned to the calling program with exception condition flag on.

OV - functions

When a function specification contains the property of both V and O - function it is called OV - function. Thus a successful call of OV - function changes the state like a O-function and returns a value like a V-function.

V - functions may be divided into four groups according to their appearance in the system or in a module of the system. They are,

a. VISIBLE
b. HIDDEN
c. DERIVED
d. non-DERIVED

VISIBLE V - functions

Visible V - functions are used as a interface call between the modules of a system or the host computer. It is always visible from other modules, thus may be used for resolving external references.

HIDDEN V - function

Hidden V - functions can not be used as an external reference . It can only be used within certain module of a system, where it is initially defined.

DERIVED V-function

Derived V - functions have values that are defined interms of the non - DERIVED V-functions of the module. Although derived V - functions are redundant, they provide a useful abstraction with in a module of the system.

non-DERIVED V-function

These are the default class of V - functions.

Function Component

A function is normally composed of four components, they are

a. PURPOSE
b. INITIALLY
c. DERIVATION
d. EXCEPTIONS
e. EFFECTS

A function may have subsets of these parts, depending on type and place of its use.

PURPOSE

This section is for documentation. Normally this is a comment explaining the purpose and operation of the function. This section is always present in a function.

INITIALLY

It defines the initial value of a function in terms of its formal arguments and the module parameters. Present only incase of non-DERIVED V-functions.

DERIVATION

This designation occurs only in the specification of DERIVED V-functions. It formally defines the expressions for DERIVED V-functions in terms of the module of the system or service function (V - type) of the host computer, the formal arguments and module parameters.

EXCEPTIONS

This defines the possible exception condition in terms of macro calls where these macros are defined formally in the EXCEPTION section of the module of the system.

EFFECTS

This defines the state change that is produced by the call of O - or OV - functions. Not present in the V - function specifications. The EFFECTS are defined in terms of V - functions, the formal arguments and the module parameters. The assertions of the EFFECTS section defines the relation of V - function before a call to V-function after the call is successful. In the EFFECTS section, V - function values before the call are presented in single quate and the V - function values after the call are unquoted.

3.4 FORMAL DESCRIPTION OR THE PHYSICAL SYSTEM

We formally specify the physical states and their attribute in terms of interaction of some defined sets and tuples.[14]

i. Let $\mathcal{L}$ be an indexed collection of tuples containing information of a presentation designed by the courseware developer.

We can represent $\mathcal{L}$ as

$$\mathcal{L} : \{I, C, M, F, D\}$$

where

$I = \langle i \rangle$	I is a boolean tuple of element t. Identifies System mode (learning,test).
$C = \langle c \rangle$	C is an integer tuple such that $\emptyset \leq e \leq 2$ and total element is t. Contains System control information.
$M = \langle m \rangle$	M is an integer tuple such that $\emptyset \leq m \leq n$, where n is the distinct numbers of presentation media . Maximum number of element in the tuple is t.
$F = \langle f \rangle$	F is a integer tuple such that $\emptyset \leq f \leq o$, where o is the distinct number of feedback media. Maximum number of element in tuple t.
$D = \langle d \rangle$	D is a integer tuple such that $\emptyset \leq d \leq v$, where in any state of the system during sequencing Maximum number of element is t.

In all the specification t in the subscripts denotes total number of states in a presentation.

ii. Let T be a set of tuple such that each tuple of the set represent the ordered set of correct answer of a state of presentation

$$T : \quad \{ \langle t_1 , t_2 , t_3 , , , t_n \rangle \mid t_i \in T_{ii} \}$$

Where n is the mumber of correct answers in one state of the presentation. T_{ii} represents a certain state in a presentation.

T only contains the correct answers on possible correct response from the student. A response which does not satisfies the condition

$$\exists n \; (t_n = r_i \mid r_i \in R_i)$$

is an incorrect answer. $\mathbb{R}$ is a set of t - tuples where each tuple represents the response of the students in any particular state of the presentation thus

$$\mathbb{R} = \{ \langle r_1 , r_2 ,,, r_q \rangle \mid r_i \in R_i \}$$

Where q is the number of students and Ri corresponds to ith state of presentation.

We define for some t the number of state, and n the number of correct answers in any state t, there is a bijection

$$f : T \rightarrow G$$

Where every element of T is mapped to every element of G which is a set representing grade point for each correct answer in T.

iii. We define a state space $\mathcal{S}$ which defines the transition to next state. We define $\mathcal{P}$ is a set of tuples such that

$$\mathcal{S} : \{ \mathcal{P}, \mathcal{A} \}$$

$$\mathcal{P} = \{ \langle P_1 , P_2, , , P_l \rangle \mid Pi \in Pi \}$$

Where $1 \leqslant pi \leqslant t$ and Pi is a set in the ith state contains the transition information.
For ai $\in$ Ai and pi $\in$ Pi, there is a bijection

$$f : A \rightarrow P$$

Where $\mathcal{A}$ is a set of tuples

$$\mathcal{A} = \{ \langle a_1 , a_2 , , , a_l \rangle \mid ai \in Ai \}$$

ai $\in$ Ai defines the limit for a specific state transition and Ai is a set in the ith state which contains ai.

If the value $g_{ni} \in G_i$ of ith state is less than or equal to the $a_{ni} \in A_i$, the next sequential state will be presented, we may write

$$\forall n((\bar{g}_{ni} \leqslant a_{ni} \mid a_{ni} \in A_i \Rightarrow P_t \rightarrow P_{t+1}) \vee (\bar{g}_{ni} > a_{ni} \mid a_{ni} \in A_i \Rightarrow P_t \rightarrow P_{p_{ni}}))$$

Where g_{ni} is the average of the grade obtained by the students. For individualized instruction $\bar{g}_{ni}$ will be replaced by g_{ni} in the above expression.

3.5 CONCLUSION

In this chapter we defined the format of the specification and the physical system upon which we propose to specify the programming language of the learning system.

CHAPTER - IV

FORMAL METHODOLOGY FOR THE DESIGN OF LSL AND ITS SPECIFICATION

4.1 INTRODUCTION

The purpose of a programming language is to render computer useful for transferring an idea or concept from someone's mind to a runnable programe. From this view point, programming is usefully visualized as a process of abstraction followed by making a series of choices of algorithms and representations that deliver a concrete version of this abstraction. Of serious concern in software construction are techniques that permit us to recognize that a certain program is reflecting the initial concept in someone's mind. With formal techniques, a specification is interposed between the concept and the programs. Its purpose is to provide a mathematical description of the concept, and the correctness of a program is established by proving that it is equivalent to the specification.

Parnas has developed a technique and notation for [7,12,13] writing implicit specifications by describing the states of an abstract state machine. In this approach, the state machine is identified with a single, representative object, and the specification describes how the state of the machine changes as a result of the applications of some of the operations . While defining the methodology and specification of the LSL, we were greatly influenced by the works done at Stanford Research Institute, California, which was again greatly influenced by the approach of Parnas in specifying software modules and in arranging them in a hierarchy. The incorporation of abstract types in a programming language appears to have originated in Algol and SIMULA.

The specification methodology of LSL may be viewed as

i. Use of hierarchy of self-contained data abstraction, where each abstraction is described by formal specification.

ii. Specification of the phases of specification and implementation.

iii. Use of assertion to characterize the representation of data abstruction in terms of lower-level abstructions.

While defining the specification for the software system, we view the system into several levels. Each level in the system acts as a manager of all objects of particular type. The system is organized as a hierachically ordered collection of abstract machines. Each machine is formally specified and is implemented by abstract programs executing on lower level machines. The proofs of correctness of the total system are thus reduced to proving the correctness of many smaller abstract programs.

4.2 METHODOLOGY FOR DESIGN and IMPLEMENTATION*

The stepwise refinement concept as introduced by Dijkstra allows a programmer in a sequence of refinement to prove the correctness of a program. Following the work of Parnas, we define a sequence of abstract machines (M_o , M_1 ,,, M_n) , where M_o is the most primitive machine which may be hardware or the implementating programming language. Each abstract machine consists of a state and a set of transformations for effecting state changes.

We may view the whole system as a hierarchy of levels [10,12,16] composed of abstract machines.

i. Single thread linearordening, one abstract machine per level.

ii. Modular organization, each level is composed of an abstract machine which is again made of one or more component abstract machines.

iii. Modularization without redundant module definition.

Some of the materials of this section has been adapted from works of the Computer Science Group at Stanford Reseqrch Institute,Ca 94025 Ref- 16. Where the work was partially supported by the National Science Foundation of U.S.A. under grant DCR74-1866.

i. Single thread linear ordering organization.

In this view each abstract machine occupies entire level and a level is implemented in terms of level immediately below it. The hierarchy may be viewed as a single thread linear ordering rathar than a tree or a directed acylic graph (fig 4.1a).

Each non primitive level i ($i > \emptyset$) of the system adds features to those available at level i - 1. Level i is higher in hierarchy than level i - 1. When a feature orignated at i-1 but also visible at i and above, we say that the feature is transmitted through level ior that level i is transparent to that feature. When a feature is not visible above level i, we say that the feature is hidden by level i or that level i hides the feathure.

ii. Modular Organization.

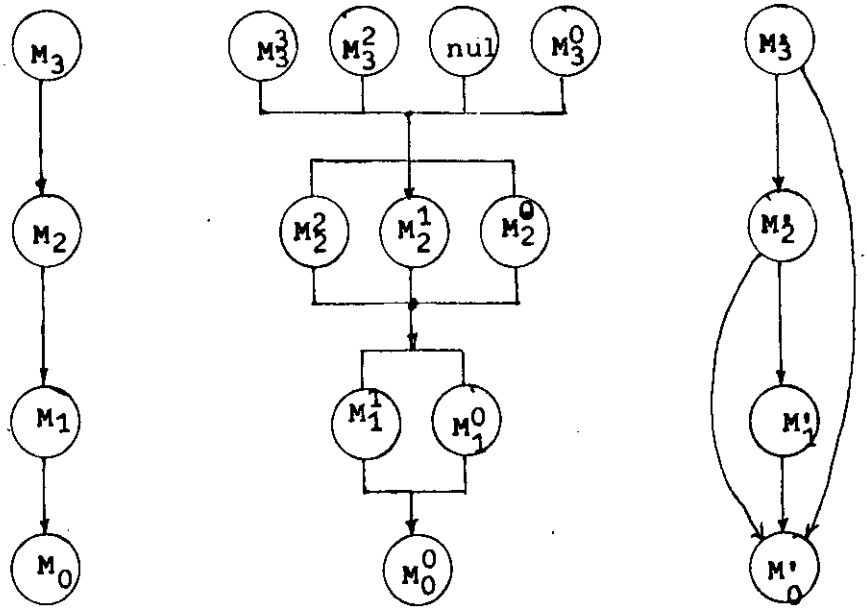
In this view an abstract machine may be divided into one or more component abstract machine called modules. The organization is shown in fig 4.1b and has following properties,

a. At level i ($i > \emptyset$), machine M_i consists of module M_i^i the defined module at level i, and at most i-1 other modules, M_i^j ($o \leq j < i$). Where these modules except M_i^i reflects the features there are transmitted from levels below i. Thus M_i^i represents the set of features that level i adds to the system and M_i^i represents the set of features transmitted to level i from machines at lower levels.

b. If a module is absent from the defination of a level's abstract machine, this absence represents the features that level i hides from the levels above it. Thus if a module M_i^j is absent, the module M_k^j must also be absent, for all $k, k > i$.

iii. Modularization without redundant module defination.

In this view only a levels' defined modules are explicitly described. This provides modularization within a level to avoid redundant module definations and at the same time retain the integrity of the first two views of the hierarchy. Shown in fig 4.1c module M_i is equivalent to M_i^i of the previous view.



a. Single thread linear ordering organization.

b. Modular organization.

i. total level contribution

$\forall i(i \geqslant 0 \Rightarrow M_i = \bigcup_{j=0..i} M_i^j)$

ii. contributions from lower levels

$\forall ij(j > 0 \wedge j < i \Rightarrow M_i^j = M_i^j \vee M_i^j = \text{undefined})$

iii. contributions from levels above

$\forall ijk\ (j > 0 \wedge j < i \wedge k > i \Rightarrow (M_i^j = \text{undefined} \Rightarrow M_k^j = \text{undefined})$

c. Modularization without redundant module defination.

for all levels above zero we have

$\forall i(i \geqslant 0 \Rightarrow M_i' = M_i^i)$

Figure 4.1. Hierarchical organization of abstract machines.

Thus from the figure at level i =3, feature from level ø and level 2 is transparent but not from level 1.

4.2.1 Stepwise Implementation

Whole process from the initial design formulation to the specification and implementation of the system can be divided into five stages. These stages are intended to represent the desired protocol for constructing a hierarchically structured software system. They are,

i. A hierarchical decomposition of the system into levels of functional abstruction.

ii. Formal specifications for the operations and data structures at each level.

iii. Formal assertions about the representation of the data structures at each level in terms of the data structures of lower level.

iv. Abstract implementation of each function of each level in terms of programs calling lower - level functions.

v. Actual implementation in terms of hardware or a programming language.

i. Step 1.

All modules that are required are selected and placed in a hierarchical module ordering. Each O and V - functions for the modules are listed and at which higher level this module is resolved.

ii. Step 2.

Each module is formally specified here. Based on the specifications for a module, it is possible to prove that specifications are self-consistant and certain global assertions are true.

Proof of self-consistancy requires that for any O - function, assertion given for V - functions are not self-contradictory over the domain of definition of these V-functions. Global assertion are written in terms of V-function of the module.

If a certain V-function applies to a module they must be true for intial value and also after any sequence of O - function calls.

iii. Step 3

Here mapping decisions are made for functions defined at i level in terms of i - 1 level. The state of a module is defined by certain assignment of values to V-functions of the module . We define state space S at M_i and T at M_i -1. A mapping function at level i is a surjective function f from T' to S, where T' $\in$ T and every element s $\in$ S is in the image of function

During stage mapping following conditions may arise

a. More than one state in level i-1 may map to a single state in i. This condition is shown in fig 4.2a.

b. Some of the states in level i-1 may not have surjective image in level i . However, a direct transition at level i may be a result of direct transition between states and confined within a level, an alternate implementation algoritm will enforce an alternate path of transition (fig 4.2b).

c. We have a third type of transition between states of a level which is a mixture of both subjective and non-surjective transition between level i and i-1. A single non - surjective transition at i may be due to several non-surjective transition at i - 1 fig 4.2c shows that a non-subjective state transition (S_1 , S_2) in i could be due to any of the following transitions in i-1 : (T_1, T_3), (T, T_4), (T_2, T_3) or (T_2, T_4).

Mapping function expressions are defined such that V-function values at level i of abstract machine M_i maps down to an expression containing V-function values of abstract machine M_{i-1} at level i-1. Although formally function is defined as an upward mapping of states. After defining mapping functions for each of the V-functions of M_i, it is important to see if the specifications are successful, i.e. if they successfully characterize the properties of a mapping function as specified in the state-space description of abstract machine.

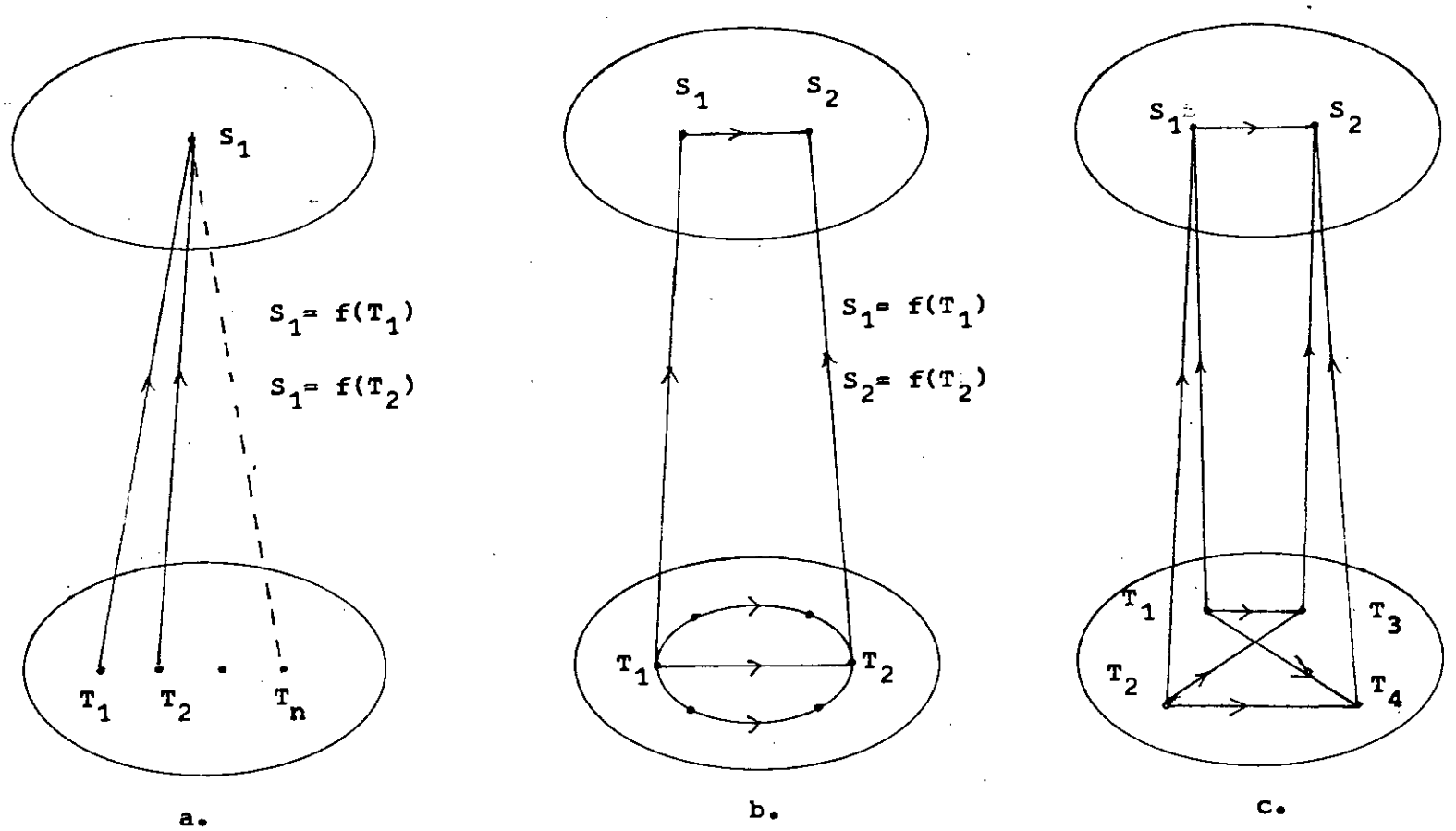


Figure 4.2. Mapping function f relating the stages of two abstract machines.

Mapping function expression is said to be consistent if the specifications are successful. If mapping function expression is inconsistent, it is impossible to find on implementation satisfying the specification of M_i and M_{i-1}. The specification of M_i and M_{i-1} and mapping functions of level i are sufficient to generate the correctness criteria for an implementation of M_i in terms of M_{i-1}.

iv. Step 4

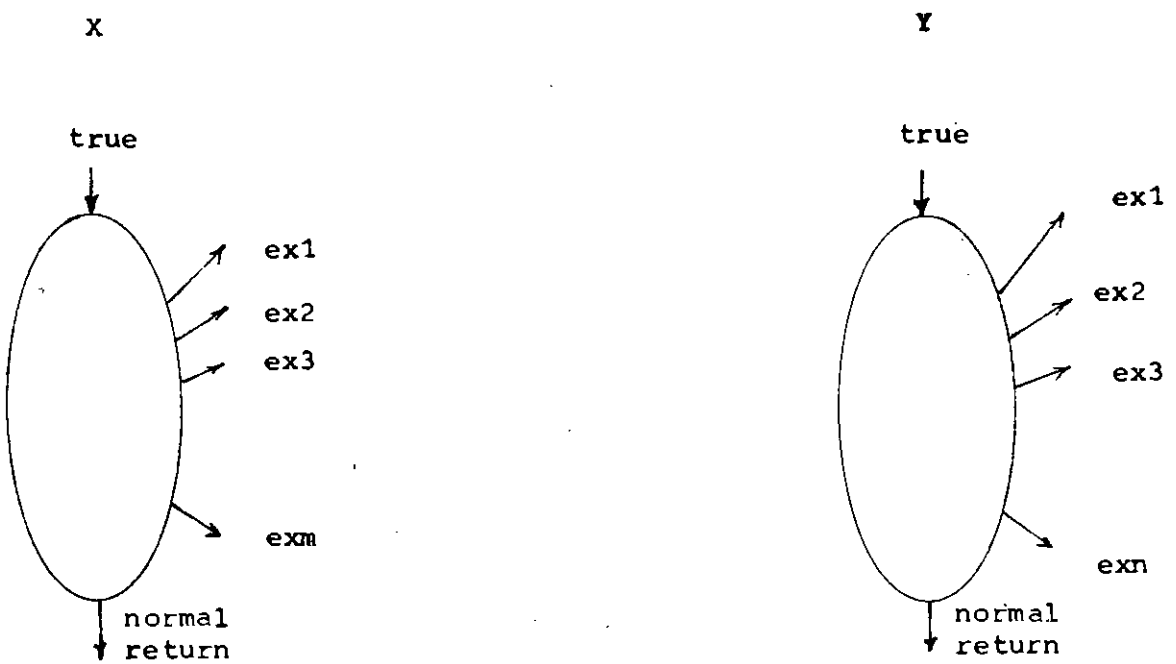
Each of the function at certain arbitray level in the hierarchy $M_i (i > \emptyset)$ is implemented as an abstract program using the function of M_{i-1} and the control space of some formally defined existing programming language (programming language for implementation of the system). These abstract programs complete the binding of the decisions that were left incomplete by the mapping functions described in step 3. The abstract program must proven to be " successful" implementation with respect to the specifications of M_i and mapping function between M_i and M_{i-1}. To carry out this test, input and output assertins are derived for the implementing programs, which if satisfied, imply a successful implementation of the abstract program.

Implementing programs has an entry point (normally with an input assertion of TRUE) and several exits. If the function which is implemented by the implementation program has n-exception conditions, then there should be n+1 exits, one for each of the n exception conditions and one for normal exit. In certain (fig 4.3) V function X the output assertions for the normal exit states that the program returns a value equal to the installed mapping functions expressed for X. In case of O - functions Y the output assertion for the normal exit consists of the effects of the mapped specification for Y.

v. Step 5.

Primitive function such as M_o and abstract programming language is implemented interms of actual codes available in well diffined programming languge . The communication mechanism between levels are also defined here. It may be,

a. Macro expansion at compile or assembly time



a. Expression for X(V-function)

b. Expression for Y(O-function)

exi = Mapped exception condition

Figure 4.3. Mapping function implementation for O and V-functions showing input and output assertions.

b. Procedure calls

c. Interprocess communication

The formal semantices of the communication mechanism must be stated and proved correct. From user point of view the end product of the total system is a software system with a behavior of the highest-level abstract machine M_n . Since the behavior and existence of lower levels are completely hidden by M_n, the users are only concerned with the top level M_n .

4.3 DESIGN APPROACH OF LSL

Outline of the hierarchical decomposition of the total system will be discussed in this section. Our goal for the system are as follows

i. While executing a certain courseware, the system should permit dynamic sharing of resources that is well-supported, efficient, flexible, and easy to use.

ii. The system should provide primitive functions such that user defined devices and user defined procedures may be easily installed.

iii. The system should have desired level of security, such that unauthorized access to the important classified information easily detected and reported.

iv. Processing should be efficient flexible and reliable with facilities for backup and recovery from hardware failures.

The system should be able to perform following functions,

i. Creating abstract resources (objects), and establishing the kinds of operation that can be performed on these objects.

ii. Granting to user(i,e, courseware developer,students), according to various user class, access rights for resources.

iii. Calling on a program that will service the resource for the caller which will check access right of caller before servicing any particular request.

4.4 TYPE MANAGERS

When designing LSL it is convenient to view each level as the manager of a particular type of abstract object.

For generalized view of the operations on set of similar abstract resources as a service to a set of users, or subjects, we may think the part of system responsible for this as a type manager. It has two important properties.

i. It may be requested to perform certain operations on objects on request by the subjects or users. An operation may change the state of one or more objects (O - functions) or may extract some information derived from the state of the object.

ii. It provides necessary protection for the objects with in its domain. It may share the protection mechanism offered by the host operating system or it may generate its own protection system. For given service request it must be able to decide whether a subject is allowed to perform a desired operation on a set of objects.

Type managers are actually a set of modules whose functions are implemented as a set of programs, so that each service request corresponds to a call. The program that implements call upon type managers at lower level of the system, each type manager using lower-level objects to represent the object it maintains.

The most important type manager in the system is the courseware type manager. A courseware contains the learning modules that will be presented to the students and a control strategy how these learning modules will be presented. Here the object of concern is the set of learning module in the courseware, the subjects are programs desiring to perform certain operation on a particular learning module of the courseware.

Following is a list of some of the possible operations that a user may wish to perform on a particular courseware.

i. Create learning modules for courseware.
ii. Edit a certain learning module
iii. Merge two learning module with a courseware
iv. Edit certain control modules (which decides control strateges).

v. Change some attribute of the courseware such as physical device type, number of students etc.

vi. Make hardcopy of the text of the learning module for off line editing by the courseware developer.

vii. Delete learning modules from courseware.

In order to perform any of these operation (and similar other operations) on a certain module of a courseware, a subject calls one of the set of programs that constitute the type manager for courseware. Before this type manager operates on courseware modules (objects) for intended operation, it makes sure that it is operating on the right object with adequate access right.

4.4.1 Type managers for courseware

In this section specifications of type manager for certain operation subset (user may define more operation and specify in the same manner) will be discussed.

A subset containing only four component will be discussed in detail from the set of managers, these are

i. Manager for capability
ii. Manager for Segment (memory management)
iii. Manager for extended type objects
iv. Manager for directories

Objects associated with each manager will be defined in terms of the operation provided by the managers. Features of the managers along with their formal specifications will be presented.

Some of the materials of section 4.4 has been adapted from the works of the Computer Science Group at Stanford Research Institute,Ca 94025, Ref-16. Where the work was partially supported by the National Science Foundation of U.S.A. under Grant DCR74-1866.

i. Capabilities

The set of objects that are accessible during the execution of a program is called the domain. Formally a function f from A to B, denoted by $f : A \rightarrow B$, is a relation from A to B such that for every $a \in A$, there exists $b \in B$ such that $\langle a,b \rangle \in f$, we say f is a function from A to B, where A is called the domain and B is called codomain of f .

A domain can be expressed as a set of descriptors, some times called capabilites. A capability is itself an object that, by virtue of the limited operations that can be performed on it, can be used by type managers to protect their own objects. The descripter or capability is an unforgeable token that identifies a particular object. Using of this token code is restricted to use of only those objects in its domain. From the view point of user, a capability consists of two parts, a unique identifier (uid) and an access vector, where uid is an integer and access vector is a boolean n-tuple.

Type managers associate a distinct uid with each object of the courseware under their control. If m operations are supported by a certain manager where $m \leq n$, then each operation will correspond to a unique entry in the access vector such that a true in a position indicates that corresponding operations may be performed. If a subject possesses a capability with a uid corresponding to object x with a true in the position corresponding to operation y, then it is said that the subject has y - access for x .

In the table T-4.1 stating specifications for capability modules we see

Visible V - functions
get_uid(c),get_access(c),create_capability,
restrict_access(c, b)

Hidden V - function
cap_uid(u), which keeps track all uids that have ever been associated with capabilities.

ii. Segments

Normally segment supports will be available from the host operating system by standard service call. However this support may have to be generated for certain host operating system, where host operating system fails to provide such facility.

The formal specification for this module is given in table T-4.2 . In the specification, a segment consists of a seguence of words and each word in the segment is referenced by an integer offset. The number of words in a segment is the size of the segment (seg_size). The upper limit on the size of any segment is "maxsize" which will depend upon the page size of the virtual memory of any particular system. There is a maximum number of segments that can be maintained and accommodated, which again depend on the size of virtual memory and size of the segment .

Since the segment provides the virtual memory interface, it is primary storage entity for procedure code, data, and capabilities. The segment is the primitive object type from which more complex and higher-level types are synthesized. The service provided by the segment manager as follows

a. Create a new segment

(create_seg(j);visible). Create a new segment of some initial size j. The segment manager will possess a capability s , by calling the capability manager, and will pass the user a capability which has the same uid as s but with true in access vector position corresponding to "read", "write" and "delete"

b. Return the value of a segment word at some offset i (read(s,i), visible). This operation requires the passing of a capability s with "read" access set to true for the segment in question. i must be a valid integer offset.

c. Change the size of existing segment to j (change_seg_size(s,j);vis) requires capability with write access set to true and a valid j.

d. Delete Segment (delete_seg(s))) requires capability with "delete" set to true for the segment in question.

The hidden V-functions h_seg_exists(u) h_seg_size(u), h_seg_size(u), h_read(u,i) takes uid as its argument where incase of similar VISIBLE functions uid is replaced by the capability. The use of uid's is more consistant in identifying segments than capabilities.

The segment is a hardware and host operating system primitive type, and the system is believed to have hardware and software host support for the rapid access to the pages in main memory on the basis of segment uid and offset.

iii. Extended - type object

For user defined device and procedure we need a type manager which will operate on user defined objects. The extended-type manager supports creatin and maintenance of user defined abstract type and object.

Extented-type manager may be viewed as a type manager which provides abstract operation. Representation of the abstract operation, which defines the concrete operation to be done on the representation is itself an object, which is called representation object. As such the type manager of extended-type object will call type manager of representation object to perform the concrete operation. Thus with the help of representation object extended-type manager can perform many advanced and difficult operations easily . Table T-4.3 gives formal specification of extended type manager.

d. Directories

Directories may be viewed as ordered representation of capabilities. For each capability these may be a name associated with the capability, as such above directory level, the objects may be referred by the name, thus each entry in the directory is a two-tuple (name, capability)

The operations that may be performed by the directory manager

- Create a directory for the courseware system
- Add new entry to directory
- Delete an entry from directory.
- Delete an existing directory.

4.5. CONCLUSION

The formal methedology is discribed in this chapter. We have decomposed the whole system into a hierarchical order of abstract machines. At the top of this hierarchy is the command processor which works as a interface between the end user (students, courseware developer, instructor). In the next chapter command processor will be discussed with specification for relevant syntax.

Table T-4.1

DECLARATIONS

boolean : b

capability : c,c1

unique identifier u

PARAMETERS

access length length of access vector

- DEFINITIONS

EXTERNAL FUNCTIONS

- EXCEPTIONS

BAD_LENGTH(⟨b⟩) : length(⟨b⟩) ≠ access length

FUNCTIONS

HIDEEN V-FUNCTION b = cap_uid(u)

PURPOSE : true for uids of all capabilites previously created

INITIALLY : FALSE

I V-FUNCTION u = get_ uid(c)

PURPOSE : returns the uid of a capability

INITIALLY : UNDEFINED

EXCEPTIONS: NONE

I V-FUNCTION(b) = get_ access(c)

PURPOSE : returns the access vector of a capability

INITIALLY : UNDEFINED

EXCEPTIONS : NONE

I OV-FUNCTION c = create_capability

PURPOSE : create a capability with a brand new uid and an access vector of all TRUE

EXCEPTIONS : NONE

EFFECTS : ∃u('cap_uid(u) = FALSE ;

cap_uid(u) = TRUE;

'get uid'(c) = UNDEFINED;

get_ uid(c) = u;

∀i(1 ≤ i ≤ access length) [get_access(c) [i] = TRUE])

I OV-FUNCTION c1 = restrict_ access(c, ⟨b⟩)

Table T-4.1 contd.

I OV-FUNCTION c1 = restrict_access(c, ⟨b⟩)

PURPOSE : create a new capability with the same uid as the one passed, but with an access vector with a subset of TRUE values from the one passed

EXCEPTIONS : BAD_LENGTH (⟨b⟩)

EFFECTS : get_uid(c1) = 'get_uid'(c)
get_access(c1)=⟨b⟩ ∧ 'get_access'(c)

Table T-4.2

DECLARATIONS

boolean	:	b
integer	:	i,j
capability	:	s(segment)
unique identifier	:	u
machine work	:	w

PARAMETERS

maxsize	:	maximum segment size
maxsege	:	maximum number of segments

DEFINITIONS

"red" ∈ abilities(s)⇔get_ access(s) [1] = TRUE

"write"∈abilities(s)⇔get_access (s) [2] = TRUE

"delete"∈abilities(s)⇔get_access(s) [3] = TRUE

nsegs = cardinality({u| h_ seg_ exists(u) = TRUE})

EXTERNAL FUNCTIONS

CAPABILITY MODULE : create_capability,get_acces,
get_uid

EXCEPTIONS

NO_SEG(u) : h_seg_exists(u) =FALSE

ADDRESS_ BOUNDS(u,i): i < o ∨ i > (hseg_size(u) - 1)

BAD_SIZE(i) i < o ∨ i > maxsize

TOO_MANY :nsegs ≥ maxsegs

NO_ABILITY(s,"a"): "a" ablities (s)

FUNCTIONS

HIDDEN V-FUNCTION b=h_seg_exists(u)

PURPOSE : true for uid's of currently existing segments

INITIALLY : FALSE

I DERIVED V-FUNCTION b =seg_exists(s)

PURPOSE : external form of h seg_exists

LET u = get_uid(s)

DERIVATION : hseg_exists(u)

EXCEPTIONS : NONE

HIDDEN V-FUNCTION j=h_seg_size(u)

PURPOSE : returns size of segment u

INTIALLY : UNDEFINED

I DERIVED V-FUNCTION j = seg_size(s)

PURPOSE : ex[illegible]l form of h_seg_size(u)

LET u = get ui[illegible]

DERIVATION : h[illegible]ize(u)

EX[illegible]S : N[illegible](u)

Table T-4.2 contd.

```
HIDDEN V-FUNCTION w = h_read(u,i)
PURPOSE :                 returns theith  word of segment u
INITIALLY :               UNDEFINED

I DERIVED V-FUNCTION w = read(s,i)
PURPOSE  :                external form of h_read(u,i)
LET u    =                get_uid(s)
DERIVATION :              h_read(u,i)
EXCEPTIONS :              NO_SEG(u)
                          ADDRESS_ BOUNDS(u,i);
                          NO_ABILITY( s, "read")

I OV-FUNCTION S    =      create_seg(j)
PURPOSE      :            creates a new segment of size j
EXCEPTIONS   :            BAD_SIZE(j);
                          TOO_MANY
EFFECTS  :  s=            EFFECTS_OF (create_capability);
            LET u =       get _uid(s)
            h_seg_exists(u) = j TRUE
            h_seg_size (u) = j;
           ∀i (i ≥ 0 ∧ i ≤ j - 1)[h_read (u,i) = 0]

I O-FUNCTION delete_seg(s)
PURPOSE : deletes segment s
LET u = get_uid(s)
EXCEPTIONS : NO_SEG(u)
             NO_ABILITY(s "delete")
EFFECTS : h_seg_exists(u) =          ; h_seg_size (u) = UNDIFINED
         ∀i (i ≥ i < h_seg_size(u))[h_read(u,i) = UNDEFINED]

I O-FUNCTION change_seg_size(s,j)
PURPOSE : changes size of segment s to j
LET u = get_uid(s)
EXCEPTIONS : NO_SEG(u);
             BAD_SIZE(j);
             NO_ABILITY(s,"write")
EFFECTS : h_seg_size(u) =j;
             ∀i(i ≥ h_seg_size(u) ∧ i < j)[h_read(u,i) = 0]
             ∀i(i ≥ j ∧ i < h_seg size(u)) [h_read(u,i)=UNDEFINED]

I O-FUNCTION write(s,i,w)
PURPOSE :  write machine word w into ith location of segment s
LET u = get_uid(s)
EXCEPTIONS  : NO_SEG(u);
              ADDRESS BOUNDS(u,i);
              NO_ABILITY(a,"write")
EFFECTS  :  h_read(u.i) =w
```

Table T-4.3

DECLARATIONS

```
boolean                 :   b
capability              :   c,c1,c2,c1i(arbitrary);
                            t,t1,t2(of type 'TYPE');
                            s(of type 'SEGMENT')

unique identifier       :   u,u1,u2,u1i(arbitray);
                            ut,ut1,ut2 (of type'TYPE')

integer                     i,j
```

PARAMETERS

```
utt                     :   unique_id for type 'TYPE'

us                      :   unique_id for 'SEGMENT'

maxobj                  :   maximum number of objects
max_impl_obj            :   maximum number of implementationobjects
```

DEFINITIONS

```
"create object" ∈abilities(t)⇔get_access(t) [1] = TRUE

"manage"∈abilities(t) ⇔ get_access (1)[2]= TRUE
nobjs  = cardinality({u| h_object_exists(u) = TRUE})
```

EXTERNAL FUNCTIONS

```
CAPABILITY MODULE       :   create_capability,get_access,
                            get-uid

SEGMENT MODULE          :   create_segment,
                            h_seg_exists,
                            nsegs,
                            maxsegs
                            EXCEPTIONS

INVALID_OBJECT u)       :   h_object_exists(u) = FALSE
UNINITIALIZED(u)        :   h_initialized(u) = FALSE
INITIALIZED(u)          :   h_initialized(u) TRUE ∨get_type(u) =utt
INVALID  TYPE(u)        :   h_get_type(u) ≠ utt ∨ u = us
NO_ABILITY(c,"a")       :   "a"∉ abilities(c)
INVALID_TYPE CORRESPONDENCE(u,ut) : h_get_type(u) ≠ ut
IMPL_OBJS_ PRESENT(u)   :   IF h_initialized(u) = TRUE ∧h_get_type (u) =
                            utt THEN

                              ∃c(c ∈h_impl_cap(u)∧
                              ∃u1 (u1 = get_uid(c)∧
                                 (h_object_exists(u1) ∨
                                  h_seg_exists(u1))))

                            ELSE FALSE

INVALID_OBJECT_TUPLE(⟨u⟩)   : ∃u1 (u1∈⟨u⟩ ∧
                                h_object_exists(u1) = FALSE)

INVALID_TYPE_TUPLE(u)       :  ∃u1(u1∈⟨u⟩ h_get_type(u1) ≠ utt )

NO_ABILITY_TUPLE(⟨c⟩"a")    :  ∃c1(c1∈⟨c⟩∧"a"∉ abilities (c1)
TOO_MANY_OBJECT(⟨ut⟩)       :   nobjs + cardinality ({ j | 1  ≤   length(⟨ut⟩)∧
                              ⟨ut⟩[j]  ≠  us}) > maxsegs

TOO_MANY_IMPL_OBJS(⟨ut⟩)    :    length(⟨ut⟩) > max_impl_obj
```

Table T-4.3 contd part 2.

FUNCTIONS

```
HIDDEN V-FUNCTION b = h_object_exists(u)
PURPOSE                    :  true for all extended-type objects that currently
                              exist
INITIALLY                  :  IF u = utt OR u = us

                              THEN TRUE ELSE FALSE

I  DERIVED V-FUNCTION b= object_exists(c)

   PURPOSE  :  external form of h_object_exists(u)

   LET u = get_uid(c)
   DERIVATION                 :  h_object_exists(u)
   EXCEPTIONS                 :  NONE
   HIDDEN V-FUNCTION u2 = h_get_type(u1)
   PURPOSE  :  returns the uid for the type of extended object u1
   INITIALLY:  IF u1 = utt V u1  = us

                              THEN utt ELSE UNDEFINED

I  DERIVED V-FUNCTION u = get_type(c)
   PURPOSE  : external form of h_get_type(u)
   LET u = get_uid(c)
   DERIVATION : h_get_type(u)
   EXCEPTIONS   :  INVALID_OBJECT(u)
   HIDDEN V-FUNCTION  b= h_initialized(u)
   PURPOSE  : true if extended object u is initialized with representation objects
   INITIALLY  :  FALSE

I DERIVED V-FUNCTION b = initialized(c)
  PURPOSE  : external form of h_initialized(u)
  LET u = get_uid(c)
  DERIVATION : h_initialized(u)
  EXCEPTIONS : INVALID_OBJECT(u)
  HIDDEN V-FUNCTION    = h_impl_cap(u)
  PURPOSE    : returns the tuple of capabilities implementing extended objec u
  INITIALLY  :  UNDEFINED

I HIDDEN  V-FUNCTION c2 = impl_cap (c2,t)
  PURPOSE    : external form of h_impl_cap(u),t is the type manager's capability
  LET u = get_uid(c2)
  LET ut = get_uid(t)
  DERIVATION : h_impl_cap(u)
  EXCEPTIONS : INVALID_OBJECT(u);
               UNINITIALIZED(u);
               INVALID_OBJECT(ut)
               INVALID_TYPE(ut);
               NO_TYPE_CORRESPONDENCE (u,ut);
               NO_ABILITY(t,"manage")

I OV-FUNCTION c = create_object(t)
  PURPOSE  :  to create an extended object of type t, returning capability c;
              it leaves the object uninitialized

  LET ut  = get_uid(t)
```

Table T-4.3 contd. last part.

```
EXCEPTIONS  : INVALID_OBJECT(ut);
              INVALID_TYPE(ut);
              NO_ABILITY(t,"create object")
              TOO_MANY_OBJECTS(tuple(t))

EFFECTS :  c = EFFECTS_OF ( create_capability);
           LET u = get_uid(c);
           h_object_exists(u) = TRUE;
           h_get_type(u) = ut;
           h_initialized(u) = FALSE

I O-FUNCTION  delete_object(c,t)

  PURPOSE   :  to delete an object from the extended type level, its
               representation objects must have been previously deleted;
               t is the type manager's capability.

  LET u  = get_uid(c)
  LET ut = get_uid(t)
  EXCEPTIONS  : INVALID_OBJECT(u);
                INVALID_OBJECT(ut);
                INVALID_TYPE(ut);
                NO_ ABILITY ( ut, " manage")
                IMPL_ OBJS_PRESENT(u)
                NO_TYPE_CORRESPONDENCE (u,ut)
  EFFECTS :    h_object_exists(u) = FALSE ;
               h_initialized(u)   = FALSE;
               h_get_type(u)      = UNDEFINED;
               h-impl_cap(u)      = UNDEFINED

I O-FUNCTION initialize(c,t,⟨t2⟩)

  PURPOSE   : to initialize the representation of extended object c to
              be new instances of the objects whose types are in the
              tuple ⟨t2⟩; t1 is the type manager's capability for c

  LET u = get_uid(c)
  LET ut1 = get_uid(t1);
  LET ut2 = get_uid( ⟨t2⟩)
  EXCEPTIONS   : INVALID_OBJECT(u);
                 INITIALIZED(u);
                 INVALID_OBJECT(ut1);
                 INVALID_TYPE(ut 1);
                 INVALID_TYPE_CORRESPONDENCE(u,ut1);
                 NO_ABILITY(t1, "manage");
                 INVALID_OBJECT_TUPLE(⟨ut2⟩);
                 INVALID_TYPE TUPLE(⟨ut2⟩);
                 NO_ABILITY_TUPLE (⟨t2⟩, "create" );
                 TOO_MANY_OBJS ( ⟨ut2⟩ );
                 TOO_MANY_IMPL_OBJS( ⟨ut2⟩)

EFFECTS     :  h_initialized(u)  = TRUE;
            ∃⟨c1⟩(LET ⟨u1⟩ = get uid (⟨c1⟩);
                  length (⟨c1⟩)= length (⟨t2⟩);
                 ∀i(1≤i <  length (⟨t2⟩))
                    [IF 'get uid'(⟨ut2⟩[i]) = us
                     THEN      i]=EFFECTS_OF(create_seg(0))
                     ELSE      ] = EFFECTS_OF (create_object( ⟨ut2⟩[i] ))]
                    _impl      u) = ⟨c1⟩)
```

CHAPTER V

COMMAND PROCESSOR & USER INTERFACE

5.1 INTRODUCTION

Command processor can be viewd as the union of the function set of all the module of the system . However, an elegent user interface can be made by replacing functions from several modules by a single all-purpose function. The user interface command processor can be considered as a level above the highest system level, but without any new facilities, loss of power, or information hiding . In this chapter the command forms available at the user interface will be described.

5.2 USER INTERFACE

System recognize two types of user

i. Courseware developer

ii. Student

The command set available to the student is a subset of the command set available to the courseware developer.

5.2.1 Student command set

We start by defining student command set which will also be available to the courseware developer by defination. To validate the command request, we have an object existance predicate a type function which may be applied to other modules of the system. Thus a function "object_exist (c)" may be defined which accepts the capability for the request and returns a true if the intended object exist corresponding to the capability.

Example

As a cold start without any instructor for self learning interactive mode, a student may wish to establish through this command set.

LOGON userid password

$\exists! n\ [\ \text{userid}(n) \wedge \text{password}(n) \Rightarrow \text{LOGON}]$

Here after the command •LOGON, userid is taken and tested if by the object exist function if any such object with the given password exist . If such object exist, its capability will be restored from the userid directory, which will possibly be a tuple composed of userid, pass word and capability (which indicates which operation are valid for the user).

The next command from the user may be

• TEACH , Subject module , attribute

This comand indicates that the student wants to learn from computer. Before this command will be executed following condition arechecked

$\forall$ (get_ access(TEACH) $\wedge$ sub mod $\wedge$attribute $\Rightarrow$ RUN TEACH)

If the proposition is true TEACH module with the subject module and attribute will be run.

In the same way one may design a member of commands which he seems appropriate for the particular system in his hand. Given in the appendix a list of command subset available from student logon.

5.2.2 Course developer command set.

Course developer establish communication with the system with the LOGON procedure. After logon a course developer or instructor may develop a courseware, may run a teaching/learning session.

For description of courseware more elegently we adapt a special notation which is a union of guarded command as indroduced by Dijastra and BNF. However instead of standard BNF notation, we have used notation developed by Glanville for describing spatial relation. The basic structure of the form is

$\langle O \rangle = [\ (X \quad] \Rightarrow E$

Where O is the command definition, X is a tuple which defines the operators and variables of operation, P is the set of tuple where this operation will be applied, The tuples represents, how text, graphics and tests are organized in a learning system. The relative type position of the tuples in tuple set P is defined during configuring or imple menting the system.

5.3 SYNTAX CONVENTION

The conventions used to illustrate syntax rule are as follows.

i. Uppercase letters and punctuation marks represent information must be coded exactly as shown

ii. Lowercase letters and terms represent information that must be supplied by the student or courseware developer.

iii. Information contained within bracket][represents an option that can be included or omitted, depending on the requirements of the program. Stacked options contained within brackets, for example

$$\left]\begin{matrix}A\\B\end{matrix}\right[$$

represent alternatives,

one and only one may be chosen.

iv. Options contained within bracket { } represent alternatives, one of which must be chosen.

v. An ellipsis (.....) indicates that a variable number of items may be included.

vi. Underlined elements represent an assumed option in the event a parameter is omitted

vii. Parentheses must be coded as shown.

Command level syntax

System at command level will be promped by displaying a "*" at column 1 this

col 1

*

in the screen indecates that system is in command level.

•C_ DIR

required for requesting services those are partinent to courseware developer

* {L, E, D} where L for create operation of text module, graph module, Test module.

E for Edit operation on text module, graph module, Test module.

D is for Delete operation on Text module, graph module, Test module.

The whole language capability will be demonstrated through an example.

Example

Suppose a courseware developer is designing a courseware. his operation sequence is as follows

i. Present theorem 1.
ii. Present question 1.
iii. If response is not true jump to step i else jump to iv.
iv. Present theorem 2
v. Halt.

We assume for simplification that we will never get into end less loop between step i end step iii, which may result if response is always false.

Before any teaching or testing can be done, the testing and learning modules must be defined in the system directory with necessary attribute.

Let us assume that we are teaching two propositions, they are

i. SKY IS BLUE

ii. SEA IS ALSO BLUE

First we teach students that sky is Blue, then we ask if sky is red. If the students' response is false we repeat the privious text again, else we teach the next proposition.

The following program creates propositions,

```
• C_ DIR
•L
L• T, TH1, D1
T• SKY IS BLUE
T• \$
L• T, TH2, D1
T• SEA IS ALSO BLUE
T• \$
L•
```

The following program creates test and response module.

```
L•E, TEST, D1,D2
E• IS SKY RED ? /
R•⟨R⟩ : : = GO TO   3 ELSE GO TO 1
R• \$
L• \ $
```

In the above two program TH1 and TH2 are of same type. TEST is different in attribute (capability) that will be presented to device D1 and Test response will be collected from device D2. Detail of syntax rule is given in appendix a.

For program sequence of actual learning presentation, we may run following program coded in LSL.

```
•P
P•⟨O⟩ = [(X) A]
X• ⟨P1, T1,P2,H⟩ /
A• ⟨TH1, TH2⟩ ,  ⟨TEST⟩ /
P• \ $
•
```

To run this program in actual situation one will code

```
•R O
—
```

* and any character left to it in the command line will be promped by the system,(maximum two characters).

5.4 CONCLUSION

We have presented the program syntax form for the LSL. A primitive example is also presented. With step by step design of the learning and program module, we can obtain very large and complex learning system. Since the system is based on red time feed back from students, it can be applied to any real time system with properly defined configuration.

CHAPTER VI

DESCRIPTION OF LEARNING SYSTEM LANGUAGE

6.1 INTRODUCTION

Learning system language, although designed for programming learning systemes, it can be used for real system applications as well. The programing language is highly structured as such whole system design is based on selecting modules and defining their interactions.

6.2 EXECUTION ATOM

The basic structure which defines the operation of a real system in LSL is the execution atom. An execution atom is defined by the following syntax

$$\langle n_1 \rangle = [\,(n_2)\ n_3\,] \Rightarrow n_4$$

where n_1 defines the execution atom. n_1 is a variable name which must be supplied by the user. n_1 defines a module or level of structure during execution. It is called name of EA. n_2 is the set of operation definer. n_2 actually a tuple, where the elements are executed in sequence if $\forall i(\ \bar{n}_{1_2} \in n_2\ )$ is true. We write $n_2 = \langle \bar{n}_{12}, \bar{n}_{22}, \ldots, \bar{n}_{n2} \rangle$ $\bar{n}_{12}$ is generally of three types, they are,

i) Output operation
ii) Input operation
iii) Test operation

n_2 may be a empty tuple in that case n_1 will be a null execution atom, which is equivalent to no operation condition of conventional programming language. n_2 is call operative function of EA.

n_3 is a set of different tuples, normally n_3 contains set of three tuples, which corresponds to element type variation of n_2. For each type of operation of n_2 there is a n - tuple in n_3. We may write

$$n_3 = \{ n_{13}, n_{23}, n_{33} \}$$

where n_{13} to n_{33} are tuples correspond to type operation in n_2. The sequence whether n_{13} is input n_{23} is output and n_{33} is test depends upon installation options. n_3 is called the domain of EA. n_4 is optional. n_4 is produced by certain special operation of n_2 upon n_3. Thus n_4 is a derived element. This in very helpful during defining nested execution atom. It is called codomain of EA.

6.2.1 Nested Execution Atom. Nested EAs are defined by replacing domain of EA by the name of another EA. If O is a primary EA,

$$\langle O \rangle = [(X)P] \Rightarrow E$$

Then we can difine nested EA O' of level 1 by

$$\langle O' \rangle = [(X)O] \Rightarrow E'$$

nested EA of level 2 as

$$\langle O'' \rangle = [(X)O'] \Rightarrow E''$$

and so on. During nesting it is important to define codomain. During nesting, operation is done on the codomain of privious EA and a new codomain is formed. The next nesting operation will operate on this codomin.

Each EA is an independent program modules, as such normally there can not be any communication between the modules through variable (there is no global variable in EA). Each EA can perform output, input, and conditional branch upon certain test operation. These output, input and test program units does not share any variable among them. As such poof of correctness can easily be establish considering these small program unit at a time. By step wise refinement method the proof of correctness can be established on proof of correctness of these smaller units.

Although a program unit (a program unit is the elements of tuple of domain of EA) can not exchange variable directly, it can call another EA and it can write to a stratch pad area or can read from that stratch pad area thus establishing communications among program units and different EAs.

6.3 PROGRAM UNIT

As it has been discussed earlier these are three distinct types of program unit in LSL. The characteristics of program unit depends upon the target user of the LSL. As such one may have to tune these program unit before implementations. Definations used in this literature are for microcomputer based learning applications. For using control of a large process plant or control of an air craft, the input output capabilities may have to be changed according to particular application at hand.

During specification of program unit the syntex forms below are used,

type, name, service
definition

type : first line of any program unit describe the type of program unit, is it output (if output what type of output, i.e., Text, graphic, sound etc similarly for input and text) input or test.

name : name of the program unit which will be referred in the tuple of operation and domain of EA. It is a variable name supplied by the user.

device : logical device type that will be used by the program unit. If omitted system scratch pad area will be assumed.

definition : Here the program unit is defined, If it is an output program unit (PU). Actual text that will be resented to device will be placed here. The definition may contain anothor single type EA (single type EA is such that it only contains one type of program unit) with no type mismatch. Definition of test program unit may contain algol like statement to perform branch and input output operation. Variable type defines here are those confirms to algol.

6. RUNNING THE SYSTEM

The implementation of the system may be viewed as interpretive or compiler type. In some critical real time application interpretive LSL may be inadequate for its slow speed. In RUN procedure EAs are presented in sequence by their name. Since no global variable subroutine and recursion is present in the program specification debugging is relatively easier.

6. CONCLUSION

In LSL efforts were to make the notion of "abstraction" the control theme of the language. The general notionof everyday computer user about language that the language and the form of expression of that language is subtle and involuted. Unless very careful one is likely to fall into what Alan perlis has called the "Turing Tarpit" Roughly stated, the turing tarpit is the argument which holds that, on theoretical grounds, any computation which can be expressed in one of the familiar programming languages can also be expressed in any of the others including turing machines. The argument is correct in so far as it goes, but it ignores our human limitations in dealing with programs and complex systems.

A programming language provides facilities which allow comprehensible expression of algorithms, but one must remember that language is not a panacea. A language, can not for example, prevent the creation of obscure program, the ingenious programmer can always find an infinite path to obfuscation.

CHAPTER VII

DISCUSSION & RESULT

While designing a real time application programming language, we choose Learning System as our target system for its potential future application prospect in Bangladesh. The purpose of the language is to provide a easy means to course designers so that with the help of a computer they can handle a relatively complex system like human learning system. The introduction of "structured Programming" by Edsger Dijkstra has revolutionized the programming methodology as an academic discipline. Following Dijkstra, we used abstraction to decompose the programming task into "intellectually manageable" parts, and the disciplined use of control structure in the development of the software specifications.

Special-purpose programming languages are one of the sources from which important new semantic ideas, eventually to be incorporated in general-purpose language, arise. Ideally, the semantic structure of a special-purpose language defines a logical framework in which the objects and processes typical for the language's intended application area can be described rapidly and conveniently, while language's syntax allows the customary dictions of this application area to be used without much distortion. The range of "typical " programs for which a special-purpose lanuages must guarantee adequate performance can vary quite widely, depanding on the language's intended application. Thus implementation of special-purpose language like LSL will pose a wide variety of challenges to the language optimizer.

LSL is not yet been field tested. The convenience limitations are thus can not be determined precisely. The syntax that has been proposed will certainly have to be revised with the feed back from the end user.

R E F E R E N C E S

1. BOYD,G.M., 'Four ways of providing Computer Assistted Learning and their probable impacts', Comp & Edu Vol 6, PP 305-310,1982.
2. CORIMER,V.K. 'Cybernatics and Information Processing in Human Learning with particular reference to audlts', Progress in Cybernetics and System Research, Vol IV,pp 14-21.
3. DIJKSTRA W.E. 'Guarded Commands non Determinacy and Formal Derivation of Programms', pp 233-242, Current Trends in Programming Methodology, Vol I, Printice Hall Inc, Englewood, Cliffs, N.J. 1978
4. DIRKEWAGER,A.I. ' Human Behavior in Information Processing System' PCSR, Vol IV, pp 14-21.
5. DWYER,T.A. and CRITCHFIELD,M. 'Multi-computer Systems for the support of inventive learning', Comp & Edu Vol 6, no 1 pp 7-12, 1982.
6. GOGUEN J.A. and WAGNER,E.G. ' An Initial Algebra approach to the Specification, Correctness and Implimentation of abstract data types', pp 80-149, CTPM, vol IV, 1978.
7. GUTTANS J.V. and HOROWITW E.M. 'The Design of data Specifications' pp 60-79, CTPM, Vol IV, 1978.
8. HATIVA N. 'Computer Guided Teaching - An effective aid for group instruction', Comp & Edu, Vol 8, no3, pp 293-303, 1982.
9. LEIBLUM,M.D. ' A CAI Service group considers Computer managed instruction and the Interactive Instructional System', Comp & Edu Vol 6, pp 159-164, 1982.
10. LISKOV,B and ZILLES S. 'An Introduction to Formal Specifications of Data Abstractions', CTPM, Vol I, pp 1-32, 1978.
11. MITCHEV P.D. 'Network Flow theory models of individualized instructional systems', PCSR, Vol Iv, pp 105-112.
12 PARNAS D.L. 'On the criteria to be used in decomposing System into modules,', CACM, 15,12(December 1972). pp 1053-58. CR in Ref-16.
13. PARNAS D.L. 'On a "Buzzword": Hierarchical Structure', Proceeding IFIP Congress 74, Vol 2, North Holland Publication (Amsteradam(1974)
14. RAHMAN S.M. and ISLAM S.M.R. 'Microcomputer Based Learning System - A Formal Approach', Regional UNESCO meeting for Universities on Computer Aided Instruction, Dhaka Nov 12-17, 1984.
15. REUVER H.A. 'Learning within the Context of General Systems Theory' PCSR, Vol IV, PP 113-121.
16. ROBINSON L. and LEVITT K.N. NEUMANN P.G. SAXENA A.R. 'The Formal Methodology ofor the Design of Operating System Software', CTPM, Vol I,pp 61-110, 1978.
17. SCHWARTZ J.T. 'Program Genesis and the Design of Programming Language, CTPM, Vol IV, PP 185-215, 1978.
18. STANAT,D.F. and McALLISTER D.F. 'Discrite Mathematics in Computer Science', Printice Hall Inc, Englewood, Cliffs, N.J. 1977.
19. WULF W.H. 'Language and Structured Programms', CTPM, Vol I, PP 33-60.
20. VAN D. VANGERAK K.D. MOONER W.F.' The Choice of CAL System,' Comp & Edu Vol 6, pp 361-368, 1982.
21. Department of Electronics, New Delhi, India - 'Extract From the report on Computer Aided Instruction

Appendixes

A

S Y N T A X S U M M A R Y

The conventions used to illustrate syntax rule are described in article 5.3.

To terminate the current command mode (current command mode is the mode where the system is currently running) one should enter<\$> . When current command mode is terminated, system will return to the last command mode from where current command mode was entered.

Description of the field used in syntax representation.

Mode Identifier : Mode Identifier identifies the mode of the command. It will always be prompted by the system.

Operation : Operation is the exact code that will be entered by the user for intended operation.

Operand : Operand for the operation, normally user defined name. Some of the part or whole of this field sometimes may be omitted, where the system will assume a default value.

Mode Identifier	Operation	Operand	Remarks
•	nil	nil	indicates system is in command mode
•	C_DIR	nil	Command directory. Valid only if logged on as courseware developer. System mode changes to course ware mode.
•	L	device	Changes system into courseware load mode. Valid only if there was any C_DIR command. device= logical device from which successive load instructions will be read. If omitted, logged on device(device from

Mode Identifier	Operation	Operand	Remarks
			which system was initially logged on) will be assumed as default.
L*	{T} for text {G} for graphics	name,device1, SP1]device2[	name=name of the program unit that will be used in domain of EA. device1=output logical device where the output will be routed, (device where text or graphics will be presented) if omitted SP1 will be assumed as default. device2=system input device.
L*	E	name,device1, SP1 device2],device3[SP2	name=name of the program unit that will be used in domain of EA. device1=logical device where the output will be routed, (devece where exam modules will be presented). device2=logical device from where response from the environment will be collected(device from whice students' answer will be collected),default is SP2. device3=system input device same as device operand in L command.
T*	nil	body of the text	body of text (exact code) that will be presented during program execution, this mode may be ended by entering the graphic code of slash or by entering end of file marker
G*	nil	body of graphic module	exact code of the module that will be presented during program execution. This mode may be ended by entering graphic string for end of file, which will be defined in the configuration modele of the system.

Mode Identifier	Operation	Operand	Remarks
E*	nil	body of the question unit	exact code of the questions that will be presented to the logical output device. If graphic code for slash is entered, this mode is ended and the system toggls between this and and response mode.
R*	nil	body of the response unit	exact code for the response function coded here. It may contain standard algol like statements. Variables are global between question (E*) and (R*) response unit. Any branching instruction with numeric label (i.e. go to n) will indicate transfer of control to nth element of the tuple of the operative function. Any transfer of control within the program unit may be performed by using a non-numeric variable label like standard control transfer statement. This label must be defined with in the program unit. Labels are global between E* and R* mode. Response for true or false may be tested by using boolean variable. Default name for this variable is R.
*	P	]device[	this command turns system into program mode. System will accept command for EA definition after system is switched into program mode. device=system input device, if omitted logged on device will be considered as default.
P*	⟨n1⟩ =[(n2) n3]⟩]n4[		n1=name of the execution unit n2=name of the operative function n3=name of the domain n4=name of the codomain

Mode Identifier	Operation	Operand	Remarks
n*	⟨pq.....⟩		pqs are the elements of the operative function tuple. p indicates that the program unit under consideration is a member of the pth tuple of the domain set. q indicates that the program unit under consideration is qth member of the tuple. p may be replaced by appropriate alphabet during installation. A value of pq of 00 indicate end of operative function, which is equivalent to normal end of job where control is transferred to host system. 00 may be replaced by H(halt) in configuration section. This mode is ended by entering slash (graphic code) and system toggles between domain and operative function mode. First letter of the name of operative function will be replaced for n in the mode identifier.
m*	⟨name11,name12,....⟩, ⟨name21,name22,.....⟩, ⟨..... ⟩		name11...namenn idicates the name of the program unit that will be executed by the operative function. These names must be defined to the system directory during execution of EA. This mode is ended by entering slash (graphic code) and system toggles between operative function and domain mode(if codomain is also defined, toggle circle becomes operative function,domain, codomain,operative function). First letter of the name of the domain will replace m in the mode identifier.
o*	⟨name11,name12,.....⟩, ⟨name21,name22,.....⟩, ⟨..... ⟩		this mode is optional, and is only available if codomain is defined during definition of EA. This defines the tuple structure of codomain. name11 through namenn are the names of the program unit that will

Mode Identi fier	Operation	Operand	Remarks
			be created during execution of EA. If names present in the directory is also defined here, those program units will be replaced by the program units defined in the codomain. First letter of codomain name will be replaced for o in mode identifier
•	R	name1,name2,...	name1,..namenare the names of different EA defined in the system. EAs will be executed one by one in sequence given in the name list.
•	TEACH	name1,name2,... X'sss'	it is a student subset of commands available to user. This command can be issued offline. name1,name2 are names of the program units that will be routed to a device from where this command was issued. X'sss' are attribute definer of this command. Detail of this will be installation dependent.
•	HCOPY	device,name1,name2 ,.....	it creates a hardcopy of the program units whose names are given in name1,..namen. device is the logical device name where this hardcopy will be routed. It is a student subset command.
•	TEST	name1,name2,...	it simulates test condition. name1,name2,.. are the name of execution atoms. TEST is similar to standard run, but no score is reported to the system. Score is reported only to the student. It is a student command subset.

B

LANGUAGE SUMMARIES

APL

APL is a general-purpose programming language with the following characteristics [reprinted from *APL Language* (Ref. 4 in APL Session Paper)]:

> The primitive objects of the language are arrays (lists, tables, lists of tables, etc.). For example, $A + B$ is meaningful for any arrays A and B, the size of an array (ρA) is a primitive function, and arrays may be indexed by arrays as in $A[3\ 1\ 4\ 2]$.
>
> The syntax is simple: there are only three statement types (name assignment, branch, or neither), there is no function precedence hierarchy, functions have either one, two, or no arguments, and primitive functions and defined functions (programs) are treated alike.

For example, $A \times B + C$ is computed by a right to left scan and would be equivalent to $A \times (B + C)$.

> The semantic rules are few: the definitions of primitive functions are independent of the representations of data to which they apply, all scalar functions are extended to other arrays in the same way (that is, item-by-item), and primitive functions have no hidden effects (so-called side effects).

The sequence control is simple: one statement type embraces all types of branches (conditional, unconditional, computed, etc.), and the termination of the execution of any function always returns control to the point of use.

For example, $\rightarrow (A > 3)/LOOP$ is equivalent to the ALGOL statement **if** $A > 3$ **then goto** *LOOP*.

External communication is established by means of variables which are shared between APL and other systems or subsystems. These *shared variables* are treated both syntactically and semantically like other variables. A subclass of shared variables, *system variables* provides convenient communication between APL programs and their environment.

The utility of the primitive functions is vastly enhanced by operators which modify their behavior in a systematic manner. For example, *reduction* (denoted by /) modifies a func-

HISTORY OF PROGRAMMING LANGUAGES

ISBN 0-12-745040-8

tion to apply over all elements of a list, as in +/*L* for summation of the items of *L*. The remaining operators are *scan* (running totals, running maxima, etc.), the *axis* operator which, for example, allows reduction and scan to be applied over a specified axis (rows or columns) of a table, the *outer product*, which produces tables of values as in *RATE*∘ . ∗*YEARS* for an interest table, and the *inner product*, a simple generalization of matrix product which is exceedingly useful in data processing and other nonmathematical applications.

The number of primitive functions is small enough that each is represented by a single easily read and easily written symbol, yet the set of primitives embraces operations from simple addition to grading (sorting) and formatting. The complete set can be classified as follows:

Arithmetic: + - × ÷ ⋆ ⍟ ○ | ⌊ ⌈ ! ⌹
Boolean and Relational: ∨ ∧ ⍱ ⍲ ~ < ≤ = ≥ > ≠
Selection and Structural: / \ ⌿ ⍀ [;] ↑ ↓ ⍴ , ⌽ ⍉ ⊖
General: ∊ ⍳ ? ⊥ ⊤ ⍋ ⍒ ⍎ ⍕

Symbol	Name/Function	Symbol	Name/Function
<	Less than	≤	Less than or equal
>	Greater than	≥	Greater than or equal
=	Equal	≠	Not equal
∨	Or	⍱	Nor
∧	And	⍲	Nand
-	Negation, Subtraction	+	Identity, Addition
÷	Reciprocal, Division	×	Signum, Multiplication
?	Monadic random, Dyadic random	⍴	Shape, Reshape
∊	Membership	~	Not
↑	Take	↓	Drop
⍳	Index generator, Index of	○	Pi times, Circular
⌽	Reversal, Rotation	⍉	Monadic transposition, Dyadic transposition
⍟	Natural logarithm, Logarithm	⋆	Exponential, Exponentiation
⌈	Ceiling, Maximum	⌊	Floor, Minimum
⍒	Grade down	⍋	Grade up
⊥	Base value	⊤	Representation
|	Absolute value, Residue		
/	Compression	\	Expansion
∘.d	Outer product	d.D	Inner product
d/	Reduction	[ ]	Indexing
⌹	Matrix inverse, Matrix division	,	Ravel, Catenation/Lamination
!	Factorial, Combination	⎕	System Information

Each APL system is generally embedded in a specialized operating system which permits the user to define APL functions through the use of the ∇ operator, to manipulate workspaces and to specify the shared variables between programs in APL and external systems. Within this operating system the user may execute APL statements directly or defer execution by the placement of statements in function descriptions.

APT

APT is a programming language for describing how parts are to be produced by a numerically controlled machine tool. Within the language a programmer can define points, lines, planes, spheres, etc. and specify the movement of a cutting tool in terms of these shapes.

In the accompanying paper on the origins of the APT language, Figs. 7 and 9 give fairly self-explanatory examples of APT programs. This summary gives additional introductory information. [See pp. 325 and 328.]

Most statements in APT programs are used either to define geometric shapes or to specify the motion of a cutter. In describing shapes, the general APT principle is that any geometrically sensible form of description can be used. For example, a point can be described in terms of X, Y, and Z coordinates, or as the intersection of two lines, or as the point tangent to a circle and a line. A line can be described as passing through two points, as passing through a point and parallel or perpendicular to another line, as passing through a point and tangent to a circle, etc. A circle can be described in terms of a center and radius, as a center and tangent to a line, as tangent to two intersecting lines, etc. For example, the geometry of Fig. 1 (below) can be described as follows:

```
A = POINT/1, 5
B = POINT/2, 3
SIDE = LINE/A, B
JILL = CIRCLE/CTR AT, B, THRU, A
```

JILL is a circle passing through point A and centered at B.

When a geometric description is ambiguous, the ambiguity can be resolved by specifying the relative magnitude of X and Y coordinates. For example, the line through A and

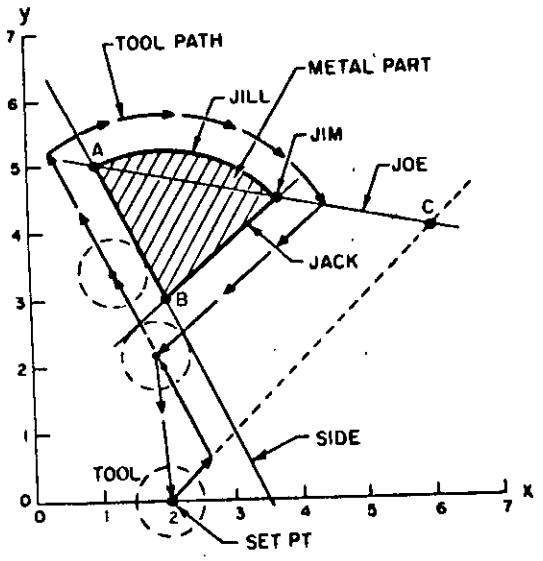


Fig. 1

C intersects the circle JILL at two points. To select the desired point, a programmer must write:

```
JIM = POINT/XLARGE, INT OF, JOE, WITH, JILL
```

Note that A is the point selected by XSMALL.

Motions of a cutter are controlled either by explicitly positioning the cutter (in terms of coordinates or increments from the present position) or by specifying continuous cutter motion along a curve. Motion of the cutter is constrained by a tangency condition to each of two surfaces, the so-called *part* surface and *drive* surface. Motions are terminated by specifying a tangency condition to a third surface, the *check* surface. When this required terminal tangency condition is met, the motion is complete. An initial motion statement positions the cutter within a specified tolerance from the drive and part surfaces and on the correct side of the part surface, e.g.,

```
FROM, POINT/2, 0
IN DIR, POINT/C
SIDE = LINE/A, B
GO TO/SIDE
TL LFT, GO LFT/SIDE
```

As the diagram shows, this command sequence states that the initial cutter position is at point 2,0; it is moved from this point in the direction of (the previously defined) point *C*. Motion continues until the cutter reaches (i.e., is tangent to) the line passing through points *A* or *B*. With the tool (i.e., cutter) on the left side of that line (TL LFT), the tool is moved to the left (with respect to the preceding motion toward point *C*) along the line called SIDE. GO LFT, GO RGT, GO FWD, etc., are commands used to indicate when the cutter is to follow a different curve as the part surface. For example, the next motion command is GO RGT, CIRCLE/CTR AT, B, THRU, A, meaning the cutter continues along SIDE until tangent to the specified circle and then it goes to the right along the specified circle.

The geometric definitions and motion commands in an APT program are transformed first into a broken line cutter path encoded as the cutter location tape (CLTAPE). The CLTAPE also includes commands for special functions such as turning coolant on and off. The CLTAPE is then transformed into codes for a specific controller-machine tool configuration. The mathematical computations yielding the coordinate sequences are performed by a part of the APT system called the ARELEM (arithmetic element) program; the transformation into machine tool codes is done by the APT *post-processor*. The use of different post-processors permits the APT language to be used for a wide variety of machine tools.

ALGOL 60

The term ALGOL is reported to have sprung from "ALGOrithmic Language." Many forget, however, that ALGOL is the second brightest star in the constellation Perseus and part of an eclipsing binary. ALGOL exhibits a variation in light, which is caused by the fact that once every 69-odd hours it is partially eclipsed by a dark body, its partner star, for about 10 hours. But note that ALGOL always manages to regain its brilliance. ALGOL and its eclipsing companion (could it be FORTRAN?) are miles and miles apart (about 6,000,000).

ALGOL 60 and FORTRAN are both algebraic languages designed primarily for writing programs to solve numerical problems. Both have simple variables and arrays, declarations, assignment statements, conditional and iterative statements, and procedures (subroutines).

ALGOL had the advantage of being developed several years after FORTRAN, and it exhibited several significant advances in programming language design. Of prime importance was its sense of simplicity, structure, and conciseness—its mathematical elegance if you will. The term "ALGOL-like" refers to these qualities as well as to the basic structure of the language. In attempting to popularize a new language many a designer has encroached upon the term "ALGOL-like", causing one eminent computer scientist to remark that "ALGOL was indeed an achievement; it was a significant advance over most of its successors."

Of course, ALGOL has its faults, and some aspects of it that were earlier praised are now considered to be clumsy, awkward and ill-defined. In many respects, by today's standards ALGOL is as inadequate as FORTRAN.

Listed below are some major features of ALGOL.

1. ALGOL 60 is described using a formal notation for syntax (Backus–Naur Form, or BNF). The defining document (only 16 pages in *Communications of the ACM*) is short, concise, and exceptionally clear. The document differentiates between the publication language and the hardware language for ALGOL.

2. The possible types of variables are *integer*, *real*, and *Boolean* (or logical). Each variable and array has to be declared.

3. The language has a nested statement structure; thus one can write:

```
if f then begin h := y; z := w end
    else if h ≠ z then h := y
```

4. The nested statement structure allows the introduction of *blocks*, which in turn allows the declaration of variables *local* to a program segment and brings on scope problems.

```
⋮
x := y;
begin real r;
    use r in here only
end;
⋮
```

5. Blocks introduce the possibility of *dynamic arrays* and systematic *dynamic storage allocation*. In the following, the size of array *b* depends on the value of *global variable n*:

```
begin array b[1:n];
    use array b here only
end
```

6. Procedures (subroutines) can be recursive. Parameters are call-by-value or call-by-name, a concept not thought well of these days. The following example illustrates *procedure definition*, *specification* of a *formal parameter* (*n*), *declaration* of a *local variable* (*i*), *assignment* to the procedure name (*factorial* := 1), recursive *function invocation* (*factorial* (*n* − 1)), and *nested statement structure:*

```
integer procedure factorial (n);
  value n; integer n;
  begin integer i;
    if n = 0 then factorial := 1
            else begin i := factorial (n - 1);
                       factorial := n*i
                 end
  end
```

BASIC

Beginner's All-purpose Symbolic Instruction Code or BASIC was originally developed by T. E. Kurtz and J. G. Kemeny at Dartmouth College in 1963–1964. The original version consisted of 14 statement constructs. BASIC was one of several languages operating in a general-purpose time-sharing system also developed at Dartmouth. The simpler user commands, numbering about 8 or 10, have since become associated with the BASIC language.

A BASIC program is created via the NEW command, listed by the LIST command, edited by a DELETE command, executed by the RUN command, and saved by the SAVE command. Previously saved programs are retrieved from their file by entering OLD and the desired program name.

Every program statement begins with an integer line number and a statement keyword. Thus, assignment statements are exactly like the FORTRAN assignment, but are prefixed with an integer label and the keyword LET. For example,

```
1150 LET X = B*B - 4.0*A*C
```

computes $B^2 - 4AC$ and stores the result in X.

BASIC manages three data types: (1) numeric variables, (2) string variables, and (3) one- or two-dimensional array variables of either string or numeric type. All numeric computation is presumably performed in floating-point arithmetic, with truncation being performed during indexing. The subscript 0 is defined for array variables. If an array variable appears without a corresponding DIM statement, default subscript ranges of 0–10 in each dimension are provided.

Input–output is done in BASIC by the READ, INPUT, PRINT, and DATA statements. READ and DATA work together by assigning values from the DATA statements to the variable list in READ. INPUT and PRINT produce results at the user's terminal, interactively.

Control is performed by GOTO, IF THEN, ON GOTO, FOR NEXT, and GOSUB statements. The GOTO statement causes an unconditional branch to any statement in the program. The IF THEN statement causes a conditional branch (two-way), e.g., `IF A = B THEN 350`. The ON statement produces a branch to one of many statements depending on the value of a truncated integer index.

Subroutines are part of the same global environment as the main program and other routines. Thus, all data is accessible to all program "modules." A GOSUB 1000 causes subroutine invocation beginning with statement 1000 and continuing until a RETURN is reached. The subroutine has access to any variable used in any other segment of the program.

FORTRAN influenced "statement functions" provide single statement functions in BASIC. A function is "called" by an instance identical to array variable usage.

The following program segment illustrates the nature of BASIC programs. The program does nothing of importance, except illustrate most of BASIC's constructs.

```
 1 LET I = 0
 5 REMARK A$ IS A STRING, B IS AN ARRAY.
10 DIM A$(3), B(10)
15 PRINT "READY?"
20 INPUT A$
25 REMARK A$ = "YES" OR "NO".
30 IF A$ = "YES" THEN 50
35 LET I = I + 1
40 IF I < 3 THEN 15
45 GO TO 99
50 REMARK "YES"
55 INPUT N
60 ON N GO TO 70, 80
65 GO TO 55
70 PRINT "N=", N
75 READ B
80 PRINT "N=", N
85 REM SHORT FOR REMARK.
90 DATA 10,20,30,35,40,50,55,60,70,80
99 STOP
```

BASIC is unusual in that variable names are limited to a single letter followed by an optional digit if numeric, and further followed by a $ if of type string.

The looping construct

```
FOR I = 1 TO 10
    :
NEXT I
```

is bracketed with a NEXT statement following the loop body, but the loop counter is tested upon entry of the loop.

Subroutines are defined by an instance of a GOSUB, thus making modularity and scope a vague concept! Built-in functions resemble a pocket calculator's capability, and arithmetic uses the exponentiation caret as current versions of ASCII have eliminated the ↑.

Later versions of BASIC (various names, depending on the vendor) include string functions, matrix operations, and sequential and random file I/O.

COBOL 60

Notation

The syntax of COBOL 60 is defined through a metalanguage that uses:

lowercase letters for metalinguistic variable names;

uppercase letters for optional fixed words in the language, which are called "noise words";

underlined uppercase letters for required fixed words in the language;
brackets, [and], to enclose optional elements,
braces, { and }, to enclose elements from which a choice must be made; these are usually listed vertically;
ellipsis, ..., to show that the previous syntactic unit is to be repeated an arbitrary number of times.

The character set in COBOL 60 consists of 51 characters, comprising the 10 digits, the 26 uppercase letters and the special symbols:

+ − * / = $ < > , . ; " ( ) blank

Identifiers are composed of not more than 30 digits, letters, and hyphens. Certain keywords and connectives, which have been defined to improve the readability of the language, are reserved and may not be used as identifiers.

Programs

COBOL 60 programs are divided into four *divisions:*

IDENTIFICATION division: This begins each program and names the program and provides overall program documentation.

PROCEDURE division: This contains the algorithms specified as a sequence of executable units. These units are arranged hierarchically as *sections, paragraphs, sentences,* and imperative or conditional *statements*. An imperative statement consists of a *verb* followed by operands. The verbs of COBOL 60 are:

ACCEPT—obtain input from a low volume device.
ADD—add two or more quantities and store the sum.
ALTER—modify a predetermined sequence of operations.
CLOSE—terminate the processing of both input and output files.
COMPUTE—permit the use of formulas.
DEFINE—allow the introduction of new verbs.
DISPLAY—allow for visual display of low volume information.
DIVIDE—divide one number into another and store the result.
ENTER—allow more than one language in the same program.
EXAMINE—replace or count occurrences of a given character in data.
EXIT—mark the end point of a loop if needed.
GO TO—depart from the normal control sequence
INCLUDE—incorporate a library subroutine into the source program.
MOVE—transfer data, with editing, to one or more data fields.
MULTIPLY—multiply two quantities and store the result.
NOTE—mark a comment.
OPEN—initiate the processing of both input and output files.
PERFORM—depart from the normal control sequence to execute one or more statements a specified number of times and return.
READ—obtain the next logical record from an input file.
STOP—halt the computer either permanently or temporarily.
SUBTRACT—subtract one quantity from another and store the result.

USE—specify special I/O exception handling procedures.
WRITE—release a logical record for an output file.

The conditional statement is of the form

IF . . . THEN . . . ELSE . . .

Conditional statements can be nested to any depth.

DATA division: This contains descriptions of the data and files used by algorithms in the PROCEDURE division. The basic data structure is a *record* which is a multidimensional heterogeneous array. Records are also the basis for external files. The specifications in the DATA division consist of a sequence of *entries*, each of which consists of a *level-number*, *data name*, followed by a series of *clauses* that describe the data. The level-number specifies the hierarchical relationship between the data item described in the entry and the data described in other entries. Level-numbers start at 01 for records and higher level-numbers (less than 50) are used for subordinate items. There are also *special* level-numbers with specific meanings. The clauses are used to specify:

data type—alphabetic, numeric, or alphanumeric,
range of values,
location of editing symbols, e.g., dollar signs and commas,
location of radix point,
location of sign,
number of occurrences in a list or table
alignment in computer words
dominant usage—computational or display.

ENVIRONMENT division: contains information about the hardware on which the program is to be compiled and executed.

FORTRAN

The FORTRAN language was intended to be a vehicle for expressing scientific and mathematical computation, while allowing efficient object code to be produced. Since there was an emphasis on scientific computation in FORTRAN and FORTRAN II, little provision was made for heterogeneous data structures. Only arrays of up to three dimensions were possible, and they had to be declared explicitly†:

```
DIMENSION ALPHA(10,15), BETA(12)
```

Input and output statements allowed a little more flexibility, in that provision was made for controlling the format of information read from or written to the outside world (as opposed to reading and writing binary information on disk or tape):

```
    WRITE (5,200) MTH, BETA, Y
200 FORMAT (6H MONTH, I3, 8H; QUOTAS, 12F4.0, 10H; FORECAST, F6.2)
```

where MTH was assigned type integer because its first character was one of I, J, K, L, M,

† Examples are given here in FORTRAN II notation.

and N, and BETA was assumed to have been declared a vector as in the earlier example. The character string 6H MONTH specified that the word MONTH should print (i.e., be written on unit 5, as indicated in the WRITE statement), with an initial space character for carriage control. The format information could be changed at "run time" in later versions of FORTRAN, but not at first.

Actual computation was accomplished by using the assignment statement, which allowed expressions on the "right-hand side," but did not allow mixed types (such as integer and floating-point). The following was acceptable:

```
X1 = F(I+1,M1) * DAD - (C + 1.0)/Y
```

In this example, F was understood to be a function name, and F(I+1,M1) a function call, if F was not previously declared to be a two-dimensional array in a DIMENSION statement. As indicated, arguments in a function call could be expressions (with expressions called "by value" and variables called "by reference"). If F were an array name, the subscripts could only have the form $c_1v \pm c_2$, where c_1 and c_2 were integer constants or missing, and v was a variable of integer type. If the variable (possibly subscripted) on the "left-hand side" of the assignment statement was not of the same type as the value of the "right-hand side" expression, the value was converted (at "run time") to the appropriate type before assignment.

Unconditional transfer of control was created by: (i) the "unconditional GO TO," as in

```
GO TO 31
```

where 31 is a statement label; (ii) the "computed GO TO," as in

```
GO TO (30,31,32,33),I1
```

where I1 took one of the values 1, 2, 3, or 4, to indicate the position in the list of the label of the next statement to be executed; and (iii) the related "assigned GO TO."

Conditional sequencing of control was accomplished by the IF statement:

```
IF (A+B-1.2) 7,6,9
```

which transferred control to the statement labeled 7, 6, or 9, respectively, depending on whether the expression A + B − 1.2 was negative, zero or positive.

Iteration was provided for the DO statement:

```
DO 17 I = 1,N,2
```

which specified that the set of statements which followed, up to and including that labeled 17, was to be executed with I first assigned the value 1, and again with I successively incremented by 2, until I exceeded the value of N. (The statements in the "body" of the iteration would always be executed at least once, since the comparison of I with N occurred after their execution.)

Programmer influence on storage allocation occurred through the COMMON and EQUIVALENCE declarations, as in:

```
COMMON X,Y,Q
DIMENSION X(17,3),Q(20),R(5)
EQUIVALENCE (R(2),Q(6))
```

which would indicate that addresses were to be assigned in a region known as COMMON

so that the array X would occupy the first 51 spaces, the variable Y would come next, and the vector Q would occupy the next 20 spaces. The vector R would also be assigned in COMMON, with the same address assigned to R(2) and Q(6). (Then R(1) and Q(5) would occupy the same space, and so on.)

In FORTRAN it was possible to call a number of "built-in" functions and functions defined by the user on a single line, but it was not possible, until FORTRAN II was available, to compile a separate subroutine or function (which differed from a subroutine in that it returned a value as a result). In FORTRAN II, a typical (a) subroutine and (b) function might be:

```
(a) SUBROUTINE SWAP(I,J)      (b)      FUNCTION LESS(M,N)
    M = I                              LESS = N
    I = J                              IF (M-N) 1,2,2
    J = M                            1 LESS = M
    RETURN                           2 RETURN
    END                                END
```

where the name of the function LESS received the (integer) value to be returned. The facility for independent translation of subroutines and functions in FORTRAN II made it possible to provide subprogram libraries in the host operating system.

GPSS

The General Purpose Simulation System language has been developed over many years and implemented on several different manufacturers' machines. Of the two current versions, GPSS/360 models can operate with GPSS V, with some minor exemptions and modifications. GPSS V is more powerful and has more language statements and facilities.

The system to be simulated in GPSS is described as a block diagram in which the blocks represent the activities, and lines joining the blocks indicate the sequence in which the activities can be executed. Where there is a choice of activities, more than one line leaves a block and the condition for the choice is stated at the block.

The use of block diagrams to describe systems is, of course, very familiar. To base a programming language on this descriptive method, each block must be given a precise meaning. The approach taken in GPSS is to define a set of 48 specific block types, each of which represents a characteristic action of systems. Each block type is given a name that is descriptive of the block action. The entities that move through the system being simulated depend upon the nature of the system. For example, a communication system is concerned with the movement of messages, a road transportation system with motor vehicles, and a data processing system with records. In the simulation, these entities are called transactions. The sequence of events in real time is reflected in the movement of transactions from block to block in simulated time.

Transactions are created at one or more GENERATE blocks and are removed from the simulation at TERMINATE blocks. There can be many transactions simultaneously moving through the block diagram. Each transaction is always positioned at a block and most blocks can hold many transactions simultaneously. The transfer of a transaction from one block to another occurs instantaneously at a specific time or when some change of system condition occurs.

A GPSS block diagram can consist of many blocks. An identification number called a

location is given to each block, and the movement of transactions is usually from one block to the block with the next highest location. The locations are assigned automatically by an assembly program within GPSS so that, when a problem is coded, the blocks are listed in sequential order. Blocks that need to be identified in the programming of a problem must be given a symbolic name.

Clock time is represented by an integral number, with the interval of real time corresponding to a unit of time chosen by the program user. The unit of time is not specifically stated but is implied by giving all times in terms of the same unit. One block type, ADVANCE, is concerned with representing the expenditure of time. The program computes an interval of time called an action time for each transaction as it enters an ADVANCE block, and the transaction remains at the block for the interval of simulated time before attempting to proceed.

The action time may be a fixed interval, including zero, or a random variable and it can be made to depend upon conditions in the system in various ways. An action time is defined by giving a mean and modifier for the block.

Transactions have a priority level and they carry items of data called parameters. These can be signed integers of fullword, halfword, or byte size, or they can be signed floating point numbers.

The program maintains records of when each transaction in the system is due to move. It proceeds by completing all movements that are scheduled for execution at a particular instant of time and that can be logically performed. Where there is more than one transaction due to move, GPSS processes transactions in their priority order, and on a first-come, first-serve basis within the priority class.

Normally, a transaction spends time only at an ADVANCE block. Once the program has begun moving a transaction, therefore, it continues to move the transaction through the block diagram until one of several circumstances arises. The transaction may enter an ADVANCE block with a nonzero action time, in which case, the program will turn its attention to other transactions in the system and return after the action time has been expended. Secondly, the conditions in the system may be such that the action the transaction is attempting to execute by entering the block cannot be performed at the current time. The transaction is said to be blocked and it remains at the block it last entered. The program will automatically detect when the blocking condition has been removed and will start to move the transaction again at that time. A third possibility is that the transaction enters a TERMINATE block, in which case it is removed from the simulation. A fourth possibility is that a transaction is placed on a chain to be removed by a change in the simulation or another transaction.

When the program has moved one transaction as far as it can go, it turns its attention to any other transactions due to move at the same time instant. If all such movements are complete, the program advances the clock to the time of the next imminent event and repeats the process.

The TRANSFER and TEST blocks allow some location other than the next sequential location to be selected. This permits the path of the transaction to vary according to the criteria set in the block.

A simple illustration of GPSS provides some of the flavor of the language. A machine tool in a manufacturing shop is turning out parts at the rate of one every five minutes. As they are finished, the parts go to an inspector who takes 4 ± 3 minutes to examine each one and rejects about ten percent of the parts. Each part will be represented by one trans-

action, and the time unit selected is one minute. This problem coded in fixed format is presented below:

```
        GENERATE   5            CREATE PARTS
        ADVANCE    4,3          INSPECT
        TEST GE    RN1,100,REJ                  REJECTS
        TERMINATE  1            ACCEPTED         PARTS
    REJ TERMINATE  1            REJECTED         PARTS
        START      1000
```

A GENERATE block is used to represent the output of the machine by creating one transaction every five units of time. An ADVANCE block with a mean of 4 and modifer of 3 is used to represent inspection. This results in the equal probabilty of the values 1, 2, 3, 4, 5, 6, or 7. After inspection 90% of the parts go to the next block. Ten percent, those which draw a random number of less than 100 out of the range of 0 to 999 proceed to the REJ location. Since there is no further interest in following the history of the parts in this simulation, both locations reached from the TEST block are TERMINATE blocks.

The last line is for a control statement START, which indicates the end of the problem definition and contains the value of the terminating counter. In this case the START statement is set to stop the simulation after 1000 transactions have reached the two TERMINATE blocks.

When the simulation is completed, the program automatically prints an output report, in a prearranged format, unless it has been instructed otherwise. The first line gives the time at which the simulation stopped. This is followed by a listing of block counts. Two numbers are shown for each block. The first is a count of how many transactions were in the block at the time the simulation stopped. The second shows the total number of transactions that have entered the block during the simulation.

The results as indicated by the total block counts at the two TERMINATE blocks were 888 and 112, respectively. These results show that of the 1000 parts inspected, 88.8% were accepted and 11.2% were rejected. The simulation results differ from the expected results of 90% and 10%, but are within the expected values for 1000 trials.

Certain block types are constructed for the purpose of gathering statistics. A realistic modification of the example allows parts to accumulate on the inspectors work bench if the inspection does not finish quickly enough. It would be of interest to measure the size of the queue of work that occurs. GPSS will automatically measure the length of stay in QUEUE block and print the results showing the average stay duration, the maximum stay, and the number of transactions which passed through the QUEUE block without delay.

JOSS

JOSS is one of the earliest interactive, multiuser languages to become operational (the first demonstration was in 1963) and as such had a strong influence on many similar systems and languages which proliferated shortly thereafter. The system (hardware and software design and implementation) was developed by J. C. Shaw, T. O. Ellis, I. D. Nehama, A. Newell, and K. W. Uncapher at the RAND Corporation. The name is an acronym derived from: JOHNNIAC—the RAND-built Princeton-type computer named in honor of the mathematician John von Neumann; OPEN SHOP—designating operation by person-

nel other than those attached to the computing center; and SYSTEM—implying that JOSS is more than just a computer language.

At the time of initial system hardware specification, no acceptable typewriter-based terminals were commercially available, nor was the JOHNNIAC configured to accept multiuser input–output of this type. Therefore, special equipment was designed and built at RAND to meet these needs. The user terminal was based on the IBM model B typewriter, essentially equivalent to those used by RAND secretaries at that time. A special character set was specified to provide the symbols required by the JOSS language. Control and interface of up to eight remote consoles was provided by multiple-typewriter-control-system (MTCS) hardware. The development of these JOSS hardware components overlapped the development of the JOSS language.

The design of the JOSS language itself was purposely limited in scope: the goal was to attract the casual user in need of modest computational power. For this reason, JOSS was restricted to numeric calculations with no attempt to provide any of the many symbolic capabilities of the general purpose computer. At the same time, a conscious effort was made to avoid any stumbling blocks—however subtle—which might deter a potential user, and this constraint eventually became the limiting factor in language and system design. For example, JOSS provides *exact* input and output of both numbers and commands. (By this is meant that there is a one-to-one correspondence between internal and external representations, thus obviating the need to learn "computer arithmetic.") In order to make JOSS a primary rather than a secondary tool, special consideration was given to the ease of specification of report-quality output on standard 8½" × 11" paper, ready for inclusion in formal RAND publications.

In operation, JOSS and the user take turns using the typewriter, and change of control is communicated to the user by visual, audible, and tactile feedback. A permanent record is supplied by the use of a two-color (black–green) typewriter ribbon. The user may enter a *direct command* on one line, followed by a carriage return; JOSS immediately carries out this command and returns control to the user. Alternately, the user may enter one or more *indirect commands:* each takes the form of a direct command preceded by a number which serves to indicate that the command is to be stored (but not carried out), to uniquely identify the stored command, and to specify a hierarchical organization of commands when a collection of these is to be carried out automatically.

Each JOSS command takes the form of an imperative English sentence (which may actually be verbalized). The command begins with a verb, and obeys the conventional rules of English for spacing, capitalization, punctuation, and spelling. Any command, direct or indirect, may be modified by appending a conditional clause; the command will be carried out (at the appropriate time) only if the expressed relationship holds. Several related commands are available to the user for controlling the sequence of execution and/or iteration of stored indirect commands.

All calculation is carried out (in decimal, to 10-digit pencil-and-paper accuracy) using explicit numeric values and/or variables designated by a single lower- or uppercase letter. Additionally, variables may be indexed to provide vector and array capability. Computational expressions are represented in the conventional way, and may be used freely. Indeed, wherever a numeric value may appear in a command, an arbitrarily complex expression denoting a numeric value may equally well appear, subject to the limitations of one line per command and one command per line.

For directing the production of formal output, the user may enter one or more *forms*

composed of strings of characters to be output directly, and of field indicators for specifying the output of numeric results in fixed and/or scientific notation. Pagination control is also provided.

These are all of the essential features of the system; with this background, the accompanying summary provides a complete description of the initial 1963 implementation of JOSS.

DIRECT or INDIRECT

```
Set x=a.

Do step 1.1.
Do step 1.1 for x = a, b, c(d)e.
Do part 1.
Do part 1 for x = a(b)c(d)e, f, g.

Type a,b,c,__.
Type a,b in form 2.
Type "ABCDE".
Type step 1.1.
Type part 1.
Type form 2.
Type all steps.
Type all parts.
Type all forms.
Type all values.
Type all.
Type size.
Type time.
Type users.

Delete x,y.
Delete all values.

Line.

Page.
```

INDIRECT (Only):

```
1.1 To step 3.1.
1.1 To part 3.

1.1 Done.

1.1 Stop.

1.1 Demand x.
```

DIRECT (Only)

```
Cancel.

Delete step 1.1.
Delete part 1.
Delete form 2.
Delete all steps.
Delete all parts.
Delete all forms.
Delete all.

Go.

Form 2:
dist.= ..... accel.= __.____

x=a
```

RELATIONS

= ≠ ≤ ≥ < >

OPERATIONS

+ − · / * ( ) [ ] ||

CONDITIONS

if a<b<c and d=e or f≠g

FUNCTIONS

sqrt(a)	(square root)
log(a)	(natural logarithm)
exp(a)	
sin(a)	
cos(a)	
arg(a,b)	(argument of point [a,b])
ip(a)	(integer part)
fp(a)	(fraction part)
dp(a)	(digit part)
xp(a)	(exponent part)
sgn(a)	(sign)
max(a,b)	
min(a,b,c)	

Punctuation and Special Characters

. , ; : ' " _ # $?
____._____ indicates a field for a number in a form.
. . . . . indicates scientific notation in a form.
# is the strike-out symbol.
$ carries the value of the current line number.
* at the beginning or end kills an instruction line.
Brackets may be used above in place of parentheses.
Indexed letters (e.g., v(a), w[a,b]) may be used in place of x, y.
Arbitrary expressions (e.g., 3·[sin(2·p+3)−q]+r) may be used in place of a, b, c, ...

JOVIAL

JOVIAL (Jules' Own Version of the International Algebraic Language) was one of the first programming languages developed primarily to aid in programming large complex real time systems. Today it remains a major language for these applications and versions of JOVIAL have been implemented on dozens of different computers. The language and its compilers have been developed primarily by the Systems Development Corporation.

JOVIAL was based on ALGOL 58 but includes numerous features not in ALGOL (58 or 60) which make it particularly useful for programming large scale systems. The most important of these is the COMPOOL (COMmunications POOL), a central repository of data descriptions which permits programmers to reference data items without concerns as to how they are represented on some particular computer.

The structure of JOVIAL programs is very similar to those of ALGOL. JOVIAL provides a block structure for compound statements; its statement structure, loop structures, and procedure calls are very similar to those of ALGOL.

Basic data items include integer and real numbers, booleans, strings, and status items for example, RED, BLUE, GREEN may be the three possible values of some status items. If the variable *x* were such an item then the assignment

```
X = RED
```

and the equality relation

```
X EQ GREEN
```

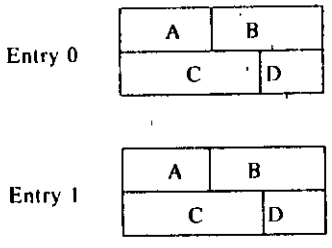
are permitted.

There are the usual arithmetic, relational, and logical operators and it has a conditional statement which can have multiple arms. Thus one can write

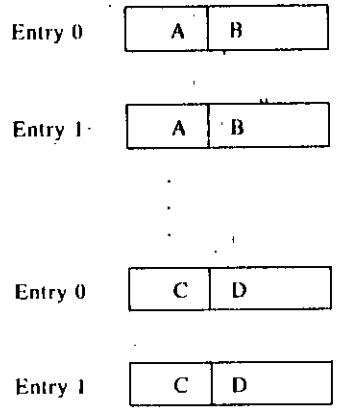
```
IFEITH P $ I = 1 $
ORIF   Q $ I = 2 $
ORIF   R $ I = 3 $
END
```

to set I to 1, 2, or 3 or leave it unmodified according to the truth of the booleans P, Q, and R.

JOVIAL differs markedly from ALGOL (and most other languages) in the area of data definitions. At the lowest level, JOVIAL provides for *items* (basic data elements like numbers, strings, and so on). At the next level are *entries*—ordered collections of items (they would be called structure or records today). At the top level are *tables*, which are sequences of entries. JOVIAL has means for the user to control how tables are actually represented in some computer, permitting for example, the user to call for "serial" or "parallel" storage of tables to simplify indexing operations when more than one word of storage is required to accomodate all items in an entry. In serial tables all words containing an entry are contiguous:



A parallel table with the same items is depicted



JOVIAL also permits machine language code inserts and provides some pseudo-operations which reference the COMPOOL to provide the shifting and masking operations required to extract items from an entry thus relieving programmers from being concerned with the detailed structure of tables and permitting this structure to change without programs being adversly affected. For example the code sequence

```
CLA ITEM  Move ITEM to accumulator
ETR ITEM  AND with a mask.
POS ITEM  Position least significant bit
          to be rightmost
```

will extract ITEM from the word containing it and position it in the accumulator.

A Micro-Manual for LISP

LISP data are symbolic expressions that can be either *atoms* or *lists*. *Atoms* are strings of letters and digits and other characters not otherwise used in LISP. A list consists of a left parenthesis followed by zero or more atoms or lists separated by spaces and ending with a right parenthesis. Examples:

A, ONION, (), (A), (A ONION A), (PLUS 3(TIMES X PI)1), (CAR (QUOTE(A B))).

The LISP programming language is defined by rules whereby certain LISP expressions have other LISP expressions as *values*. The function called *value* that we will use in giving these rules is not part of the LISP language but rather part of the informal mathematical language used to define LISP. Likewise, the italic letters e and a (sometimes with subscripts) denote LISP expressions, the letter v (usually subscripted) denotes an atom serving as a variable, and the letter f stands for a LISP expression serving as a function name.

1. *value* (QUOTE e) = e. Thus the value of (QUOTE A) is A.
2. *value* (CAR e), where *value* e is a nonempty *list*, is the first element of *value* e. Thus *value* (CAR (QUOTE(A B C))) = A.
3. *value* (CDR e), where *value* e is a nonempty list, is the list that remains when the first element of *value* e is deleted. Thus *value* (CDR (QUOTE(A B C))) = (B C).
4. *value* (CONS $e1$ $e2$), is the list that results from prefixing *value* $e1$ onto the list *value* $e2$. Thus *value* (CONS (QUOTE A)(QUOTE(B C))) = (A B C).
5. *value* (EQUAL $e1$ $e2$) is T if *value* $e1$ = *value* $e2$. Otherwise, it value is NIL. Thus *value*(EQUAL (CAR (QUOTE(A B))) (QUOTE A)) = T.
6. *value* (ATOM e) = T if *value* e is an atom; otherwise its value is NIL.
7. *value* (COND (p_1 e_1) . . . (p_n e_n)) = *value* e_i, where p_i is the first p whose value is not NIL. Thus

value (COND ((ATOM (QUOTE A)) (QUOTE B)) ((QUOTE T) (QUOTE C))) = B.

8. An atom v, regarded as an variable, may have a value.
9. *value* ((LAMBDA (v_1 . . . v_n)e)e_1 . . . e_n) is the same as *value* e but in an environment in which the variables v_1, . . . , v_n take the values of the expressions e_1, . . . , e_n in the original environment. Thus

value ((LAMBDA (X Y) (CONS (CAR X) Y)) (QUOTE (A B)) (CDR (QUOTE (C D)))) = (A D).

10. Here's the hard one. *value* ((LABEL f (LAMBDA (v_1 . . . v_n) e)) e_1 . . . e_n) is the same as *value* ((LAMBDA (v_1 . . . v_n) e) e_1 . . . e_n) with the additional rule that when (f a_1 . . . a_n) must be evaluated, f is replaced by (LABEL f (LAMBDA (v_1 . . . v_n)e)). Lists beginning with LABEL define functions recursively.

This is the core of LISP, and here are more examples:

value (CAR X) = (A B) if *value* X = ((A B) C), and *value* ((LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))))) (QUOTE ((A B) C))) = A. Thus ((LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))))), is the LISP name of a function ff such that ff e is the first atom in the written form of e. Note that the list ff is substituted for the atom FF twice.

Difficult mathematical type exercise: find a list e such that *value* e = e.

Abbreviations

The above LISP needs some abbreviations for practical use.

1. The variables T and NIL are permanently assigned the values T and NIL, and NIL is the name of the null list ().

2. So as not to describe a LISP function each time it is used we define it permanently by typing (DEFUN $f(v_1 \ldots v_n)e$). Thereafter ($f\ e_1 \ldots e_n$) is evaluated by evaluating e with the variables $v_1, \ldots, v_n$ taking the values value $e_1, \ldots$ *value* e_n respectively. Thus, after we define (DEFUN FF (X) (COND ((ATOM X) X) (T (FF (CAR X))))), typing (FF(QUOTE ((A B) C))), gets A from LISP.

3. We have the permanent function definitions

(DEFUN NULL (X) (EQUAL X NIL)) and
(DEFUN CADR (X) (CAR (CDR X))),

and similarly for arbitrary combinations of A and D.

4. (LIST $e_1 \ldots e_n$) is defined for each n to be

(CONS e_1 (CONS . . . (CONS e_n NIL))).

5. (AND p q) abbreviates (COND (p q) (T NIL)). ANDs with more terms are defined similarly, and the propositional connectives OR and NOT are used in abbreviating corresponding conditional expressions.

Here are more examples of LISP function definitions:

```
(DEFUN ALT (X) (COND ((OR (NULL X) (NULL (CDR X))) X)
                     (T (CONS (CAR X) (ALT (CDRR X))))))
```

defines a function that gives alternate elements of a list starting with the first element. Thus (ALT (QUOTE (A B C D E))) = (A C E).

```
(DEFUN SUBST (X Y Z) (COND ((ATOM Z) (COND ((EQUAL Z Y) X) (T Z)))
                     (T(CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))))))),
```

where Y is an atom, gives the result of substituting X for Y in Z. Thus

```
(SUBST (QUOTE (PLUS X Y)) (QUOTE V) (QUOTE (TIMES X V)))
                                        = (TIMES X (PLUS X Y))
```

You may now program in LISP. Call LISP on a time-sharing computer, define some functions, type in a LISP expression, and LISP will output its value on your terminal.

The LISP Interpreter Written in LISP

The rules we have given for evaluating LISP expressions can themselves be expressed as a LISP function (EVAL e a), where e is an expression to be evaluated, and a is a list of variable-value pairs. a is used in the recursion and is often initially NIL. The long LISP expression that follows is just such an evaluator. It is presented as a single LABEL expressions with all auxiliary functions also defined by LABEL expressions internally, so that it references only the basic function of LISP and some of abbreviations like CADR and friends. It knows about all the functions that are used in its own definition so that it

```
(LABEL EVAL (LAMBDA (E A)
      (COND ((ATOM E)
             (COND ((EQ E NIL) NIL)
                   ((EQ E T) T)
                   (T (CDR ((LABEL
                             ASSOC
                             (LAMBDA (E A)
                                     (COND ((NULL A) NIL)
                                           ((EQ E (CAAR A)) (CAR A))
                                           (T (ASSOC E (CDR A))))))
                            E
                            A)))))
            ((ATOM (CAR E))
             (COND ((EQ (CAR E) (QUOTE QUOTE)) (CADR E))
                   ((EQ (CAR E) (QUOTE CAR))
                    (CAR (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE CDR))
                    (CDR (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE CADR))
                    (CADR (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE CADDR))
                    (CADDR (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE CAAR))
                    (CAAR (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE CADAR))
                    (CADAR (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE CADDAR))
                    (CADDAR (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE ATOM))
                    (ATOM (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE NULL))
                    (NULL (EVAL (CADR E) A)))
                   ((EQ (CAR E) (QUOTE CONS))
                    (CONS (EVAL (CADR E) A) (EVAL (CADDR E) A)))
                   ((EQ (CAR E) (QUOTE EQ))
                    (EQ (EVAL (CADR E) A) (EVAL (CADDR E) A)))
                   ((EQ (CAR E) (QUOTE COND))
                    ((LABEL EVCOND
                            (LAMBDA (U A) (COND ((EVAL (CAAR U) A)
                                                 (EVAL (CADAR U)
                                                       A))
                                                (T (EVCOND (CDR U)

                                                           A)))))
                     (CDR E)
                     A))
                   (T (EVAL (CONS (CDR ((LABEL
                                         ASSOC
                                         (LAMBDA (E A)
                                          (COND
                                           ((NULL A) NIL)
                                           ((EQ E (CAAR A))
                                            (CAR A))
                                           (T (ASSOC E
                                                     (CDR A))))))
                                        (CAR E)
                                        A))
                                       (CDR E))
                                  A))))
            ((EQ (CAAR E) (QUOTE LAMBDA))
             (EVAL (CADDAR E)
                   ((LABEL FFAPPEND
                           (LAMBDA (U V)
                                   (COND ((NULL U) V)
                                         (T (CONS (CAR U)
                                                  (FFAPPEND (CDR U)
                                                            V))))))
                    ((LABEL
                      PAIRUP
                      (LAMBDA (U V)
                              (COND ((NULL U) NIL)
                                    (T (CONS (CONS (CAR U) (CAR V))
                                             (PAIRUP (CDR U)
                                                     (CDR V))))))
                     (CADDAR E)
                     ((LABEL
                       EVLIS
                       (LAMBDA (U A)
                               (COND ((NULL U) NIL)
                                     (T (CONS (EVAL (CAR U) A)
                                              (EVLIS (CDR U)
                                                     A))))))
                      (CDR E)
                      A))
                    A)))
            ((EQ (CAAA E) (QUOTE LABEL))
             (EVAL (CONS (CADDAR E) (CDR E))
                   (CONS (CONS (CADAR E) (CAR E)) A)))))
```

Fig. 2. LISP EVAL function.

can evaluate itself evaluating some other expression. It does not know about DEFUNs or any features of LISP not explained in this micro-manual such as functional arguments, property list functions, input-output, or sequential programs.

The function EVAL can serve as an interpreter for LISP, and LISP interpreters are actually made by hand-compiling EVAL into machine language or by cross-compiling it on a machine for which a LISP system already exists.

The definition would have been easier to follow had we defined auxiliary functions separately rather than include them using LABEL. However, we would then have needed property list functions in order to make the EVAL self-applicable. These auxiliary functions are EVLIS which evaluates lists of expressions, EVCOND which evaluates conditional expressions, ASSOC which finds the value associated with a variable in the environment, and PAIRUP which pairs up the corresponding elements of two lists.

EVAL is shown in Fig. 2 on page 712.

PL/I

PL/I was developed in two distinct stages. First the language NPL was conceived by a joint user–IBM committee. IBM alone then developed PL/I by clarifying and refining the rather incomplete specification of NPL. This overview describes the major features of PL/I, and George Radin's paper makes clear the distinctions between these two languages.

The characteristics of PL/I were strongly influenced by two factors: what was known (in 1964–65) about commercial, scientific, real-time, and system programming applications; and the features of FORTRAN, COBOL, and ALGOL which were useful in implementing those applications. The result is a language of great expressive power, which occasionally requires the programmer to manage great complexity.

PL/I supports a broad spectrum of scalar data types, which can be combined in arrays and structures. Each scalar constant or variable is specified within a broad range of numeric, string, label, file, and pointer data types. In addition, each numeric data element has a mode, base, scale, and precision. While the user can specify values for each of these attributes for every variable, the language provides a set of default values. PL/I also supports considerable generality in both the syntax and semantics of arithmetic expressions and provides for automatic conversion between data of widely different types. For example:

```
SALAD: PROCEDURE OPTIONS(MAIN);
        DECLARE APPLES FIXED,
        ORANGES        CHARACTER(40),
        MIXED_FRUIT    FLOAT;

        APPLES = 4.7;
        ORANGES = 6.31;
        MIXED_FRUIT = 2 * APPLES + 2 * ORANGES;

END SALAD;
```

The form of PL/I programs generally follows ALGOL block and procedure structure, with the addition of multiple entry points and possibly separate compilation of procedures. Parameters, including labels and pointers, are passed by reference. Control constructs in PL/I reflect the influence of ALGOL. The following example illustrates many of these features in PL/I.

```
OUTER: PROCEDURE OPTIONS(MAIN);
        DECLARE X (10) FIXED;
                /* 10 ELEMENTS OF DEFAULT BASE AND PRECISION */

        SORT: PROCEDURE (A); /* EXCHANGE SORT */
                DECLARE A (*) FIXED,
                        /* THE SIZE OF A IS INHERITED
                           FROM THE ACTUAL PARAMETER */
                        (I, TEMP) FIXED,
                        FLAG BIT(1) INITIAL ( '1'B );

                DO WHILE (FLAG);
                        FLAG = '0'B;
                        DO I = 1 TO HBOUND(A, 1) -1;

                              IF (A(I) > A(I + 1) THEN DO;
                                    TEMP = A(I);
                                    A(I) = A(I + 1);
                                    A(I + 1) = TEMP;
                                    FLAG = '1'B;
                              END;

                        END;

                END;
        END SORT;

        GET LIST (X); /* READS 10 ITEMS INTO X */
        CALL SORT(X);
        PUT LIST (X);
END OUTER;
```

When corresponding formal and actual parameters are of differing types, PL/I converts the actual parameter and stores it in a local temporary on procedure entry. However, a similar conversion does not take place on exit. Therefore, in this example, if A and X differed in one or more attributes, the contents of X would not have been changed on exit from the procedure SORT.

PL/I also provides exception handling facilities (ON-conditions and ON-units) as well as programmer controlled storage allocations and deallocation. All of these may require changing the run-time context of a program at points other than block and procedure boundaries.

SIMULA

1. Motivation

SIMULA was conceived in the early 1960s by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center [3]. The most prevalent version of the language today is SIMULA 67 [1].

SIMULA was designed for system description and system simulation. A system in this case is a collection of interdependent elements with a common goal [4]. A natural way to simulate a system is to describe the life cycle scenarios of its elements. This approach combines the local attributes of elements and the protocols for interelement communication into a single structure. Elements may exist for the duration of the simulation or be created and destroyed during its course.

2. Response

SIMULA permits the description of life cycle scenarios by means of the *class* concept. This, like the rest of SIMULA, is an extension of the basic concepts of ALGOL 60; in this case, procedures and blocks. SIMULA extends the ALGOL 60 block concept by freeing it from its inherently nested structure and allowing the generation and naming of coexisting block instances. These block instances are known as *objects* and are generated from templates called *class declarations*. The principal extensions which convert ALGOL 60 to SIMULA provide the ability to:

1. Declare a *class;*
2. Generate *objects* of a declared *class;*
3. Name the generated *objects;*
4. Form a hierarchical structure of *class declarations*.

3. Classes and Objects

The form of a class declaration in SIMULA parallels the form of a procedure declaration in ALGOL 60. The run-time consequences, however, are substantially different. In ALGOL 60, a reference to a declared procedure creates an instance of that procedure which only exists for the length of time that control is "inside" the procedure, i.e., executing statements from the procedure body. The ALGOL 60 procedure semantics are typically realized by the familiar mechanism of the run-time stack.

SIMULA, in contrast, permits objects to exist even though control is "outside" of them. Control is transferred to the first statement in the object upon object creation, as in ALGOL 60. Control passes out of the object either upon completion of the last statement or by means of the special statements *detach* and *resume*. Once control has left, the incarnations of the variables of the object persist, and may be accessed by a later execution of the same object or by other objects. This differs from the ALGOL 60 semantics, which define the incarnations of variables local to a block or procedure to be lost when control passes from that block or procedure.

The *detach* statement allows an object to be separated from the block in which it was created, and thereby survive the termination of the creating block. This contrasts with

ALGOL 60 where the created block or procedure instance is always "inside" its creator, and is terminated simultaneously with it. An object which executes a *detach* is said to be detached [1, 4]. The *resume* statement causes the resumption of execution of a named object at the point where that object last ceased execution (by means of a *detach* or *resume*). As a consequence objects have a control relationship like that of coroutines [2]: a single control point that moves from object to object at predetermined points, with no "nesting" or "return" structure imposed.

Objects are generated by means of the *new* statement which gives the *class* (template) name and the actual parameters to be associated with the instance. The name of a newly created object can be stored in a variable for subsequent reference.

4. Object–Object References

Each element of a simulation will have attributes whose values determine the state of the element. Since elements are specified in SIMULA by *class declarations*, their attributes are naturally represented by the variables which appear in the declarations. Objects which are created from these declarations. Objects which are created from these declarations may have occasion to reference the variables of other objects. SIMULA provides two notations to allow such reference. The first, the "dot" notation, is of the form X·Y, and defines a reference to variable Y in the object whose name is X. (Recall that objects can be named upon creation.) The second notation is *inspect* X which allows direct access to the variables of some outside object X by use of the names given to them in the declaration of the class of which X is an instance.

5. Hierarchical Declarations

SIMULA permits the construction of complex objects from simple ones by a process of concatenation of declarations. Consider a *class* declaration C1, which consists of a formal parameter list, a specification part, a set of variable declarations, and a body of statements. This class declaration can be used as a basis for a subsequent class declaration C2, in a manner that causes the formal parameter list given for C2 to be concatenated with that of C1, the specification part of C2 to be concatenated with the specification part of C1, etc. C2 is therefore defined as an extension or increment over the existing C1, and is called a *subclass* of C1. Objects which are generated from subclass declarations are called *compound objects*. The process can continue indefinitely; C3 could be declared by concatenation on C1, etc.

6. Declaration Libraries

SIMULA allows class declarations to be collected in a library and included into a program by a simple reference in that program.

7. The Supplied Simulation System

The *class* concept of SIMULA is quite general and does not directly provide a useful simulation system. Most simulations using SIMULA are built on top of two supplied classes. The first, called SIMSET, provides a doubly linked circular list processing facil-

ity. The second, called SIMULATION is built using SIMSET, and provides support for the element life cycle scenarios of a given model. This supplied simulation system is quite compact, requiring less than twenty pages for its description [4] and can be very simply incorporated into a specific model.

8. The Process View of Simulation

The life cycle scenarios supported by the supplied class SIMULATION are called *processes*, and consequently SIMULA is said to support the *process view* of simulation. Processes proceed logically in parallel (by means of multiplexing over a single processor), and the objects which comprise them may be executed piecemeal (by the coroutine structure provided by *detach/resume*) or as single procedures. The model builder can combine piecemeal and single-procedure execution as he sees fit, and thereby achieve a natural representation of the system being simulated. The expressive power of SIMULA can be described also by considering the universal structure of simulations: a two-level hierarchy, consisting of a control program which manages the simulation clock and selects lower-level modules for execution. In SIMULA, these lower level modules can be processes, SIMSCRIPT event routines, GPSS block handlers, CSL activity modules, or the modules of continuous system simulations (see [4]). In this view SIMULA includes all of the aspects of the named systems and approaches as well as one distinctively its own.

9. Concluding Observations

The supplied simulation system is described completely in SIMULA, and therefore may be modified or enhanced as necessary to support the needs of a given model. Recent additions to SIMULA permit modification of the supplied system without loss of any of the compile and run time features the supplied system provides to uncover logical inconsistencies in the model. The supplied simulation system of SIMULA has been extended to form, for example, the database and operating system language OASIS [10] and the information system of SOL [5].

```
SIMULA begin integer uninfected;
  activity sick person;
  begin integer day;  Boolean symtoms;  set environment;
    uninfected := uninfected-1;  symptoms := false;
    hold(incubation);  symptoms := true;
    for day := 1 step 1 until length do
    begin if draw(probtreat[day], u1)
      then activate new treatment (current);
      infect(Poisson(contacts, u2), environment);
      hold(1) end
    end sick person;
    procedure infect(n, S);  value n;  integer n;  set S;
  begin integer i;
    for i := 1 step 1 until n do
    if draw(prinf×uninfected/population, u3) then
```

```
    begin include (new sick person, S);
      activate last (S) end
  end infect;
  activity treatment(patient);   element patient;
  begin element X;
    extract patient when sick person do
    if symptoms then
    begin terminate(patient);
      for X := first(environment) while exist(X) do
      activate new treatment(X) end
    else if draw(probmass, u4) then terminate (patient)
  end treatment;
  unifected := population;
  activate new sick person;   hold(simperiod)
end SIMULA
```

REFERENCES

1. Birtwistle, Graham, M., Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard, SIMULA *Begin*, student-litteratur, 1973.
2. Conway, Melvin E., Design of a Separable Transition-Diagram Compiler. *CACM*, 6(7) (1963) July.
3. Dahl, Ole-Johan, and K. Nygaard, SIMULA—An Algol based Simulation Language. *CACM*, 9 (9): 671–681 (1966).
4. Franta, W. R., *The Process View of Simulation*., Amsterdam: Elsevier/North-Holland, 1977.
5. Sol, H. G., *SIMULA Software for the Development of Information Systems*, Technical Report, Department of Management Science and Business Economics, State University of Groningen, Groningen, the Netherlands.
6. Unger, Brian, and Donald Bibulock, James Gosling, and Doug Robinson, *OASIS Reference Manual*, TR No. 7711514, Dept. Computer Science, University of Calgary, February 1977.

SNOBOL

The SNOBOL programming languages introduced or popularized most of the concepts of character string processing. The original SNOBOL had just one data type: a string of characters. It could appear in a program as a literal delimited by quotation marks, or as a value assigned to a variable:

```
PET = "MY CALICO CAT"
```

Concatenation, a basic string operation, forms a new string from two or more strings placed end to end:

```
MYPET = "HERE IS" PET "."
```

concatenates the string "HERE IS", the value in the variable PET, and the string "." to form the string "HERE IS MY CALICO CAT." and assigns it to the variable MYPET.

Another fundamental operation is the pattern match. It examines the contents of a string as specified by a sequence of pattern elements. The elements may be strings (variables or literals) or special patterns delimited by asterisks. The original special patterns matched a

fixed length string, an arbitrary string (between two other elements), or a string "balanced with respect to parentheses" (for algebraic expression processing).

A program is a sequence of statements. Each statement may contain up to three parts; all are optional. The first part of a statement is a label, allowing branching to that statement. The second part is the rule, specifying the action to be taken. The final part, the go to, controls flow to the next program statement. The sample SNOBOL program shown below demonstrates several of the language features. It reads lines of text, changes the vowels to asterisks, and prints the result.

```
[1] NEXTL SYS .READ *LINE/"80"*     /F(END)
[2]       VOWEL = "A,E,I,O,U,"
[3] NEXTV VOWEL *V* "," =           /F(LIST)
[4] VSTAR LINE V = "*"              /S(VSTAR)F(NEXTV)
[5] LIST  SYS .PRINT "LINE=" LINE /(NEXTL)
[6] END   NEXTL
```

The rule of a statement can specify assignment, pattern matching, or pattern matching with replacement. In an assignment rule (as shown in line 2 of the sample), an expression to the right of an equals sign is evaluated and assigned to the variable name on the left of the sign. The expression can be a variable or literal, a string concatenation expression, or a simple arithmetic expression. (Numeric literals are represented as strings of digit characters: "80".) In a pattern match the string to be examined is named, followed by the pattern definition (shown in line 1, combined with an input request). Execution of a match will either succeed or fail; this outcome is tested by the go to. The programmer may optionally specify a replacement for a matched substring in a match-and-replace rule by placing an equals sign after the pattern definition and specifying a replacement string expression to its right (line 4). The resulting string is expanded or contracted as required.

The go to of a statement can specify either an unconditional transfer to another statement after the statement is executed or can test the success or failure of the operation. The go to field is introduced by a slash character. For an unconditional go to (line 5), the destination label appears in parentheses as the remainder of the field. Conditional branching is indicated by preceding the open parenthesis by an S for success or an F for failure. Both may be specified, in either order (line 4). If no go to field appears or the condition which occurs was not specified, execution continues with the next statement.

As a detailed description of the actions in the sample, line 1 reads a card image from the system input file and assigns the 80 characters to the variable LINE. If there is no input data, the program branches to END to terminate execution. Line 2 assigns the string of vowels and commas to the variable VOWEL. Line 3, labeled NEXTV, pattern matches on the string VOWEL, assigns the letter to the variable V and replaces the letter and comma by a null (zero length) string. When VOWEL is null, the statement fails, branching to LIST (line 5). Line 4 matches and replaces the first occurrence of the vowel in V with an asterisk. If the match fails, the program goes to NEXTV to get the next vowel. Line 5 (LIST) prints the line, preceded by "LINE=", and branches to NEXTL in line 1 to read the next data line. The label END on line 6 indicates it is the last line of the program which begins execution at statement NEXTL.

SNOBOL2 and SNOBOL3 did not significantly change the basic concepts or structure. They did provide additional operations, such as numeric comparison functions: .LE(N,"10")

SNOBOL4, the latest version, expanded the basic concepts to a more general data manipulation language. The general form of the language remained the same, but with major changes in the syntax rules for patterns. Patterns and pattern structures were expanded and became new assignable data types. Both integer and real numeric types and a full set of their operations were added. Data structuring was introduced, with predefined structure data types ARRAY and TABLE. An ARRAY is a collection of data items, which need not be of uniform data type, referenced by numeric subscripts. A TABLE is an associative array, with each element referenced with a unique associated value which is not necessarily an integer PET["DOG"] The programmer may define data structures as additional data types.

This example of a SNOBOL4 program performs the same functions as the previous SNOBOL program, but demonstrates use of the pattern data type:

```
[1]        VOWEL = ANY("AEIOU")
[2] NEXTL LINE = INPUT              :F(END)
[3] VSTAR LINE VOWEL = "*"          :S(VSTAR)
[4]        OUTPUT = "LINE=" LINE  :(NEXTL)
[5] END
```

Line 1 defines the pattern VOWEL to be any of the characters listed in the string. Line 2 reads the next input line and assigns it to the variable LINE, with a branch to END if data has been exhausted. Line 3 examines each character, replacing the first vowel by an asterisk. The program loops on this line until all have been replaced. Line 4 writes the line to the output file and goes back for more input. Line 5, with the label END, indicates the end of the program.

A single-statement SNOBOL4 program which produces the same result is:

```
[1] LOOP OUTPUT = "LINE=" REPLACE(INPUT,
    +        "AEIOU","******")  :S(LOOP)
[2] END
```

The REPLACE function provides the desired action with only a single pass over the input line.

REFERENCES

1. Griswold, Ralph E. *The SNOBOL4 Programming Language*, 2nd edition. Englewood Cliffs, New Jersey: Prentice-Hall, 1971.
2. Griswold, Ralph E., and Griswold, Madge T. *A SNOBOL4 Primer*. Englewood Cliffs, New Jersey: Prentice-Hall, 1974.
3. Gimpel, James F. *Algorithms in SNOBOL4*. New York: Wiley, 1976.

ACKNOWLEDGMENTS

The following people prepared language summaries: Charles L. Baker (JOSS), Thomas Cheatham (JOVIAL), William R. Franta (SIMULA), Bernard A. Galler (FORTRAN), David Gries (ALGOL), Kenneth Iverson (APL), Ted G. Lewis (BASIC), John McCarthy (LISP), Julian Reitman (GPSS), Robert F. Rosin (PL/I), Douglas T. Ross (APT), Michael Shapiro (SNOBOL). The COBOL summary was anonymous.

C

LANGUAGE HISTORY CHART

First letter of each name has been aligned with the approximate date on which work began.

THIS TYPE STYLE indicates languages of major importance, because of their wide usage or technical significance.

THIS TYPE STYLE indicates languages of moderate importance.

THIS TYPE STYLE is used for all other languages.

Parentheses were used to indicate alternate names, or the later addition of the sequence number "1."

═══ indicates that the second language is a direct extension of the first

─── indicates that the second language is an approximate extension of the first, i.e., very similar to the first, but not completely upward compatible

- - - - indicates strong influence; sometimes the second language is "like, or in the style of" the first

· · · · indicates an approximate subset

Each of the following marks is associated with the language above or to its left:

● indicates preliminary or informal specifications or manual

■ indicates a public manual, or formal publication via technical paper, or public presentation

▲ release for usage outside development group

General Comments

This chart represents only the personal opinions of the author as far as value judgments are involved, and the author's best estimate in many cases as far as dates are involved. The indications of the start of the work are the most questionable.

The information for languages in 1971 is based solely on those listed in "Roster of Programming Languages–1971," Computers and Automation, Vol. 20, No. 6B (June 1971), pp. 6-13.

In most cases, dialects with differing names have been omitted. This has the unfortunate effect of appearing to minimize the importance of some languages which spawned numerous versions under differing names (e.g., JOSS).

Languages for specialized application areas (e.g., simulation, machine tool control, civil engineering, systems programming) have not been included because of space considerations. This explains the absence of such obvious languages as APT, GPSS, SIMSCRIPT, COGO, BLISS.

Acknowledgment: The idea for such a chart in such a format came from the one by Christopher J. Shaw entitled "Milestones in Computer Programming" and included with the [ACM Los Angeles Chapter] SIGPLAN notices, February 1965.

"Programming Languages: History and Future"
by Jean E. Sammet
Communications of the ACM, Vol. 15, July 1972