AD-760 547

DCPL.-A DISTRIBUTED CONTROL PROGRAMMING
LANGUAGE

Denis D. Seror

Utah University

Prepared for:

Rome Air Development Center
Defense Advanced Research Projects Agency

December 1970

PII Redacted

DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST
QUALITY AVAILABLE.

COPY FURNISHED CONTAINED
A SIGNIFICANT NUMBER OF
PAGES WHICH DO NOT
REPRODUCE LEGIBLY.

AD 760547

DCPL - A DISTRIBUTED CONTROL PROGRAMMING LANGUAGE

University of Utah

D D C

MAY 25 1973

Approved for public release;
distribution unlimited.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
University of Utah Salt Lake City, Utah 84112	Unclassified
	2b. GROUP

3 REPORT TITLE

DCPL - A DISTRIBUTED CONTROL PROGRAMMING LANGUAGE (U)

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

5 AUTHOR(S) (First name, middle initial, last name)

Denis D. Seror

6 REPORT DATE	7a. TOTAL NO OF PAGES	7b. NO OF REFS
December 1970		
8a CONTRACT OR GRANT NO	9a ORIGINATOR'S REPORT NUMBER(S)	
Contract F30602-70-C-0300		
b. PROJECT NO		
c.	9b OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.	RADC-TR-73-110	

10 DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11 SUPPLEMENTARY NOTES Monitored by	12 SPONSORING MILITARY ACTIVITY
Murray Kesselman RADC (ISCE), GAFB, NY 13441 AC 315 330-2018	Defense Advanced Research Projects Agency Wash DC 20301

13 ABSTRACT

In this report, a computation is considered a system of asynchronously cooperating "independent" programs (coroutines) linked by paths of information along which messages are sent.

A programming language called DCPL, a Distributed Control Programming Language, in which such computations may be expressed, and which may be considered as a system-oriented programming language, is presented. A tree-structured representation and a very dynamic binding give to a DCPL program the flexibility of the highest level programming languages together with the potential of concurrency of the asynchronous computational structures.

The locality of references which is exhibited in any DCPL program allows a new computer organization using sequential storage devices with large transfer rate instead of random-access storage devices with relatively low transfer rate. Moreover, the computer is expected to achieve a large throughput by taking the parallelism into account.

1a

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Graphics						
Computer Graphics						

14

DCPL - A DISTRIBUTED CONTROL PROGRAMMING LANGUAGE

Denis D. Seror

PUBLICATION REVIEW

This technical report has been reviewed and is approved

MURRAY KESSELMAN

RADC Project Engineer

id

ACKNOWLEDGMENTS

TABLE OF CONTENTS

iv

ABSTRACT*

In this thesis, a computation is considered a system of asynchronously cooperating "independent" programs (coroutines) linked by paths of information along which messages are sent.

A programming language called DCPL, a Distributed Control Programming Language, in which such computations may be expressed, and which may be considered as a system-oriented programming language, is presented. A tree-structured representation and a very dynamic binding give to a DCPL program the flexibility of the highest level programming languages together with the potential of concurrency of the asynchronous computational structures.

The locality of references which is exhibited in any DCPL program allows a new computer organization using sequential storage devices with large transfer rate instead of random-access storage devices with relatively low transfer rate. Moreover, the computer is expected to achieve a large throughput by taking the parallelism into account.

CHAPTER 1

INTRODUCTION

1.1   Introduction.

In this thesis, a computation is considered a system of
asynchronously cooperating "independent" programs (coroutines)
linked by paths of information along which messages are sent.

A programming language called DCPL, a Distributed Control
Programming Language, in which such computations may be expressed,
and which may be considered as a system-oriented programming
language, is presented. A tree structured representation and
a very dynamic binding give to a DCPL program the flexibility
of the highest level programming languages together with the
potential of concurrency of the asynchronous computational
structures.

The locality of references which is exhibited in any
DCPL program allows a new computer organization using (inexpensive)
sequential storage devices with large transfer rate instead of
(expensive) random access storage devices with relatively low
transfer rate. Moreover, the computer is expected to achieve
large throughput by taking the parallelism into account.

1.2   Variables in computing processes.

In his paper "On certain basic concepts of programming languages" [27] Niklaus Wirth wrote:

> "The elementary concepts of computing processes are:
>
> o There exist certain quantities, to be called "values" and elementary classes or types (possibly only one) of values among whose elements given elementary relationships hold. These relationships or mappings are represented in a computer by its operations which generate a ne' value (called result) which has the specified relationship to the given value(s) (called operands).
>
> o There exist cells (usually called "variables") which are able to contain a value, and which have a name. That name serves to refer to the contained value.
>
> o There exists an operator for the assignment of a new value to a cell.
>
> ... "

These concepts are widely accepted today, and they underlie any actual implementation of a conventional programming language.

In DCPL we support the first part of the quotation: there is a universe of values structured in classes or types, and mappings from some classes to some possibly different classes which are actualized by operations. There may be, for instance, in our universe, integer and logical values forming the classes I and L, and the operations:

operations	operators	mappings
addition	+	$I \times I \rightarrow I$
disjunction	v	$L \times L \rightarrow L$
negation	~	$L \rightarrow L$
equality	=	$I \times I \rightarrow L$
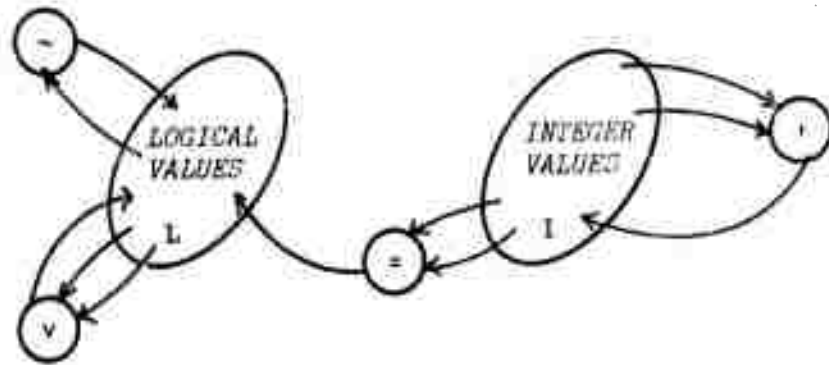
Fig. I-1

In fact, it is not our intention to impose any restriction on our universe. Consequently, we are leaving the list of types and operations open-ended. For this reason, and since syntax has received a rather speedy treatment, the emphasis being placed upon semantics, it would be proper to consider DCPL as describing a family of languages rather than defining completely one specific language.

DCPL, however, does not support the second part of the quotation: one variable in DCPL has no meaning by itself; a system of mutually bound variables defines communication paths.

I.2.1 An example.

Let us consider the evaluation of the following simple expression:

(1) $(2 + 3 \rightarrow x;\ (6-1) \rightarrow y;\ x + y \rightarrow x\ ;\ y + 1 \rightarrow y\ ;(2 \times y) - x\ )$

in a conventional programming language (fig. I-2):
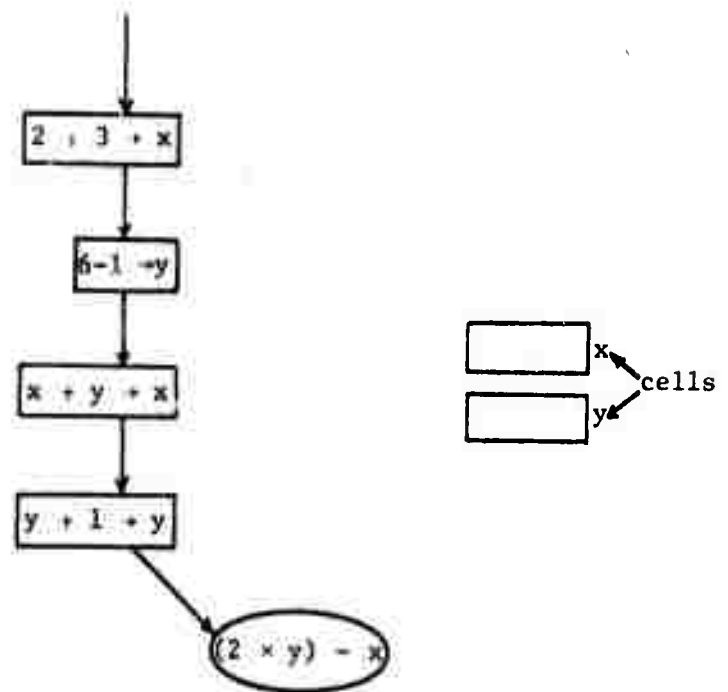
Fig. I-2

- there are two cells x and y ;

- the expression is viewed as a sequence of statements. The execution of any statement is a simple process involving some cells: whenever the statement $x + y \to x$, for instance, is executed, the values contained in x and y are retrieved, and their sum is then stored in x;

- the statements are executed serially one after the other. The last item is not a statement; its value is considered to be the value of the expression.

The same expression may be interpreted in a quite different way in DCPL. In order to study gradually the notions involved, we start by interpreting a much simpler expression without variables:

(2) $(2 + 3 + 2) + 2 + 1$

Association having to be done on the right, the expression may be represented as the tree of fig. I-5a. We may view each node as a simple automaton, and each edge as a channel of information. Whenever a node represents an integer, it sends up spontaneously along the edge its own value, and then vanishes (fig. I-3).

Whenever a node represents an addition, it waits until it receives a value from both the right and the left side; then it adds the two values and sends the result up along the channel, and vanishes (fig. I-4). The evaluation of   expression (2) is displayed in fig. I-5.

Let us consider again expression (1). The expression may be represented as a syntax tree (fig. I-6).

6
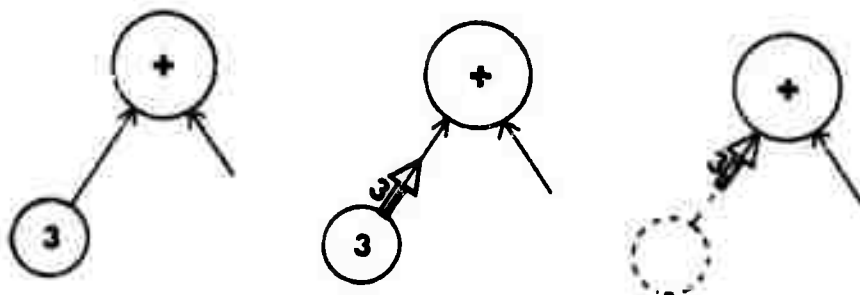
Fig. I-3
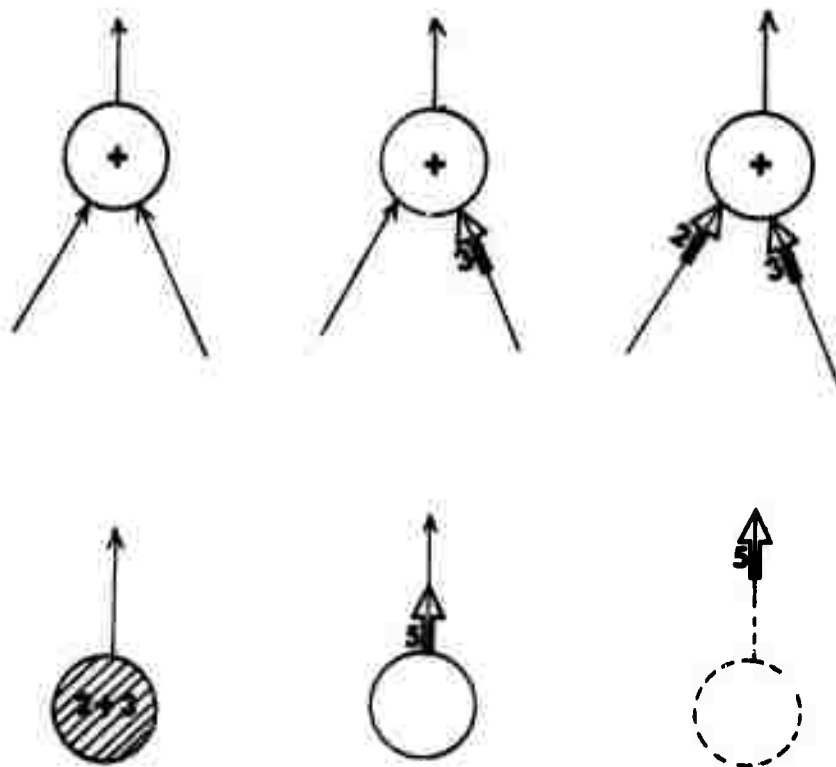
Fig. I-4
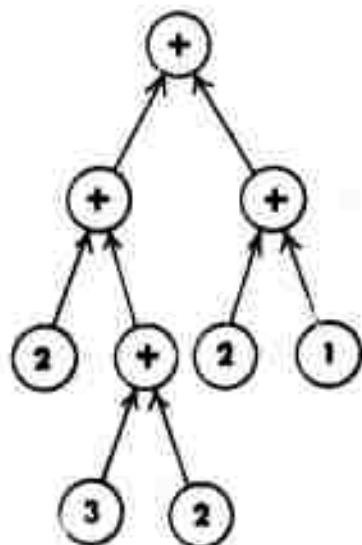
Fig. I-5a
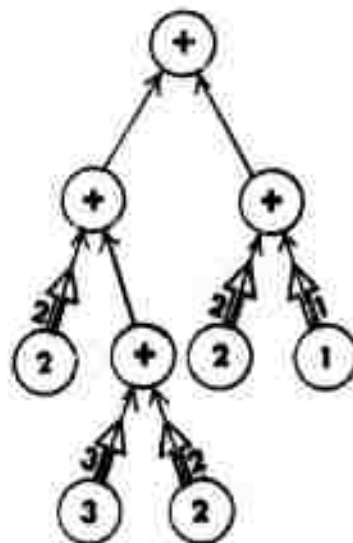
Fig. I-5b
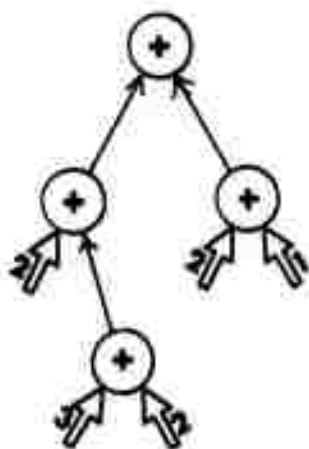
Fig. I-5c
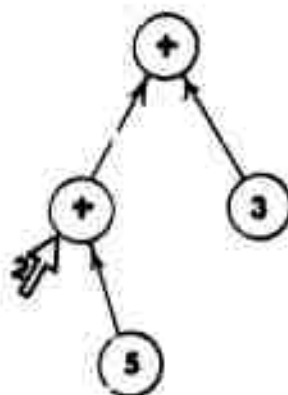
Fig. I-5d
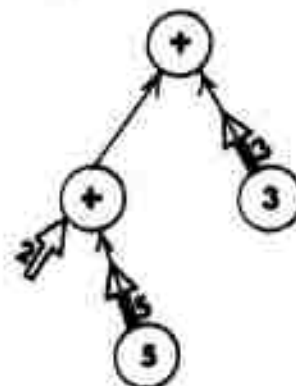
Fig. I-5e
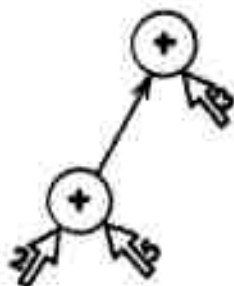
Fig. I-5f
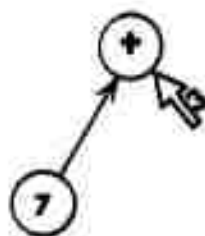
Fig. I-5g

Fig. I-5h

Fig. I-5i

Fig. I-5j
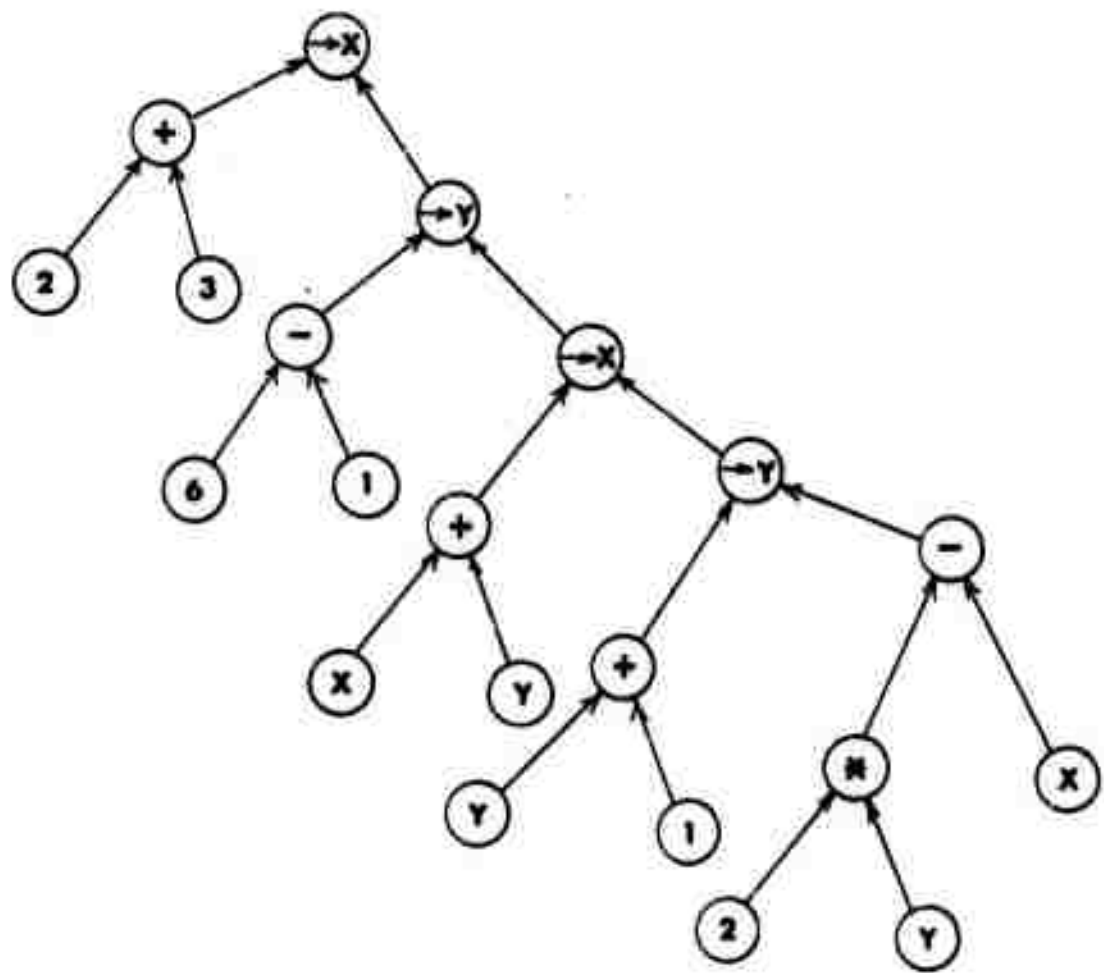
Fig. I-6

Each node is here again considered as being an automaton which has an address in some address space. Whenever a node A knows the address of another node B, the former may send a message to B (fig. I-7). Moreover any node may send a message up the tree; any node receiving such a message may either pick it up or pass it along upward.

The nodes representing values or operations have the same behavior as before. Whenever a node represents some variable x,y, etc..., it sends spontaneously up the tree its name and its address. Such a message can be picked up by a binder (→x , →y are binders) containing the same variable name only if the message reaches the binder by the right. The binder possesses then the address of the node which has originated the message (fig. I-8).

The various nodes which act spontaneously are completely independent from one another. However, in order to have figures

---

Note about the syntax: → is considered as being a triadic operator (operator of degree 3),

representing in fact
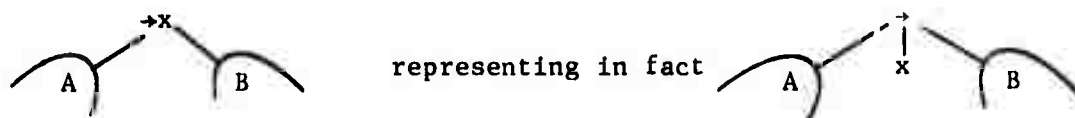
Whenever an operator Op is triadic, A Op B;C represents:
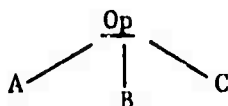
The association being done on the left, A Op$_1$ B;C Op$_2$ D;E represents:

Fig. I-7

address of the
originating node X
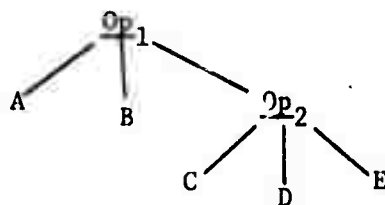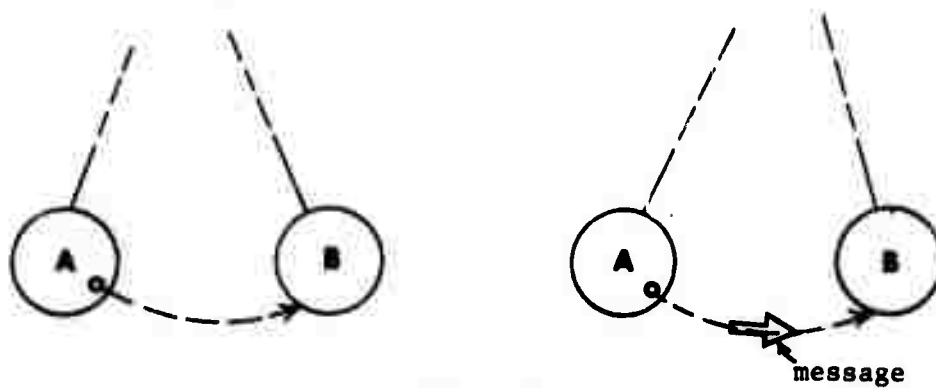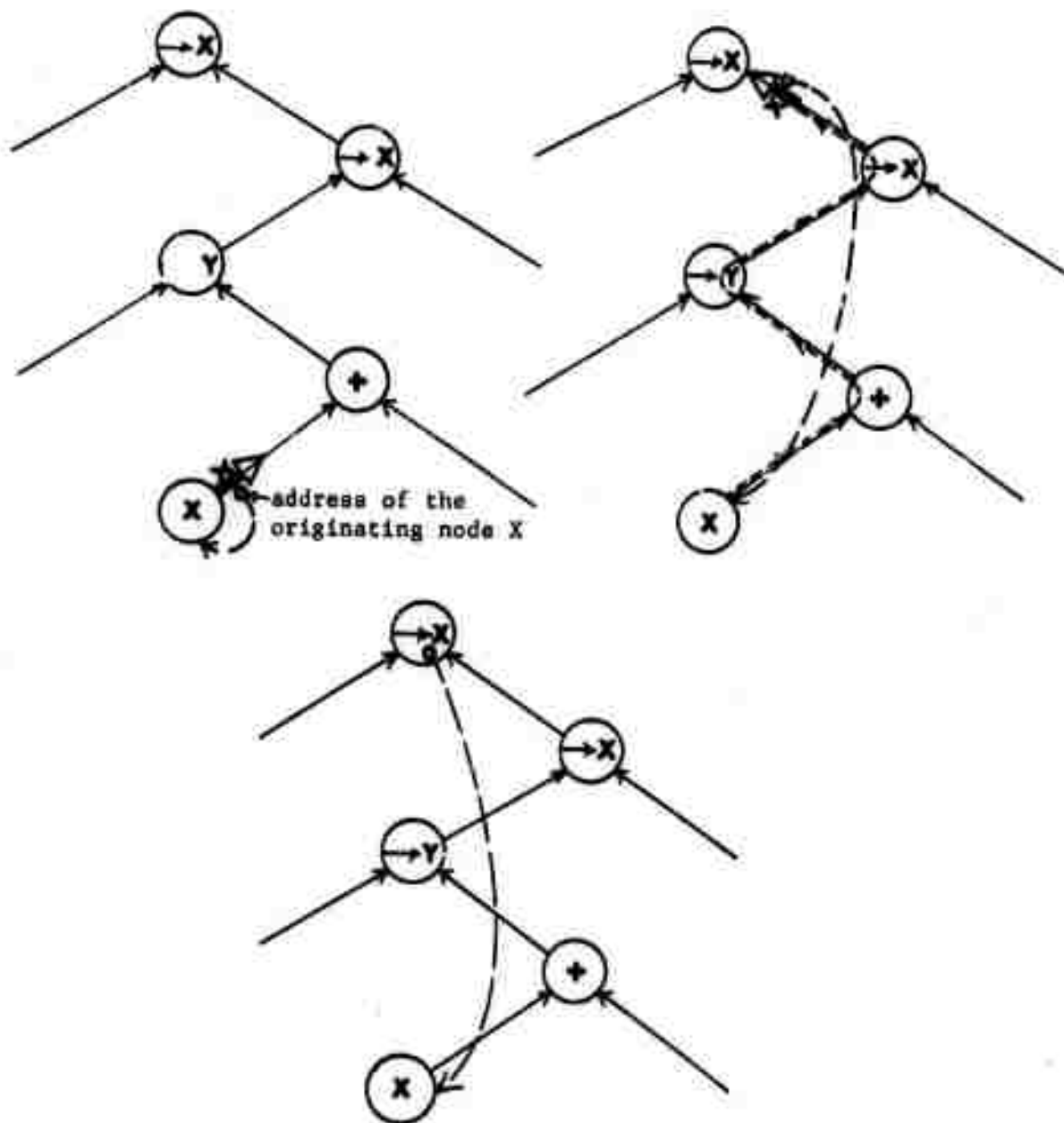
Fig. I-8

more readable and to display the binding in a more visible way, the binding operations are performed first.

Whenever a binder has received a value on its left, and addresses of variable nodes on its right, it sends the value received to each variable node whose address it has received (fig. I-9).

Whenever a binder receives a value from its right, it passes this value up along the tree and vanishes (fig. I-10).

The description of the evaluation is given in fig. I-11.

I.2.2 Variables as defining paths of information.

If we view a computation as a sequence of statements, variables are to be considered as denoting cells in which some results may be stored for subsequent use. Conversely if beforehand variables are considered as denoting cells, it is necessary to be assured that no attempt will be made to use a value before it has been produced, or to overwrite a value which is still to be used. As a result some sequencing has to be done, sequencing whose viscosity will decrease the amount of possible parallelism.

If we view a computation as occurring in space and in time, we will use variables whenever we want some information to be transferred from one place to some other place(s). Whenever a variable is free, it has no meaning by itself. A system of mutually bound variables is used as a symbolic device defining paths of information along which information flows toward computation. Whenever the needed information reaches an operator node, the simple process it represents would be activated asynchronously. Its completion may result in sending some information to some other nodes.
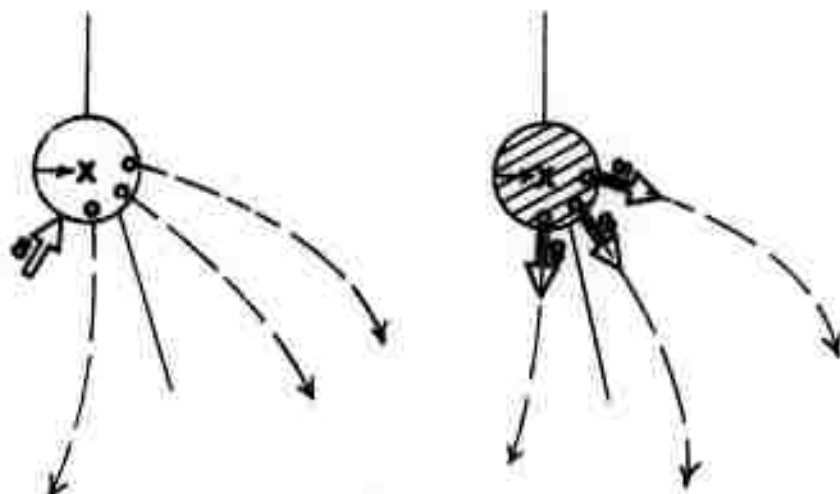
Fig. I-9

Fig. I-10
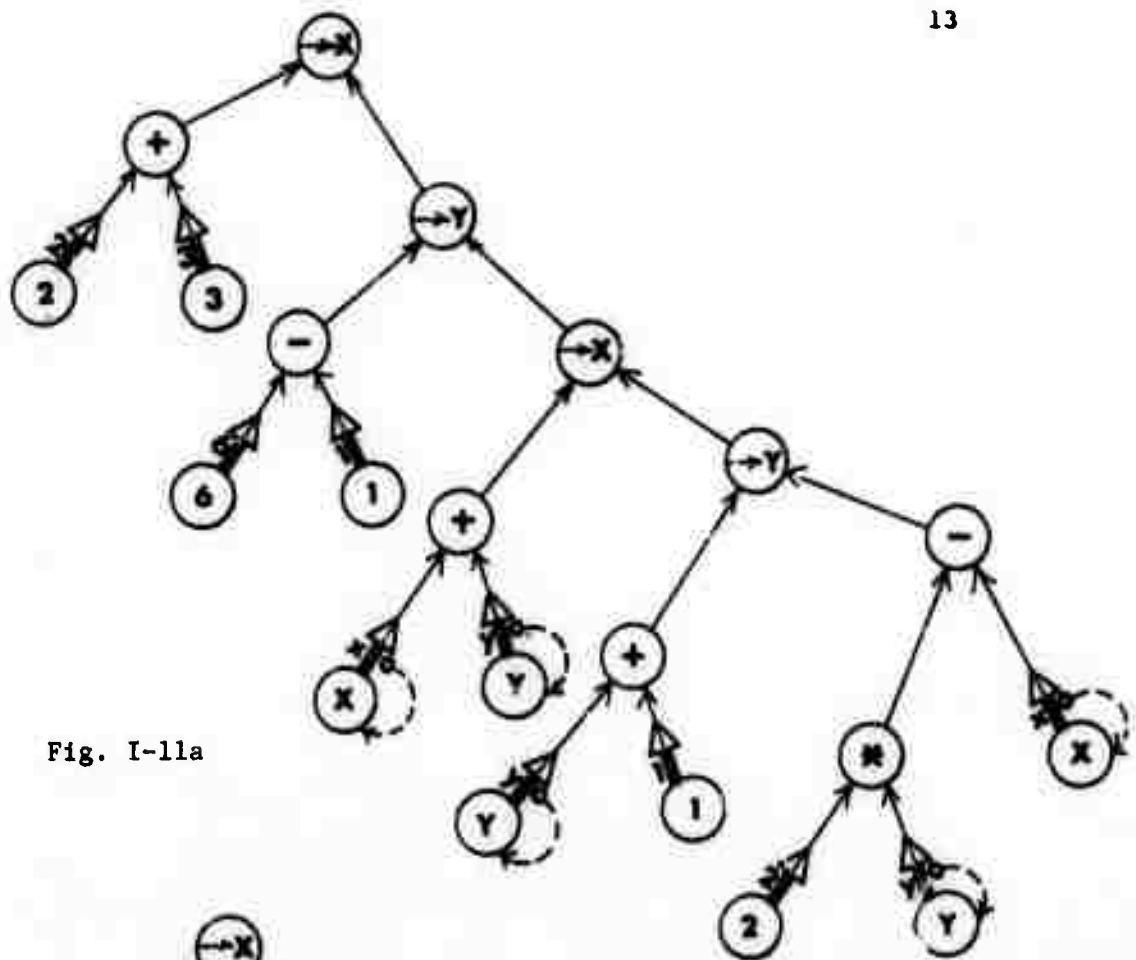
Fig. I-11a
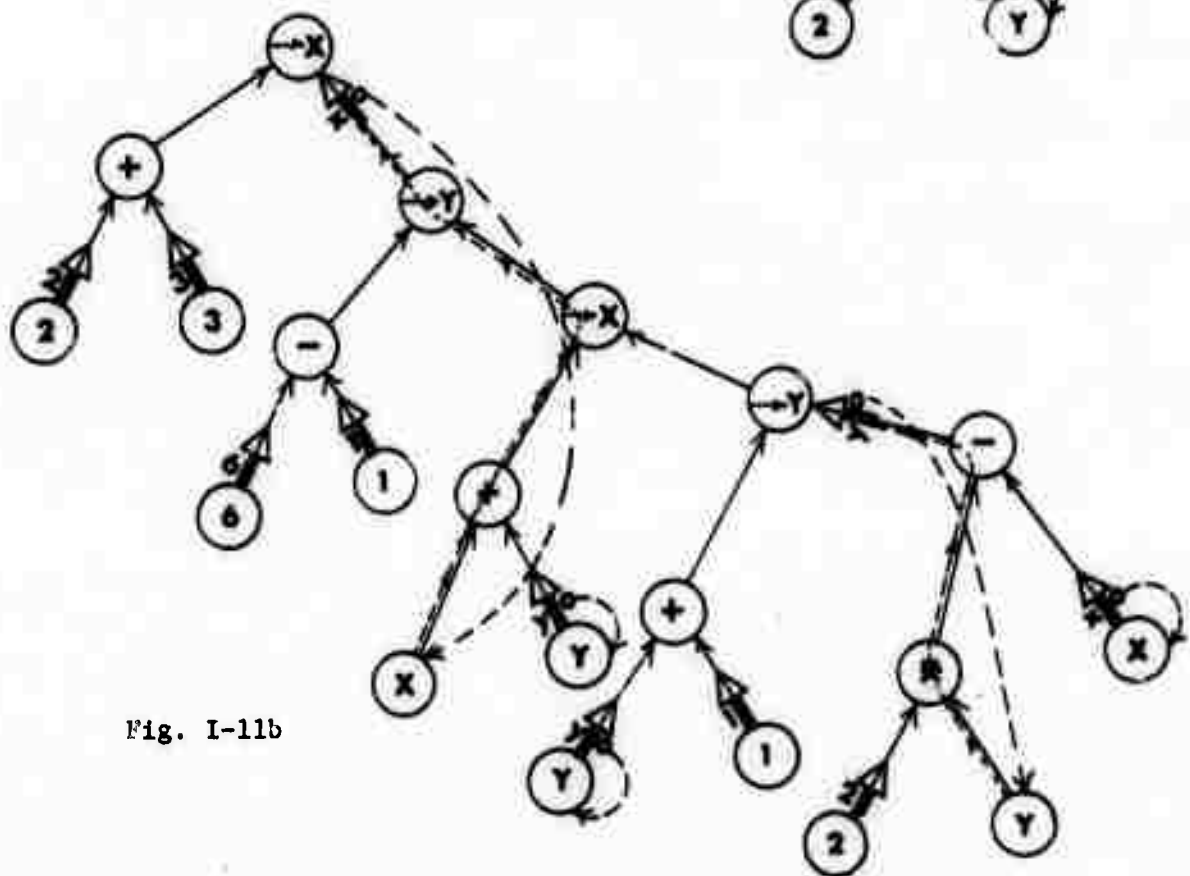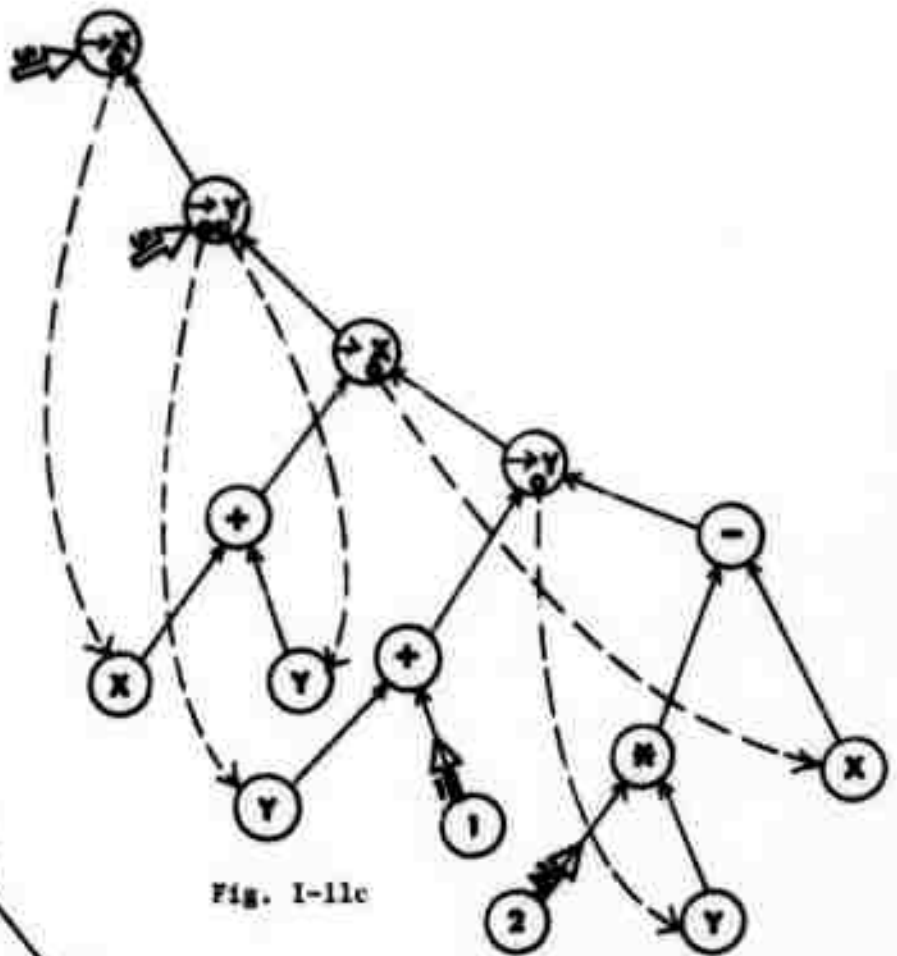
Fig. I-11b
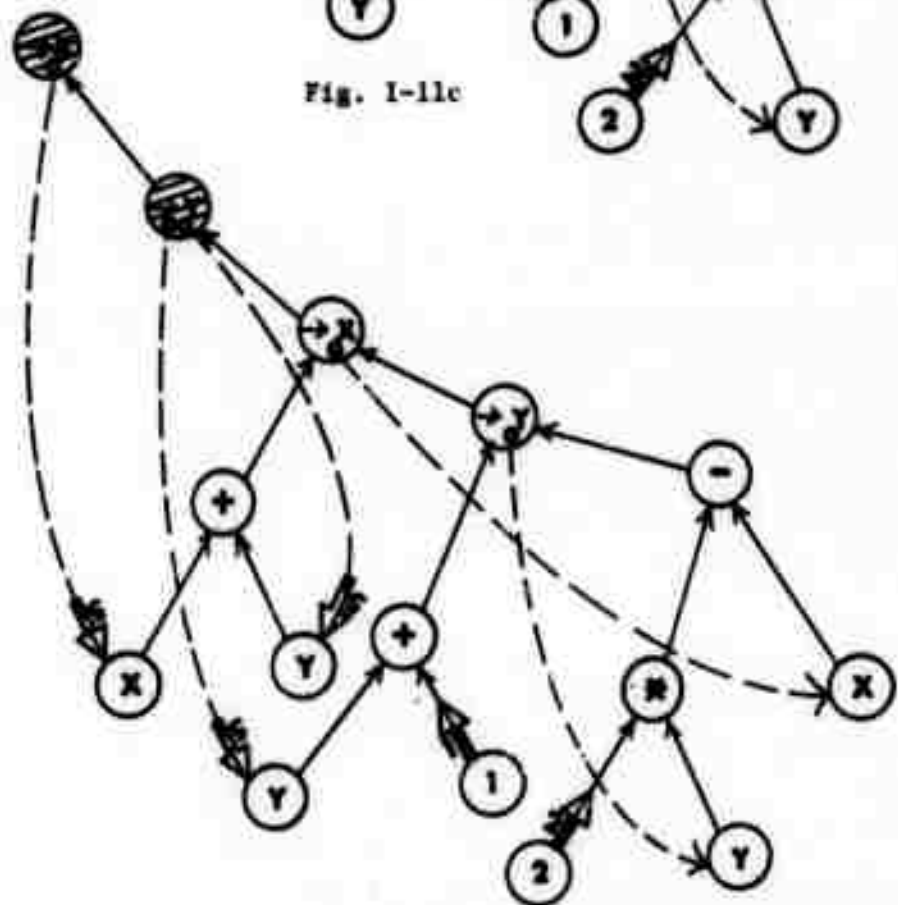
Fig. I-11c

Fig. I-11d
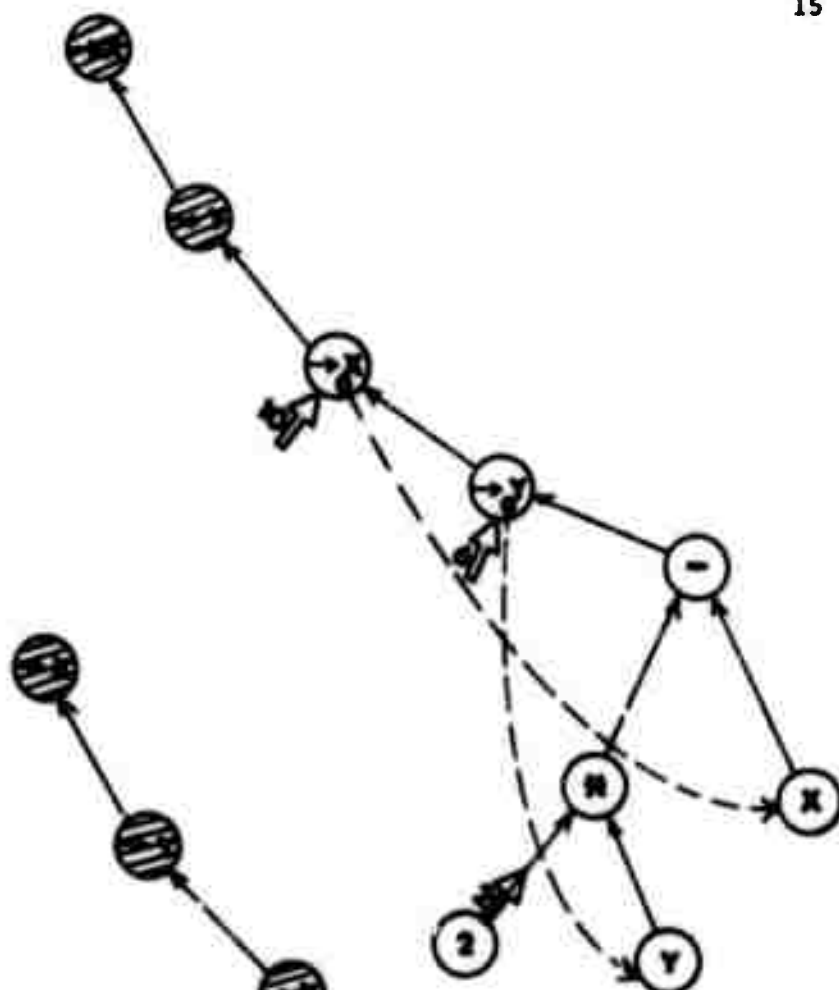
Fig 1-11e
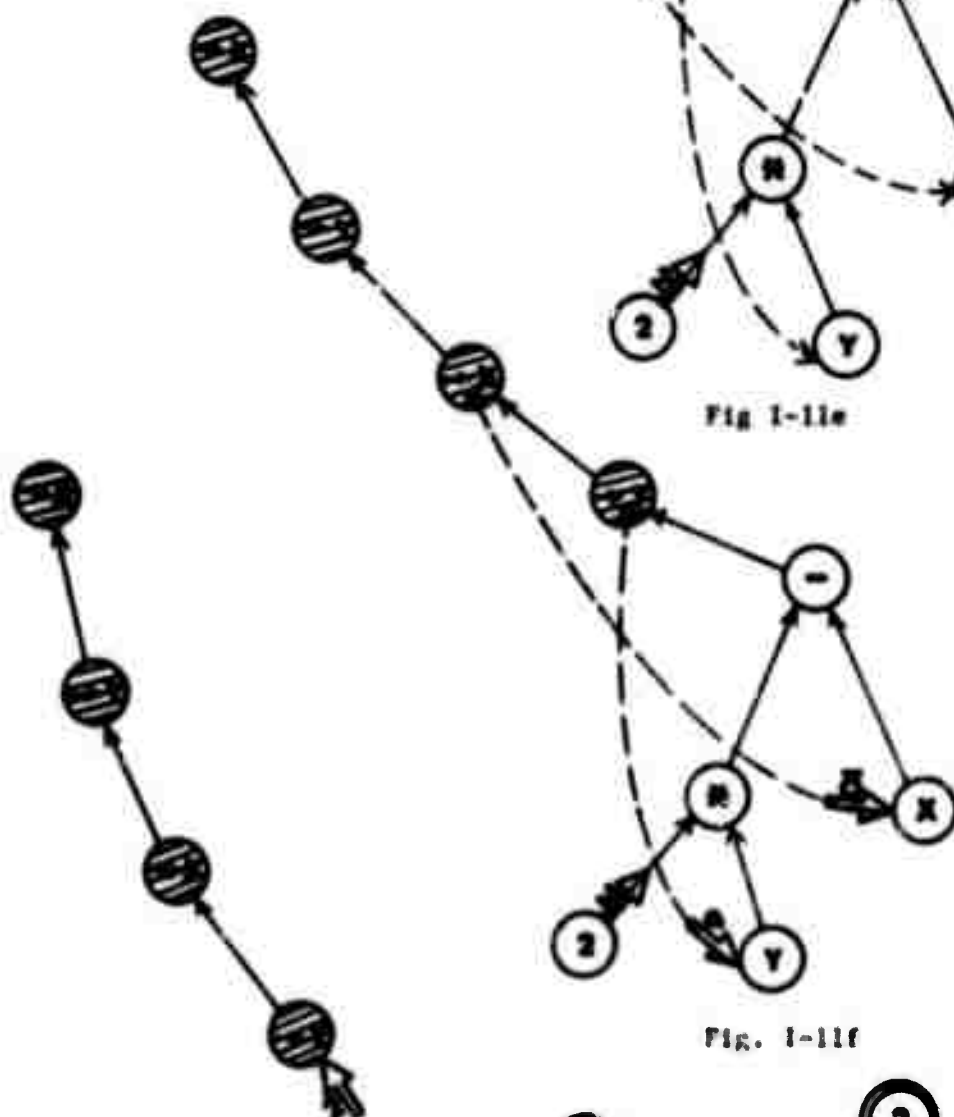
Fig. 1-11f
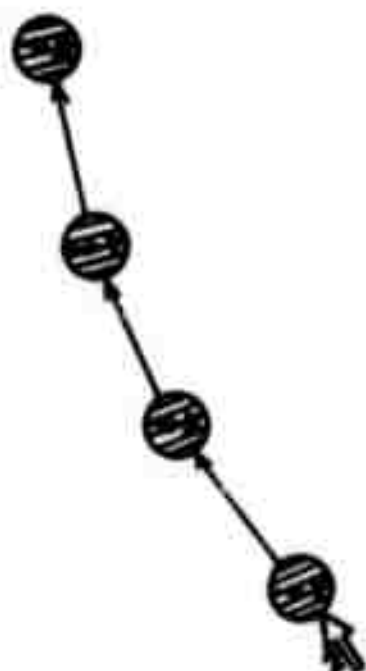
Fig. 1-11g   Fig. 1-11h   Fig. 1-11i

In DCPL a computation may be regarded as an object composed
of small automata which react to one another. Since a variable may
receive a computation as value, representing for instance a proce-
dural argument, the object representing a computation may expand
and shrink with a behavior which may lead us to think of Von
Neumann's self reproducing automata [24]. We may notice however
that our structures are not to be implemented in some cellular
space but programmed on a storage device.

The binding of variables superimposes to the tree structure
a graph structure similar to a program graph. Fig. I-12, for instance
displays the graph of our previous example. Such a program graph
accounts for all the possible parallelism (or preferably concurrency)
which may occur in the computation.

I.3   DCPL as a programming language.

As a programming language, DCPL has much in common with
languages emphasizing expressions (rather than statements) and
having to some extent the lambda calculus as background machine
(McCarthy [18]; Landin [12,13,14]; A. Evans [7]). Any computation
is a structured object whose evaluation produces a value. Moreover
we can have procedural arguments: a procedure may be constructed
in some place, produced as a value and sent to some other places
where copies of the procedure are created (implemented in space).

However, DCPL presents many peculiarities: a computation,
viewed as an object, may produce explicit side-effects on the
environment in which the computation is embedded. Together with
a binder 'lambda' which binds the variables which are to receive
an argument from the environment (as in some extent "value" in
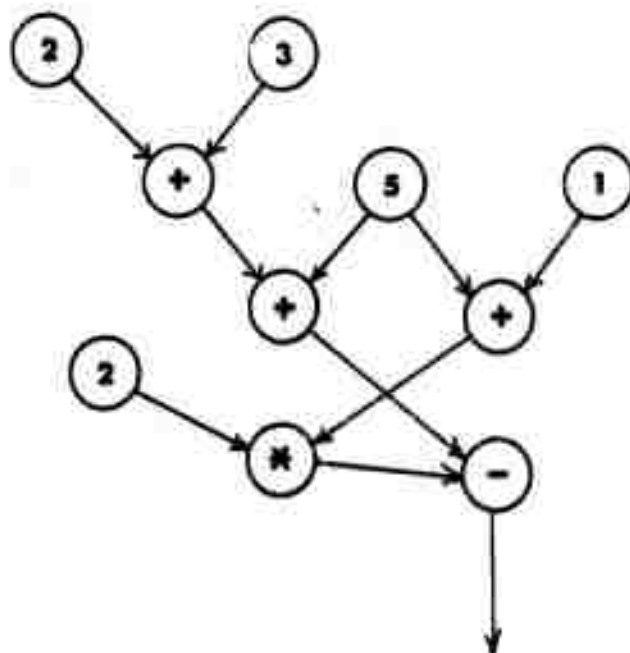
Fig. 1-12

Algol 60), there is a binder 'mu' which binds the variables which
are to send an argument to the environment. 'Mu' (fig. I-13) , as a binder,
is in some way similar to "result" in Algol W [26] with this impor-
tant difference: an argument may be sent to the environment before
the computation has been completed or even while the computation
is actively worked out; as a result a computation may ask to the
environment how to pursue the process or, some special conditions
having occurred, if the computation is not to be cancelled , etc ...

As it has been already mentioned, DCPL is a programming
language implicitly displaying parallelism to a large extent.
Indeed, there is no need for special devices such as fork, join,
parbegin/parend etc ... which determine parallelism explicitly.

DCPL is a system-oriented programming language, this aspect
being discussed in the next section.

I.4   DCPL as a system-oriented programming language.

"System" is regarded here as denoting a group of interacting
procedures constituting a collective entity. A sophisticated
industrial organization, an administration, a hospital, are systems:
a number of departments are services interacting to one another.
If a computer is to be used integrated in such an environment, it
is likely it should look like an information network; moreover
programming should reflect such an organization.

In DCPL we are able to write asynchronously cooperating
"independent" programs (coroutines) linked by paths of information
along which messages are sent, and to write them recursively, i.e.
any one of the previous programs may itself be a construct of
cooperating independent subprograms (fig. I-14). It is possible
to embed in DCPL sequential programs and to master their synchro-

Fig. I-13
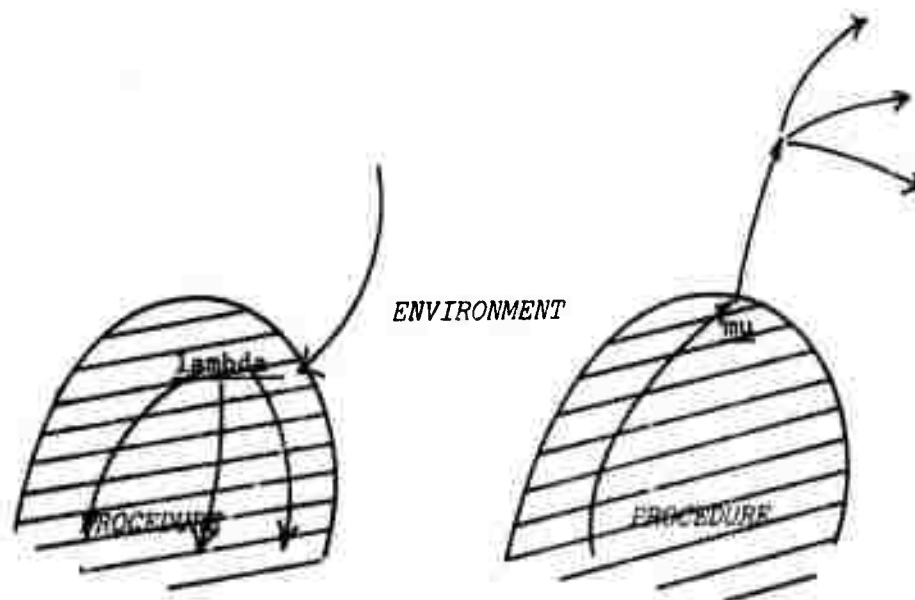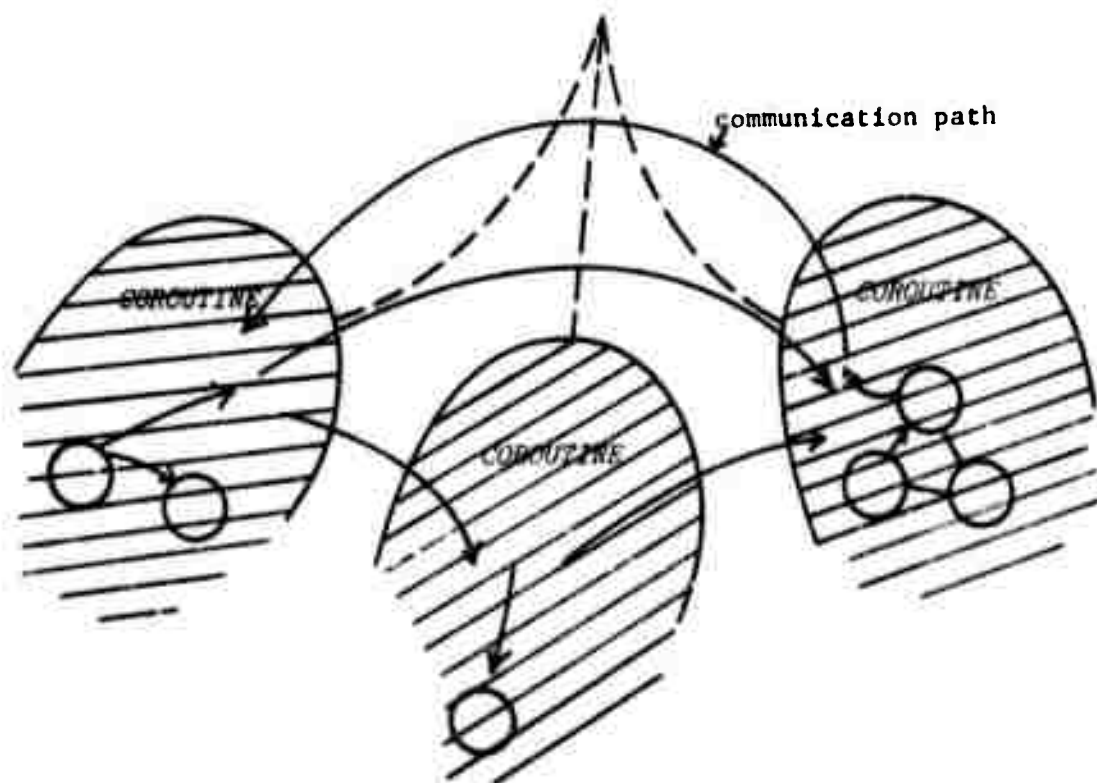
Fig. I-14

nization. For such programs DCPL looks like a host system . In fact
it is possible to write in DCPL a hierarchy of host/guest systems.

Paths of information permit as well to have full programming
generality: the same program may be debugged in a testing environment,
made available in a program shop and put by some user in his own
environment without the need of any surgery (Krutar [8]) . In
conventional programming languages, such a programming generality
may not be available for two reasons: 1. A procedure in general
contains the names of some other procedures to call; as a result
the former procedure is bound to the environment which contains the
latter ones. The situation is better when procedural arguments
are allowed. 2. Input/output operations are performed with parti-
cular instructions (read, write etc ...); thus it is not possible
to debug a procedure in a testing environment with I/O devices
simulated by some programs. In DCPL inputs and outputs are con-
sidered as paths of information coming into and going from the
procedures. Such paths may be connected as well to I/O devices
as to programs.

I.5   Machine organization.

DCPL gives to machine organization a new perspective.

Whenever a program is expressed in a current programming
language, the control may jump from one place to any other one,
a same cell may be accessed from quite different places. This
results in a serious lack of locality. This would not be of any
importance if today's computers were still Von Neumann type machines:
one processor has access to a random access memory whose cells
may be considered as being all "equidistant" from the processor.

Processors becoming faster and faster (and cheaper and cheaper)
the trend in machine organization is to hierarchies of memories.
However, unless many iterative computations are expected to occur
in the fastest level, it is necessary to have at any level of the
hierarchy a transfer rate large enough to "feed" the processor.

A large tranfer rate may be obtained by taking at each
level a large block as unit of transferable information (the slower
the level of memory, the larger the block).

One may believe, however, that only a few words in such
blocks would be really used. For this reason Jack Dennis suggests
in [4] that information should be moved on demand with the word
as information unit, a large transfer rate being assured by perfor-
ming many computations in parallel.

In DCPL it is possible to consider a program as a construct
of "simply-connected" computations which, once triggered, could
be brought in the fastest level of memory and completed without
the need for any additional information.

Moreover it is possible to replace random access memory
by sequential rotative memory. This will be discussed extensively
in part III of this thesis.

In part II DCPL is presented. Part one is concerned with
preliminaries and discussions which the author believes to be relevant
to the subject and important. Some readers might prefer to skip them
and go directly to part two (beginning p. 86).

PART ONE

PRELIMINARIES

Fig. II-1

Fig. II-2

Fig. II-3

environments   Fig. II-4

ENVIRONMENT
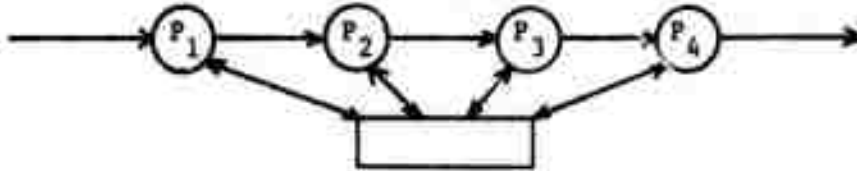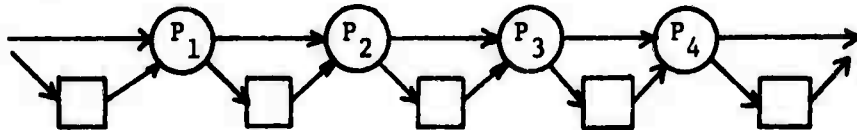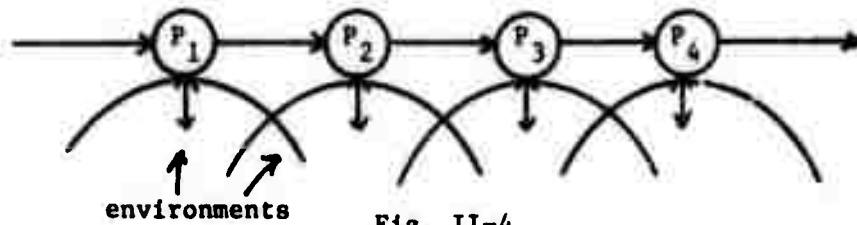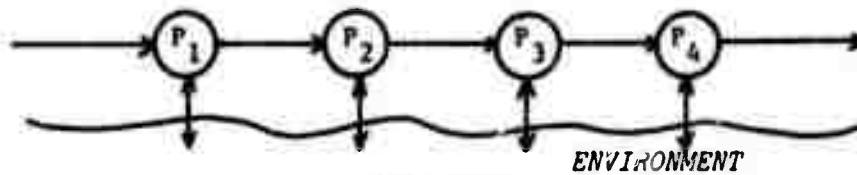
Fig. II-5

signal. On receiving it, the processor expects to find the object

to be transformed at some specific place. When the transformation

has been completed, it puts the resulting object at some (possibly

different) place and sends a control signal to the next processor

to operate. In fig. II-2 one single place is used; a cascade of

places is used in fig. II-3.

The latter situation may be somewhat abstracted. To each

processor is associated an environment (environments may overlap

one another). Each processor is able to perform a specific transformation

on its environment. The process is controlled by a control signal

as before (fig. II-4). In fig. II-5 there is one common environment

accessible by each processor.

In the light of these latter interpretations the arrows

in fig. II-1 appear to have two purposes: 1. They order in time

the occurences of the different transformations, carrying an implicit

control signal (which is the object itself); 2. They specify,

for each processor, on which object the corresponding trar._.ormation

is to be done.

II.1.3 Production line.

Up to now we were interested in transforming one object into

another one. Let us suppose we want to apply a process to a sequence

of incoming objects, fig. II-1 being interpreted as representing

a production line.

For instance we may consider a row of objects advancing on the

communication path. Any processor performs repetitively its transformation

on each incoming object. If the various processors operate at different

rates, it becomes necessary for each arrow to act as a first-in-first-out

queue (fig. II-6).

In some instances, it may be preferable to just have one queue before the process. It is then necessary to synchronize in some way the various processors.

A straightforward solution is to process only one object at a time, one processor at most being at work at any time: P_1 is not to accept any new object before P_4 has completed the transformation of the current one. As shown in fig. II-7 , a backward path links P_4 to P_1. On completion of its task P_4 sends a control signal on this path. On receiving it, P_1 initiates the processing of a new object.

Let us abstract the situation in the following way. We represent an object as well as a control signal by a token placed on the corresponding arrow. An elementary transformation is triggered whenever all the incoming arrows contain a token (fig. II-8b): the tokens are then removed from these arrows and the transformation is in progress (fig. II-8c). On completion of the transformation, a token is placed on each outgoing arrow (fig. II-8d). The processor then stays idle until there is again a token on each incoming arrow (fig. II-8a).

The process is described in figures II-9a to II-9j .

A similar scheme may be used whenever the process uses places to store objects as in fig. II-2 and in fig. II-3 (fig. II-10 and II-11). There is one token representing a control signal, which performs a loop (the servicing of the input and the output in these examples is not discussed here).

Fig. II-6

Fig. II-7

Fig. II-8a
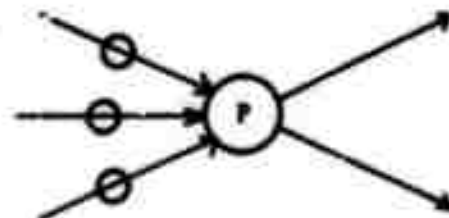
Fig. II-8b

Fig. II-8c

Fig. II-8d

28

Fig. II-9a

Fig. II-9b

Fig. II-9c

Fig. II-9d

Fig. II-9e

Fig. II-9f

Fig. II-9g

Fig. II-9h

Fig. II-9i

Fig. II-9j
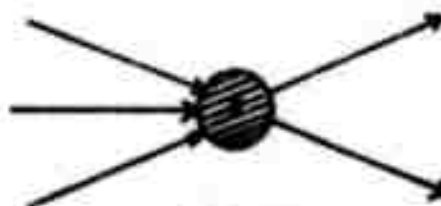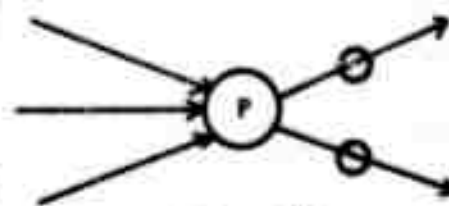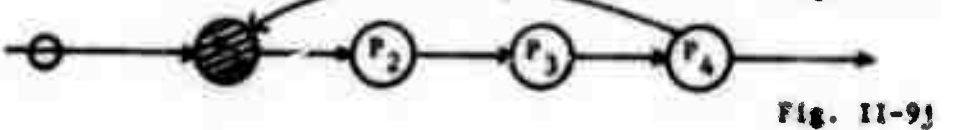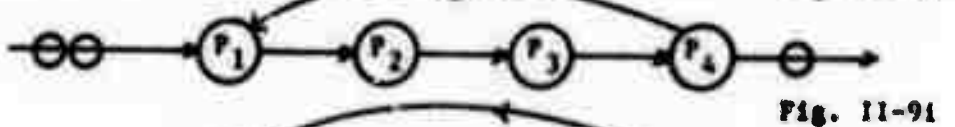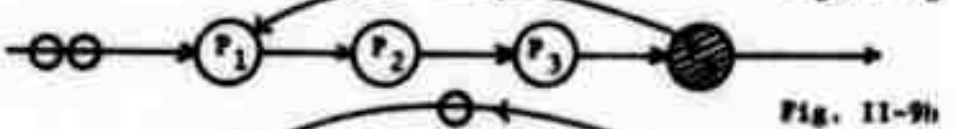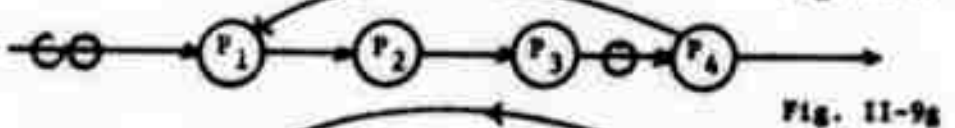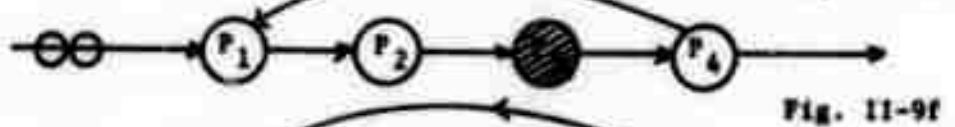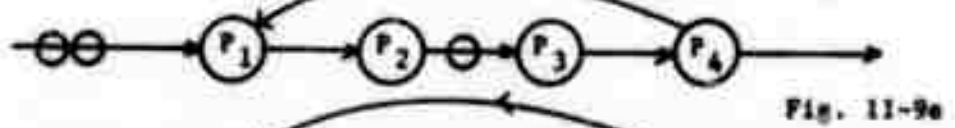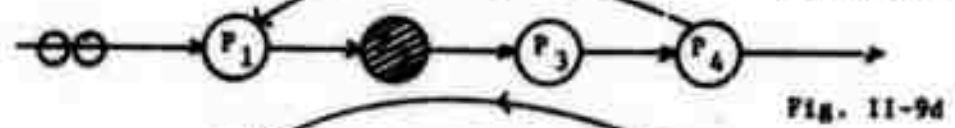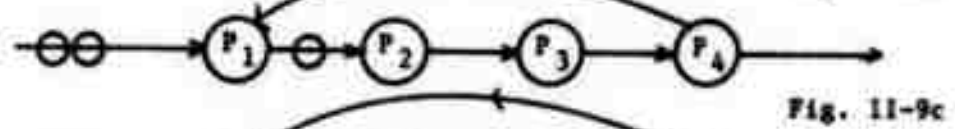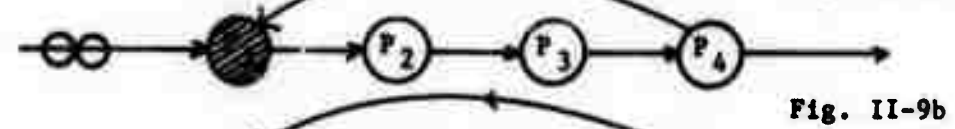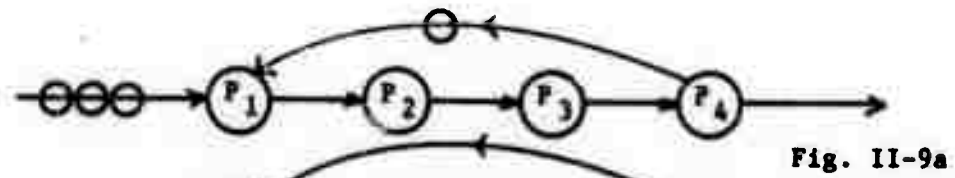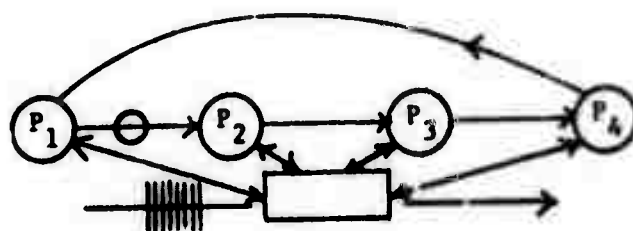
Fig. II-10

Fig. II-11

Fig. II-12

processor at work
processor idle
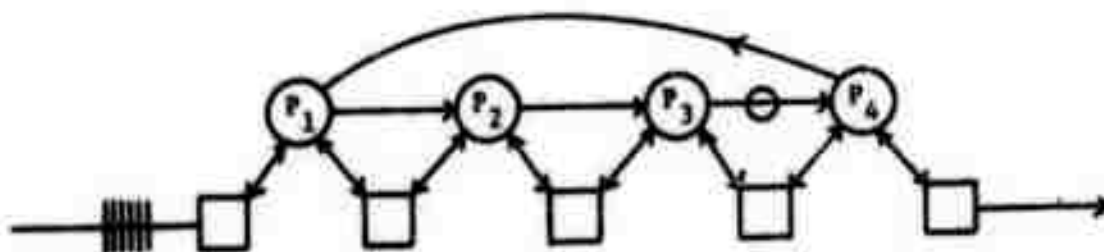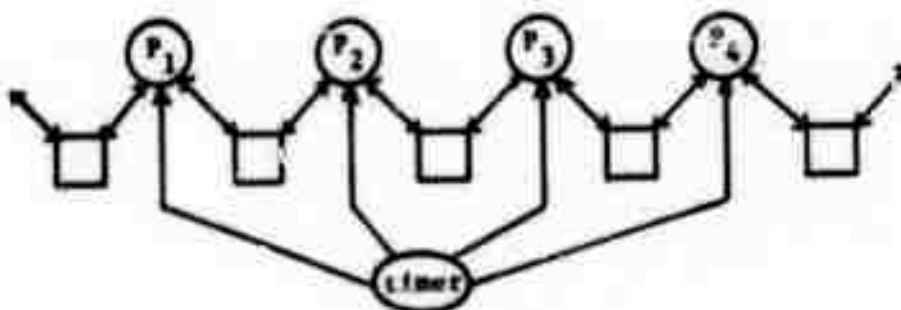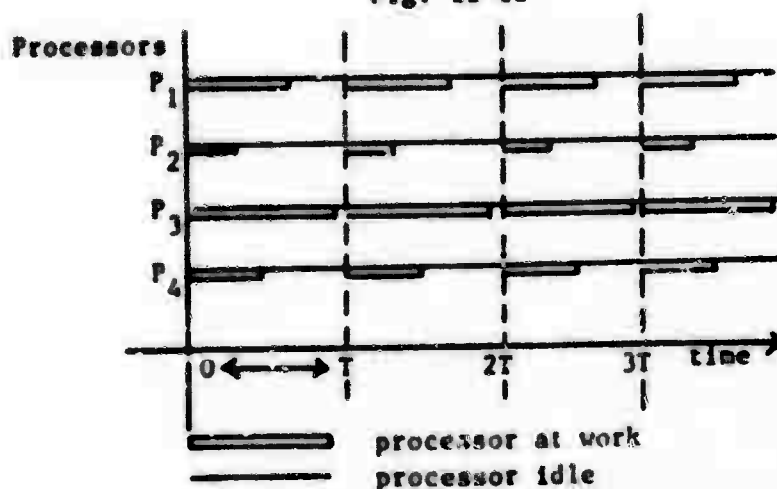
Fig. II-13

11.1.4 **Pipe-line**.

If a large throughput (number of objects processed per unit of time) is desired, such straightforward solutions are not satisfactory: at any time only one processor is working; we would prefer to have the various processors working concurrently on different objects.

If the time required by one processor to transform one object is independent of the object and if this time is known, synchronization may be performed with a central timer. Every T (an amount of time during which any processor may perform its task on one object) the timer triggers all the processors by sending them a control signal (fig. 11-12 and 11-13).

A more general solution may be achieved by replacing the backward path in fig. 11-7 by a sequence of backward paths as displayed in fig. 11-14. Fig. 11-15a-d describe the process.

The processes we have just described are called **pipe-lines**. In fig. 11-12 the pipe-line is **synchronous**; in fig. 11-14 the pipe-line is **asynchronous**.

11.1.5 **Petri-net**.

The situation of fig. 11-14 may be modeled by a Petri-net (fig. 11-16).

The behavior of a Petri-net is very close to the behavior described in fig. 11-8. A Petri-net is made out of transitions (bars in fig. 11-16), and places (circles in fig. 11-16). An arrow may lead from a place to a transition, or from a transition to a place; a place in the former case is called an input place of the transition, and in the latter case, an output place of the transition
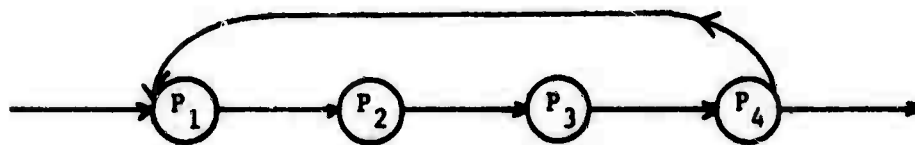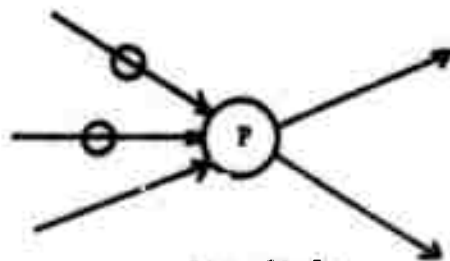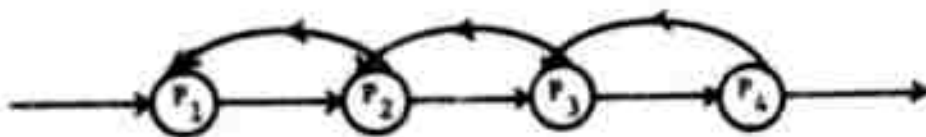
Fig. II-14

Fig. II-15a

Fig. II-15b

Fig. II-15c

Fig. II-15d

place filled by a token

place empty

"P₁ busy" transition "P₂ busy" "P₃ busy" "P₄ busy"

Fig. II-16

input places

transition

output places

Fig. II-17

filled place

empty place

transition ready to fire

Fig. II-18a

right after the firing
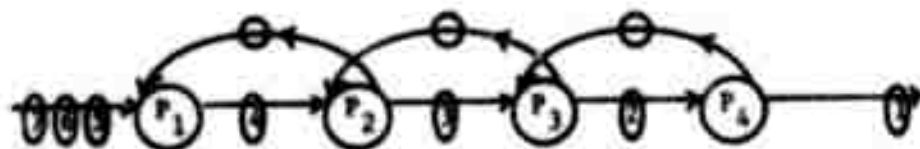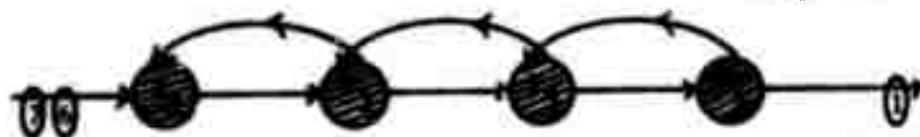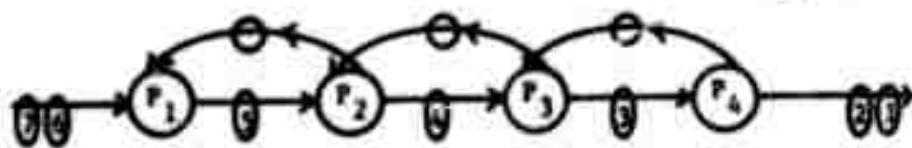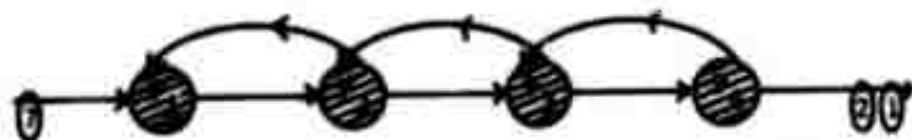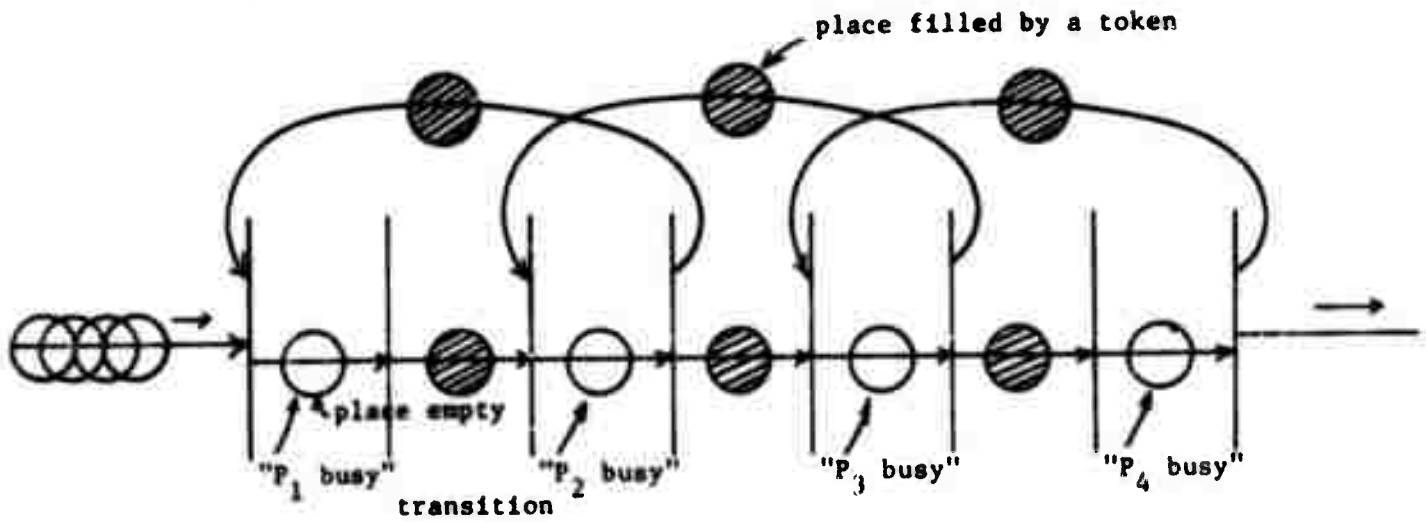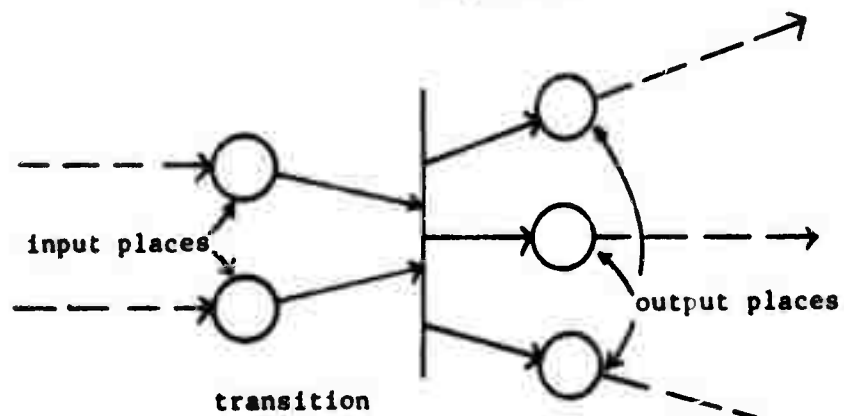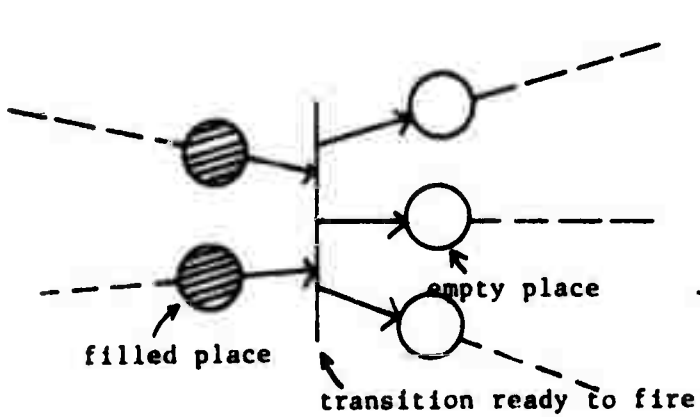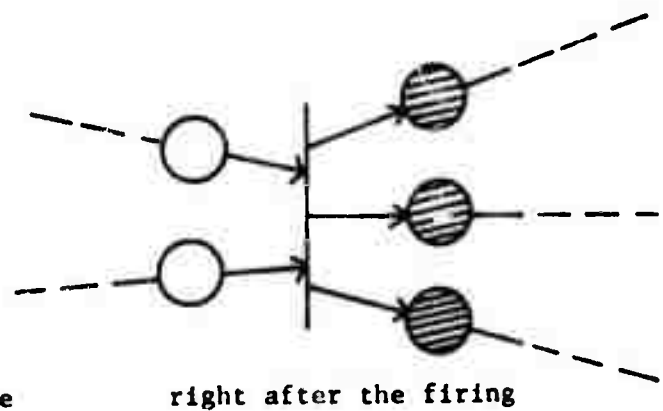
Fig. II-18b

(fig. II-17). A place is either empty or filled by a token. A transition is ready to fire whenever all its input places are filled. (It is supposed that the Petri-net is such that all the output places are empty at the time of firing. For a detailed discussion, please see Holt [9]). The firing of a transition may be viewed as a spontaneous and instantaneous operation: each input place is emptied, a token is placed in each output place of the transition (fig. II-18a-b).

It may be interesting to note that a place may be the input of several transitions. Several of these transitions may be ready to fire at the same time; however only one transition among them may fire at a given time (fig. II-19a-b).

Remark.

A given situation may be modeled in quite different ways depending on the emphasis which is to be placed on various conditions. For instance, fig. II-20 and fig. II-16 account quite differently for the same situation.

II.2   A process as a system of transformations.

In the previous section we discussed simple processes whose elementary transformations were applied sequentially to some object.

In general an elementary process may be applied to one or several inputs producing one or several outputs (fig. II-21a). We require of any would-be elementary process that it may only be applied when all its inputs are available and that its outputs are available only after the process has been completed (fig. II-21b).
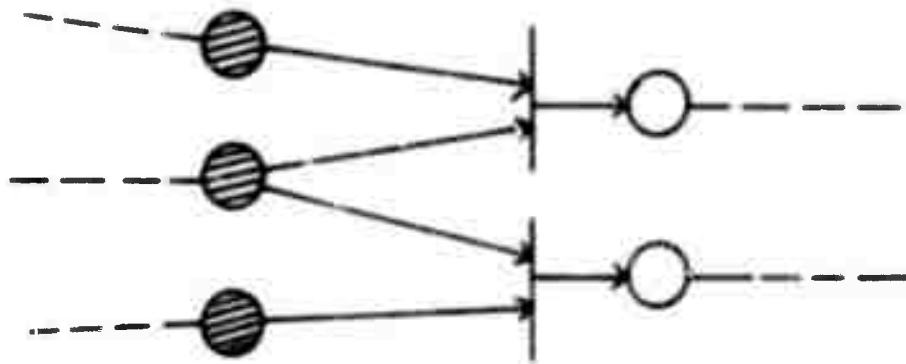
two transitions are ready to fire

Fig. 11-19a
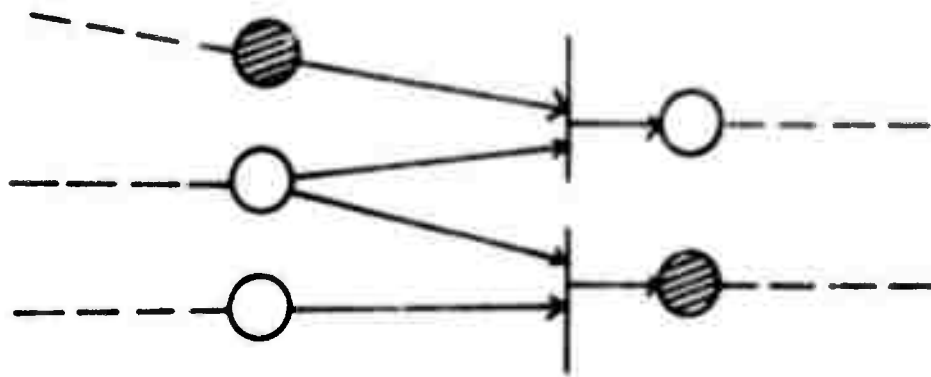
only one transition has

actually fired

Fig. 11-19b
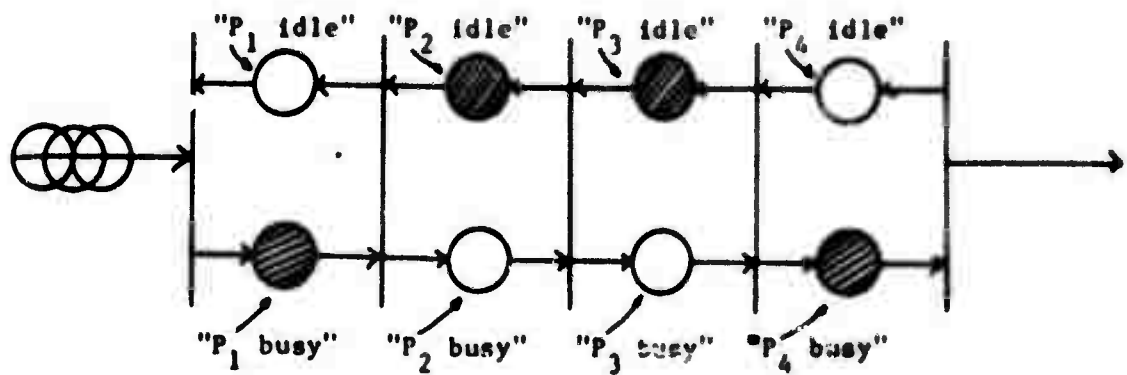
Fig. 11-20
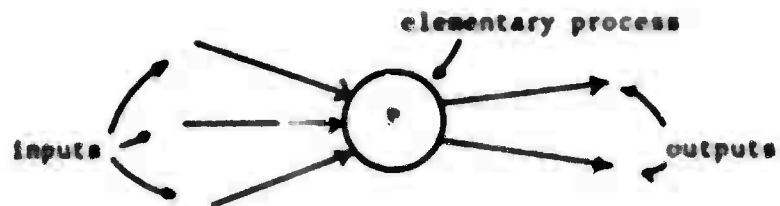
Fig. 11-21a
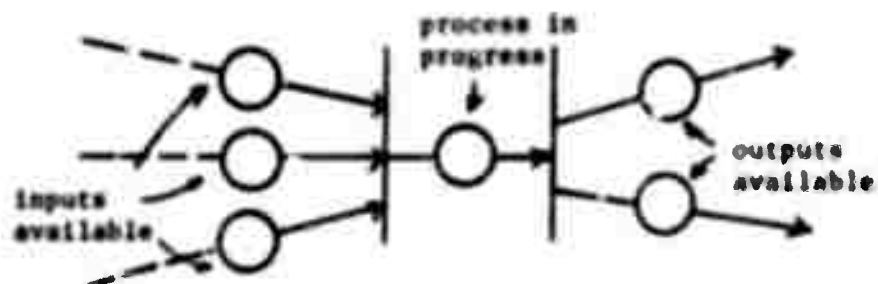
Fig. 11-21b
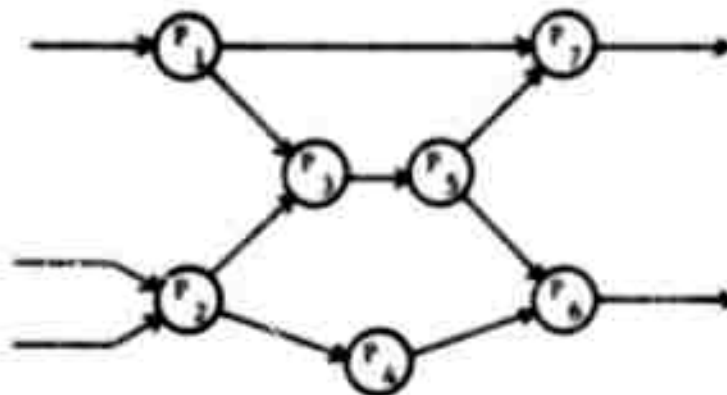
Fig. 11-22

II.2.1 The object-flow model.

Let us now consider a process leading from a set of objects
to another set of objects by applying different elementary processes
$P_1, P_2, \ldots$ as indicated in fig. II-22.

By and large, our previous discussion may be applied here again.
Each node may be considered as being a processor. Whenever an object
is produced it advances on the corresponding path until it reaches
the next processor. A processor is triggered whenever there is an
object on each of its input lines.

II.2.2 Control signals.

In another implementation there are places in which and from
which objects may be stored and retrieved. Control signals may travel
on the arrows. A processor is triggered whenever it has received
a control signal on each of its input lines. The processor retrieves
the objects stored in its input places, performs the associated
elementary process, stores the resulting objects in its output
places, and sends a control signal on each of its output lines.
Fig. II-23 and II-24 present two possible configurations.

II.2.3 Determinacy.

We may abstract such a situation by considering each processor
as being able to perform a transformation on its environment after
it has been triggered by some control signals (fig. II-25). However
it is important to notice that the result of the process may depend
on the order in which elementary processes have been applied if
the different environments are interdependent or if there is one
global environment. A process is said to be deterministic or

Fig. II-23

Fig. II-24

ENVIRONMENT

Fig. II-25

first
input available
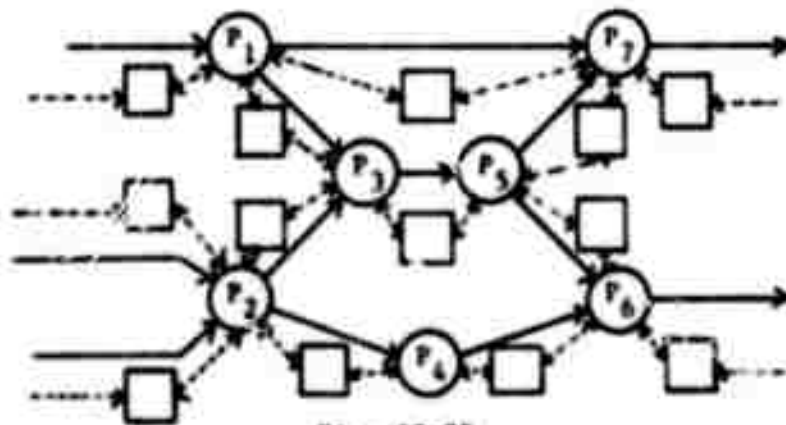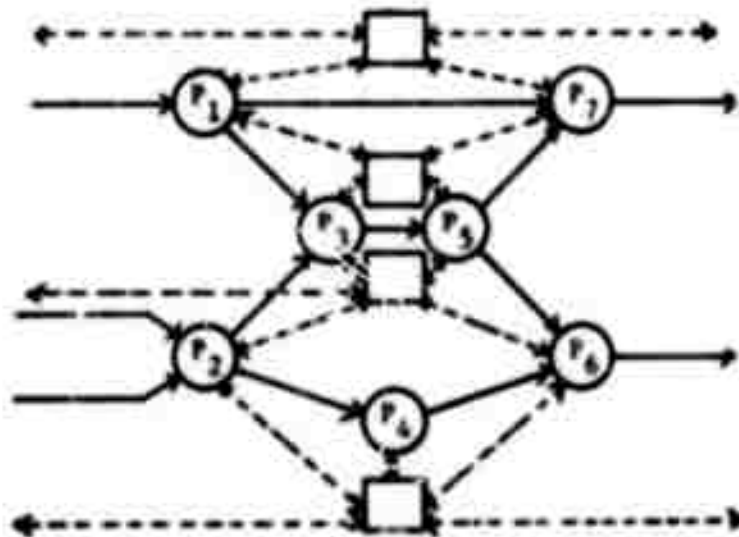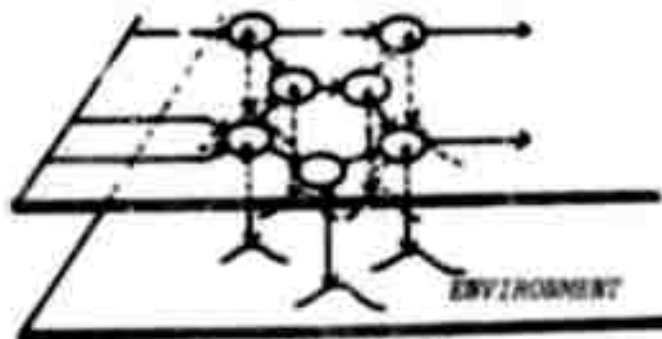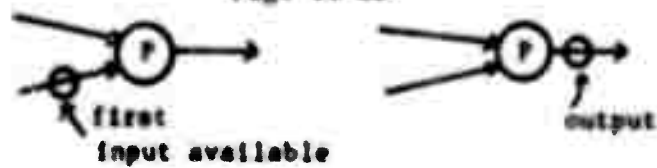
output

Fig. II-26

output-functional if the resulting objects or the resulting environments do not depend on the order in which elementary processes have been applied[15].

A process is said to be completely functional[a] if, for any elementary process P_i, the objects or the local environment to which P_i is applied do not depend on the order in which elementary processes are applied, at the time when P_i is triggered.

The process of fig. II-22 with objects flowing on arrows is determinate if the elementary processes are output-functional; we owe this result to the requirement we made about elementary processes (Fig. II-21). However the process would not be determinate if we had allowed races to occur with, for instance, an elementary process having as an output the first input received (fig. II-26).

The processes of fig. II-22 and II-23 are determinate; however, some other configurations of places would have given non-determinate processes.

II.2.4 Service-on-demand.

A same elementary process may occur at different locations in a process. Let us assume that the process of fig. II-27 is to be carried out by a team of four workers, each worker being able to perform a specific elementary process A,B,C or D.

The flow chart of the process is displayed on a table with a light bulb on each node and a place on each arrow. Whenever all

---

a) The term "determinate" is often used instead of deterministic. However determinate is sometimes opposed to deterministic and means then completely functional. Such opposition is not a very important one from the point of view of this thesis, and we shall use "determinate" in general. However, whenever the opposition is meaningful we will use deterministic and completely functional.

Fig. 11-27

light bulb

empty place

place filled
by an object

turned on light
bulb
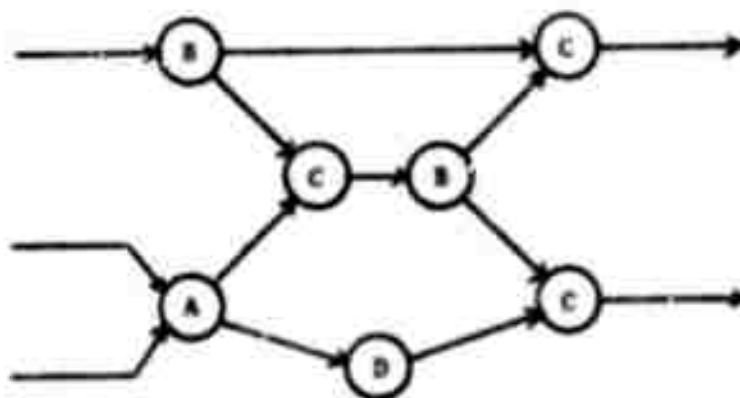
Fig. 11-28a   Fig. 11-28b   Fig. 11-28c
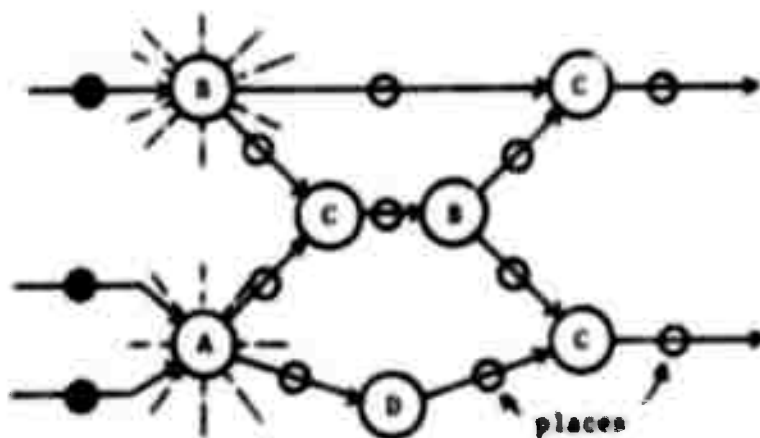
places

Fig. 11-29

input places of a node are occupied by objects, the bulb on this
node is automatically turned on (fig. II-28).

At the beginning the objects to be processed are put on the
input places (fig. II-29).

Whenever a worker is idle, he looks for a turned-on bulb
representing an elementary process he is able to perform. On finding
one, he takes the objects from the input places, performs the
corresponding elementary process and puts the resulting objects
on the output places; then the worker is idle again.

In such an organization, the workers are said to _service_
on demand the elementary processes. The demands are issued locally
whenever some local condition occurs. The same scheme may be used
if one worker is able to perform more than one elementary process.
However, if several workers were able to perform a same elementary
process and if they could have access concurrently to the flow
chart, a system of arbiters(assigning to each idle worker an elementary
process to be performed) would be necessary (see Patil [20]).

Let us consider again fig. II-22. Two elementary processes
are said to be _ordered_ if there is a path of arrows leading from
one to the other. The executions of two such elementary processes
are then necessarily ordered in time. Two elementary processes are
said to be _concurrent_ if they are not ordered. When the whole process
is carried out, two such elementary processes may be performed
either concurrently or one after the other, according to considerations
which are not relevant to the level of organization in which we are
interested here. In the situation of fig. II-25, a sufficient
condition for the process to be deterministic is that any pair

of concurrent elementary processes (considered as operators on the environment) commute. A sufficient condition for the process to be completely functional is that any pair of concurrent elementary processes have "independent" environments (fig. 11-37). This latter condition holds in the situations displayed in fig. 11-23 and 11-24.

11.2.5 Sequential execution.

Let us now suppose that the process of fig. 11-27 is to be performed by one worker executing the different elementary processes one after the other. At any time there is a set of pairwise concurrent elementary processes which are ready to be executed (in our previous scheme these processes are those whose bulbs are turned on). Our worker may choose at random one process in this set and perform it. The occurence of this elementary process is no longer in the set, but at its completion, new elementary processes might be in the set. Then the worker repeats the same sequence of actions until the whole process is completed.

If the process is planned to be executed by only one worker, it may be described as a sequence of elementary processes. Obviously different sequences are possible (fig. 11-31 and 11-32). Together with the sequence we have to know the inputs and the outputs of each elementary process and how one is related to another. In fig. 11-33 corresponding inputs and outputs are linked together. In fig. 11-34 there are four places; to each input (resp. output) is associated a place from which (resp. in which) the object is to be taken (resp. stored). These figures display the same topologies as fig. 11-32 and fig. 11-35.

42

local environments
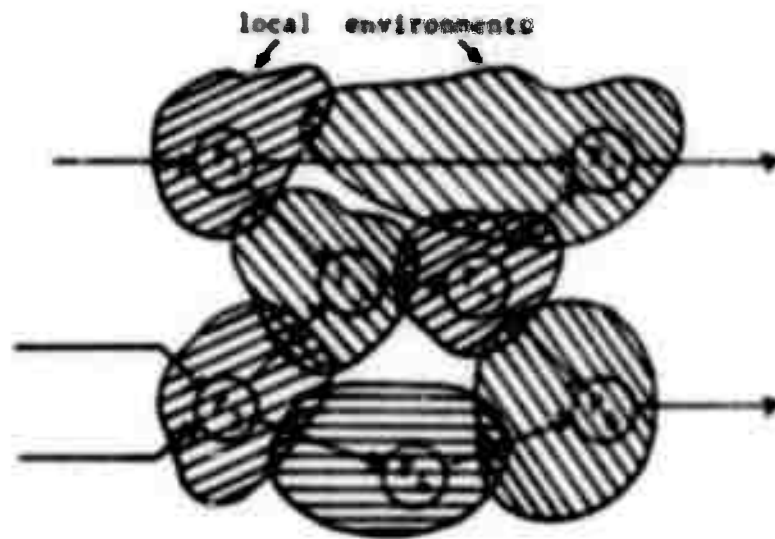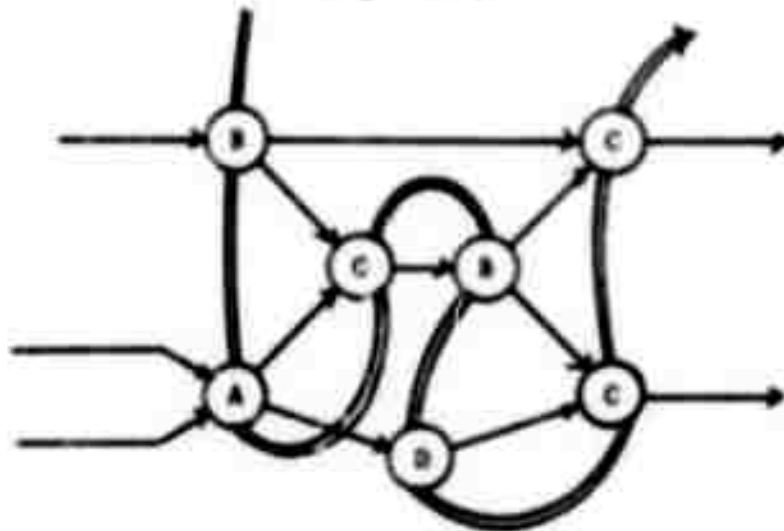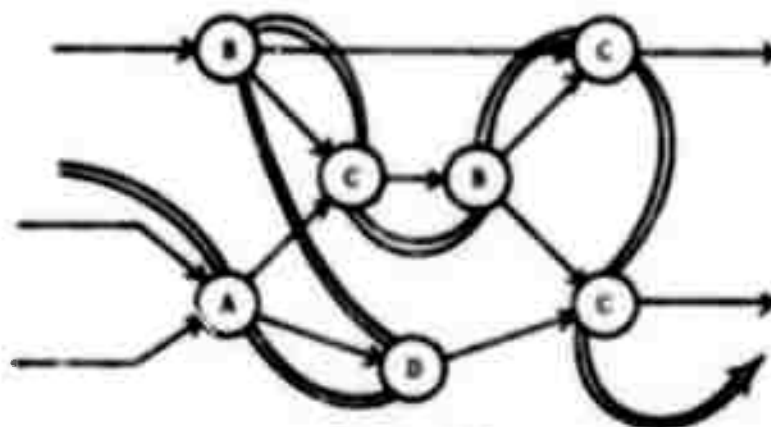
Fig. 11-30

Fig. 11-31

Fig. 11-32
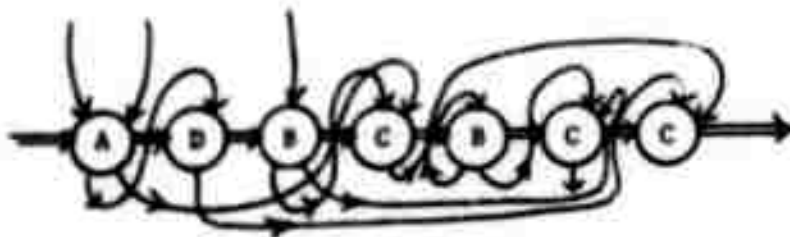
Fig. 11-33
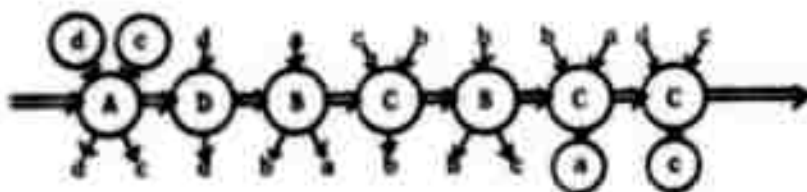
Fig. 11-34
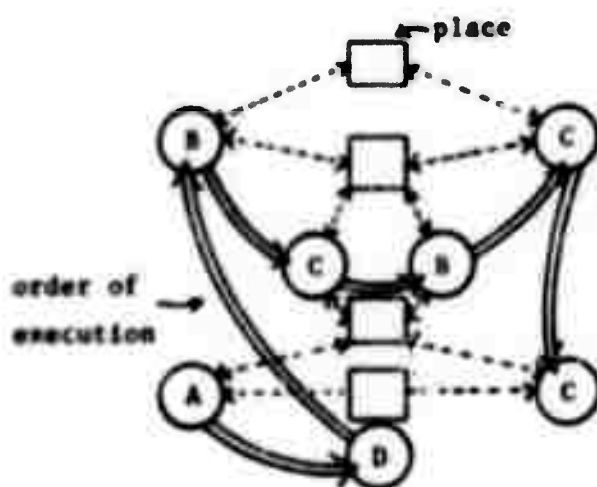
Fig. 11-35
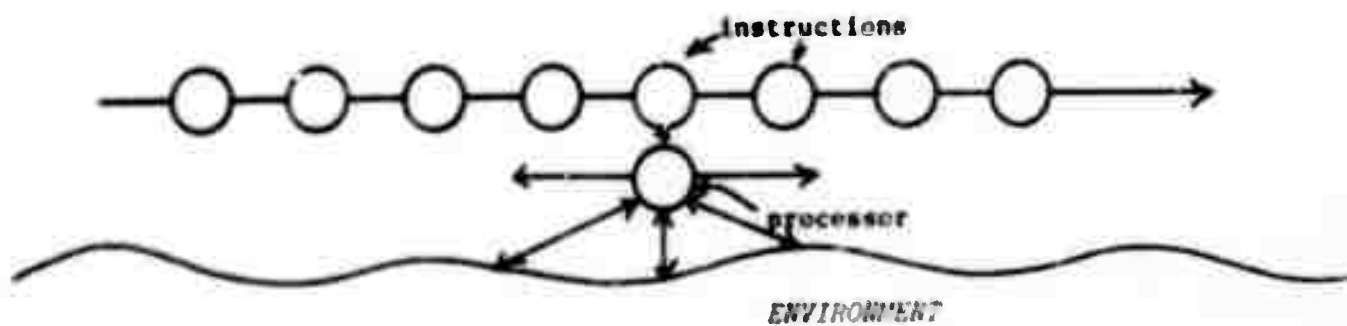
Fig. 11-36

II.2.6 Centralized vs. distributed control machine.

Abstracting the situation, we may consider the description of the process as a sequence of instructions. One processor scans such a sequence performing one at a time the corresponding elementary transformations on the environment (fig. II-36).

We call a machine with one or several processors working in such a way a centralized control machine. On the other hand we call a distributed control machine a machine whose special purpose processors act on a service-on-demand basis: any action is triggered by local conditions occurring asynchronously.

Although such a distinction may appear rather artificial under certain circumstances, the notion of a distributed control machine will be helpful in the sequel.

CHAPTER III

A COMPUTATION AS THE REALIZATION

OF A FORMAL OBJECT

So far, we have not particularized our discussion to any
special object or to any special elementary process. In this chapter
we consider formal objects called obs which may be built up, and
then realized relatively to our universe of values and operations.

III.1 Combinations and obs.

The following notions are due to Curry [3]. They are very
similar to Landin's notion of applicative structure [12]. In many
other works, such notions are expressed from a syntactic point
of view.

An ob is a formal object which is either primitive and
called an atom, or built up by combining already constructed obs
according to some elementary processes called combinations.
A combination is a particular elementary process: an ob being
given, we know at once what is its structure; we know whether
or not the ob is an atom, and if the ob is not an atom we know
how it has been constructed: by applying which combination to
which obs. It is therefore understood that obs constructed by
different processes are different as obs.

As a result, concatenation of symbols and strings, into
strings, is not a combination. In Lisp, forming the "cons" of
two S-expressions is a combination. In the same way a sentence

which is syntactic relatively to a phrase structure grammar is
not an ob; however, any syntax tree of the same sentence may be
considered as an ob.

A symbol denoting a combination is called a _combinator_.
The _degree_ of a combination or of a combinator is the number
of obs to which the combinator is applied. An atom is considered
as a combination of degree zero.

Fig. III-1 represents an ob O, with C_0 , C_2 and C_3 being
respectively combinators of degree zero, two and three. The figure
displays the "topology" of O, i.e., the different obs, _components_
occurring in the construction of O.

III.2 _Realization of a process or of an object_:
categories and functors.

Let us consider an elementary process with input and output
places (upper part of fig. III-2). When realizing such a process,
each place is mapped into a value of the universe of values and
operations (see introduction), each elementary process into an
operation, such that the value associated to the output place is
obtained by applying the operation (associated to the elementary
process) to the values associated to the input places (fig. III-2).

It is possible to realize an ob by assigning a value to
each component and an operation to each combination with the same
rule as previously: the value associated to the ob, result of the
combination, is the value obtained by applying the operation
(associated to the combination) to the values associated to the obs,
operands of the combination (fig. III-3).
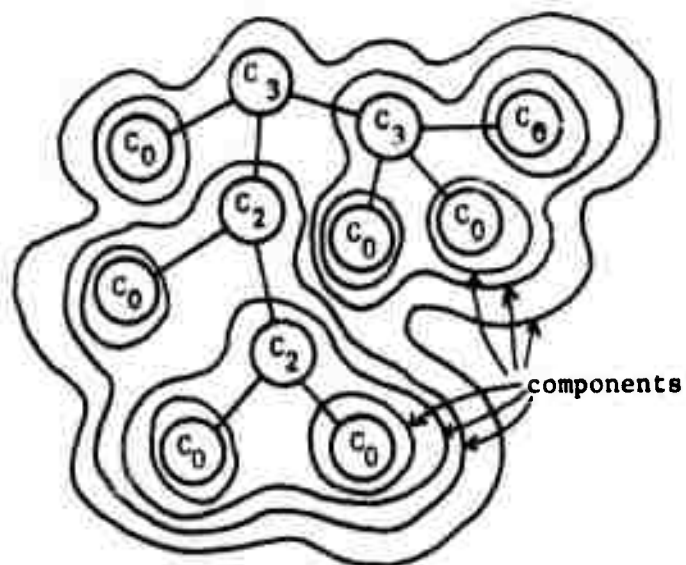
Fig. III-1
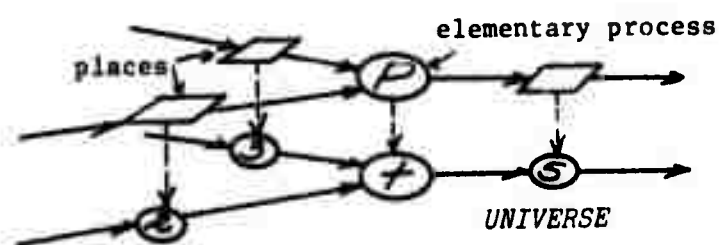
Fig. III-2 (to be viewed in space)
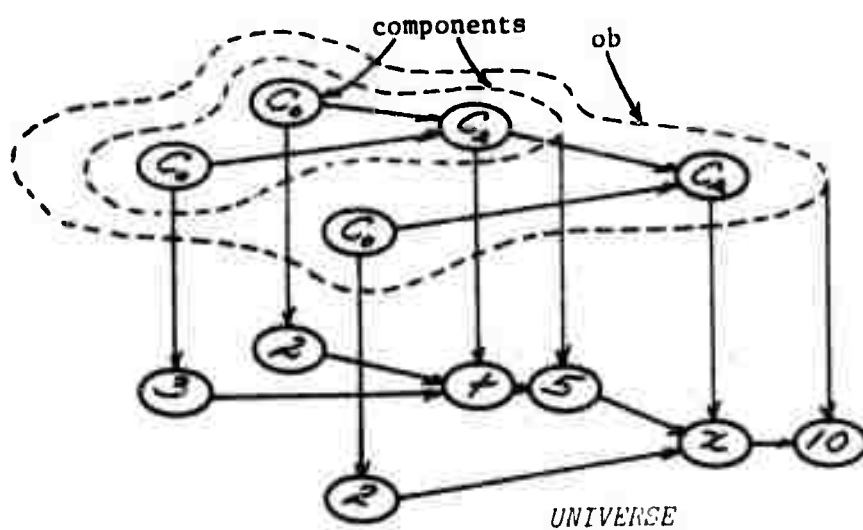
Fig. II-3   (to be viewed in space)

It is useful to abstract slightly such situations by introducing a variation of the concept of categories and functors (see Mitchell [17] ; as defined here, categories may be extended in a trivial way to be mathematical categories).

A category is a set of objects among which some given relations, which we shall suppose to be mappings, hold. So, for instance, the previous universe is a category. Any ob is also a category: the components of the ob are the objects of the category, and the relations are here the mappings actualized by the combinations.

Let C and C' be two categories. A functor is a mapping $T: C \to C'$ which associates:

- to each object O in C, an object $T(O)$ in C'

- to each mapping R in C, a mapping $T(R)$ in C'

such that whenever

$$O_1 , O_2 , \ldots , O_n \xrightarrow{\quad R \quad} O_1' , O_2' , \ldots , O_p' \quad \text{in C}$$

we have either

$$T(O_1), T(O_2), \ldots , T(O_n) \xrightarrow{\quad T(R) \quad} T(O_1'), T(O_2'), \ldots , T(O_p')$$

and T is called a covariant functor (fig. III-4a), or

$$T(O_1'), T(O_2'), \ldots , T(O_p') \xrightarrow{\quad T(R) \quad} T(O_1), T(O_2), \ldots , T(O_n) \quad .$$

and the function is called a contravariant functor (fig. III-4b).

III.3 Iterative and recursive processes.

The previous definitions are significant because covariant and contravariant functors lead to two different computational schemata. We shall see that they correspond to the intuitive distinction of an iterative versus a recursive computation, when applied to sequential processes.

T is a covariant functor:
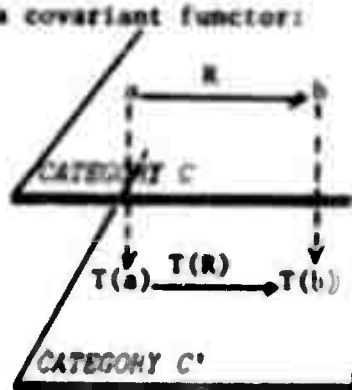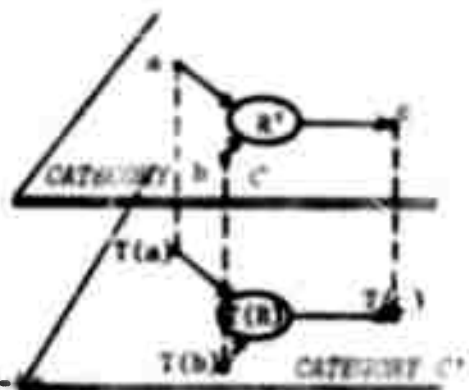
Fig. III-4a

T is a contravariant functor:
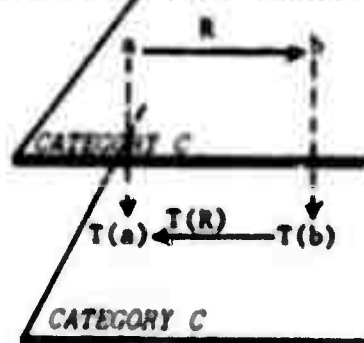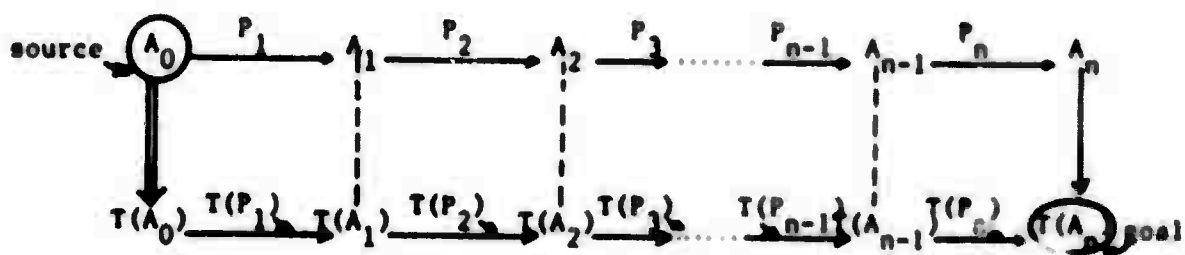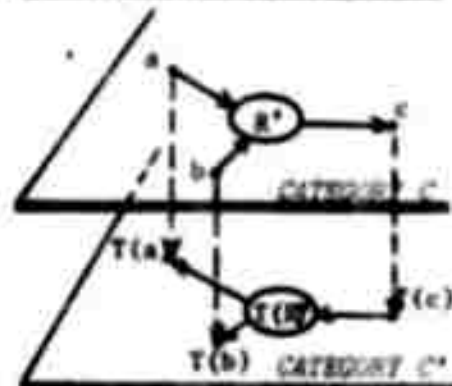
Fig. III-4b

Fig. III-5
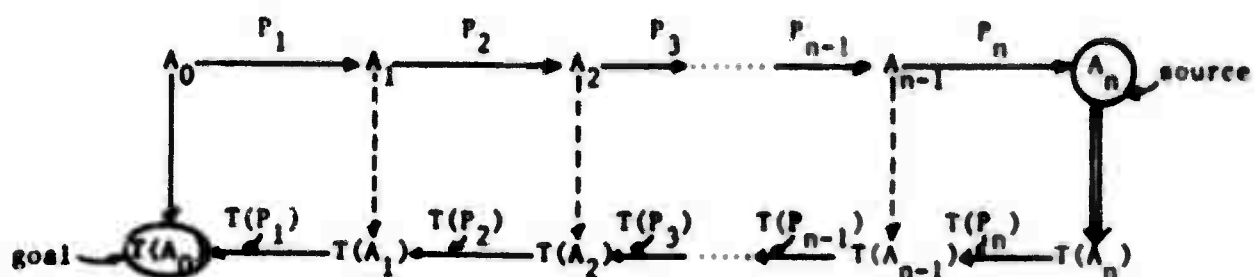
Fig. III-6

To any mapping R is associated an operation $T(R)$ (which may be primitive or not) in the universe of values and operations, and the correspondance $R \rightarrow T(R)$ is given beforehand. For some objects a_i , called _sources_, the values $T(a_i)$ are given in the universe of values. For some b_j the values $T(b_j)$, called _goals_, are to be determined.

Let us consider a sequence of objects, each object (the first one excepted) being obtainable by applying a particular elementary process to the previous object in the sequence. To each elementary process P_i is associated an operation $T(P_i)$, T being a covariant functor. The first object A_0 is the source; the value associated to the last object A_n is the goal (fig. III-5).

The computation of $T(A_n)$ may be performed straightforwardly with one storage place whose contents will be called the current value. At the beginning $T(A_0)$ is the current value; the control then proceeds sequentially from the source to the last object of the sequence; whenever a mapping P_i is encountered, the operation $T(P_i)$ is applied to the current value, and the result is taken as the new current value. When the control reaches A_n , the current value is the goal $T(A_n)$.

In pseudo-Algol, the process may be described by means of an iteration:

```
... currentvalue := T(A_0) ;
    for i := 1 step 1 until n do
        currentvalue := T(P_i)(currentvalue) ;
    .........
```

We consider now the same sequence of objects, T being a contravariant functor. A_n is the source and $T(A_o)$ the goal (fig. III-6).

It may seem that $T(A_n)$ being given, the computation of $T(A_o)$ is completely similar to our previous computation $T(A_o)$ and $T(A_n)$ being permuted: $T(P_n)$, $T(P_{n-1})$, ... , $T(P_1)$ would be applied sequentially to $T(A_n)$. However, in general, this reverse sequence of mappings is not known beforehand.

The control goes as before from A_o to A_n ; whenever a mapping P_i is encountered, the corresponding operation $T(P_i)$ is placed at the top of a last-in-first-out queue (a stack). After the control has reached A_n , the computation proceeds as in our previous example, the operations being retrieved, now, from the stack.

The arrows on the diagram may serve as a built-in stack. Let us consider as an example the factorial function:

(1) $fact(n) = \{$ **if** $n = 0$ **then** 1 **else** $n \cdot fact(n-1)\}$

Let us compute $fact(3)$. The goal is $fact(3)$, the source is 0 with $fact(0)$ being 1. The diagram is displayed in fig. III-7.

We can interpret the program as a diagram modification scheme. The arrows $T(P_i)$ are constructed as the control proceeds along the arrows P_i. When the control reaches the source the two sequences are bound. Then the control proceeds towards the goal, performing the operations $T(P_i)$. (Fig. III-8).

We may wonder what interpretation an iterative factorial may have.

52

goal   fact(3)   +3   fact(2)   +2   fact(1)   +1   fact(0)=1   source

Fig. III-7

goal

+1
+3

+3   +2

+3   +2   +1

+3   +2   +1   source

+3

+3

goal

Fig. III-8

fact(4)   fact(3)   fact(2)   fact(1)

Fig. III-9
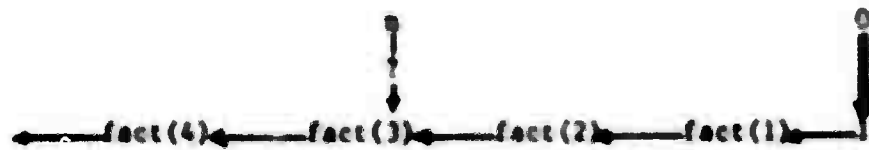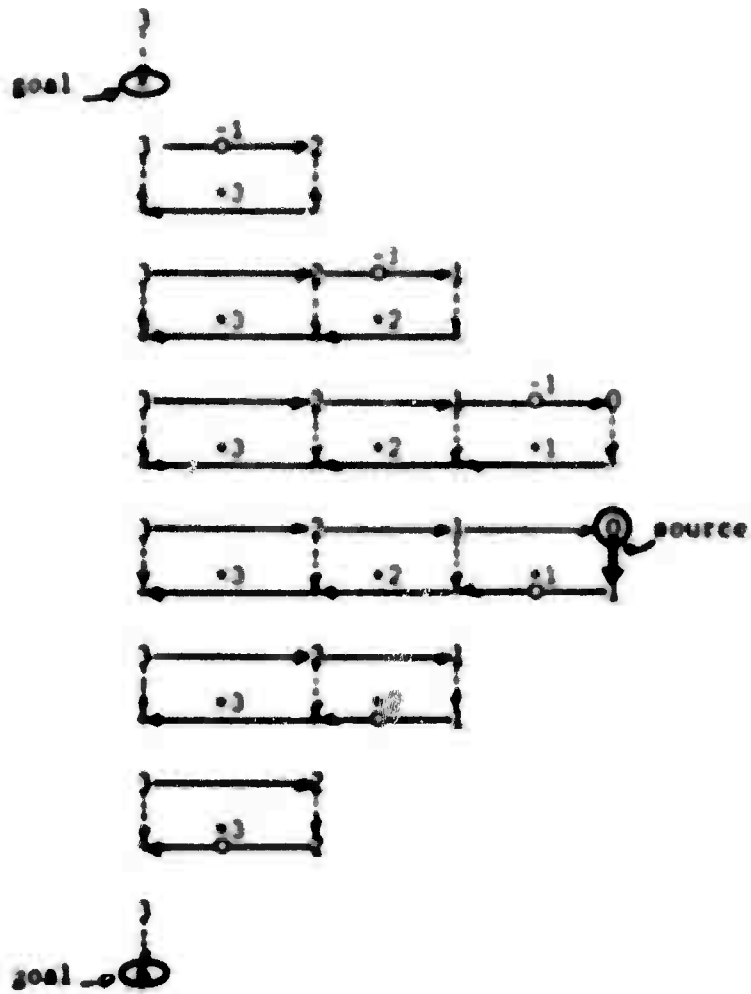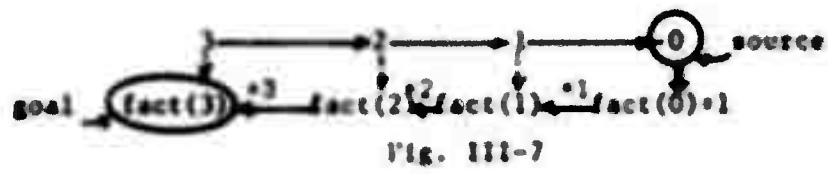
A "while-iterative" factorial: (2) | x := 1 ; y := 1 ;

                                    while $x \neq n$ **do**   **begin** $y := x \cdot y$;

                                                 $x := x + 1$ **end** ;

                                    y |

A "for-iterative" factorial: (3) | y := 1;

                                    for $i := 1$ **step** 1 **until** n **do** $y := i \cdot y$;

                                    y |

In these programs, the programmer knows beforehand what the sequence $T(P_n)$, $T(P_{n-1})$, ... is. As a result fact(0) being known, fact(1),
fact(2), ... may be computed iteratively, one after the other (fig. III-9).
In the first program, at each step, a test determines whether the goal
is reached or not. In the second program, the programmer knows
beforehand that the goal will be reached whenever the control
has passed three arrows.

As a result we can consider algorithm (1) as the most general.
Algorithm (2) uses a particular situation. Algorithm (3) uses a
still more particular situation.

In the previous discussion elementary processes were restric-
ted to transform one object into another one. The general case where
elementary processes are applied on several input objects and produce
several output objects may receive a rather similar treatment.

Objects and mappings relating one object to another constitute
a category (fig. III-10). Let T be a functor, associating to each
elementary process P_i an operation in the universe $T(P_i)$. If T is
covariant (fig. III-11) we can take the input objects a,i,j as
sources, the goals being the values of the output objects c and n.
The computation is performed with a distributed control. If T is
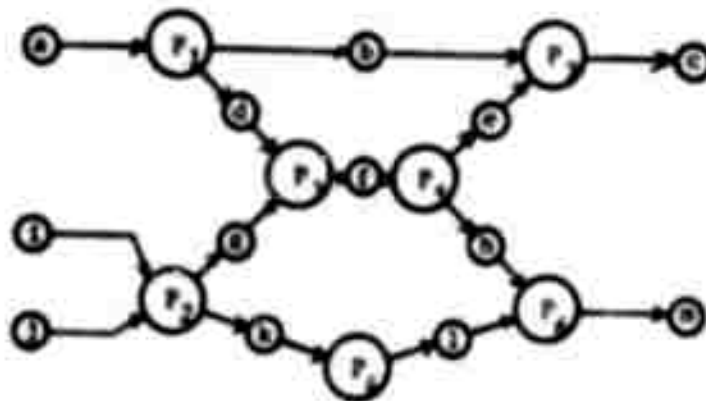
Fig. III-10
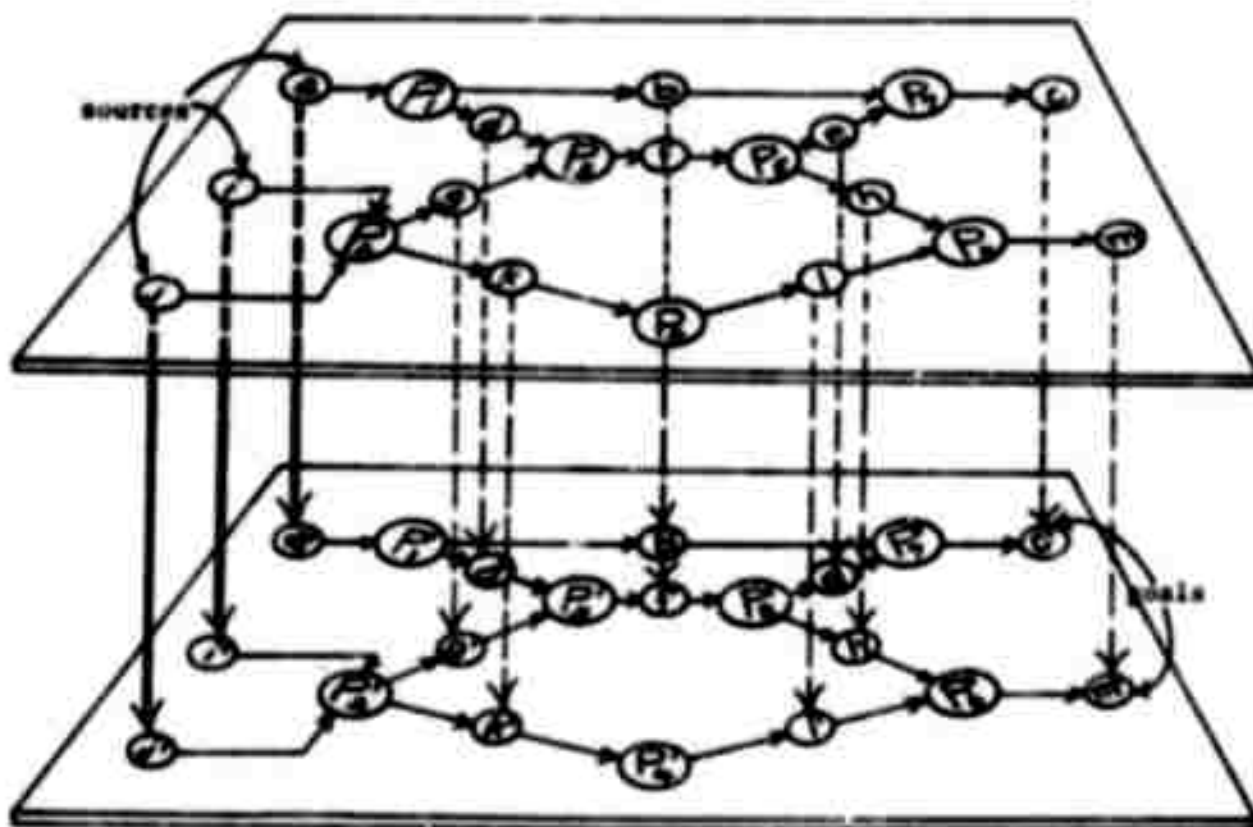
T is a covariant functor (in this figure X' denotes T(X) for any symbol X)

Fig. III-11

contravariant (fig. III-12) the output objects are taken as sources,
the goals being the values of the input objects $T(a)$, $T(i)$, $T(j)$.
The arrows representing the mappings $T(P_i)$ are used as a first-in-
last-out-like storage device. We may notice that a stack,being an
essentially sequential storage device, cannot be used here.

III.4 Synthetic and inherited attributes.

III.4.1 Definitions.

When applied to an ob, covariant and contravariant functors
lead to the notions of synthetic and inherited attributes. Indeed,
it is sometimes interesting to associate to each component of an ob
an attribute, these attributes being structurally related to one
another. In our terminology, the components of an ob are considered
to constitute a category, which is mapped by some functor into the
universe of values and operations. If the functor is covariant,
the attribute of the given ob may be computed whenever the attributes
of the atoms (primitive obs) are known. Such attributes are called
synthetic attributes. If the functor is contravariant, the attribute
of an atom may be computed whenever the attribute of the given ob
is known. Such attributes are called inherited attributes (see
Knuth [11] where a syntactic point of view has been taken).

III.4.2 The value of an expression as a synthetic attribute.

The expression $(2 \cdot 3 \cdot 2) \cdot 2 + 1$ (see section 1.2.1)
may be regarded as an ob Its evaluation requires only one synthetic
attribute, the value.
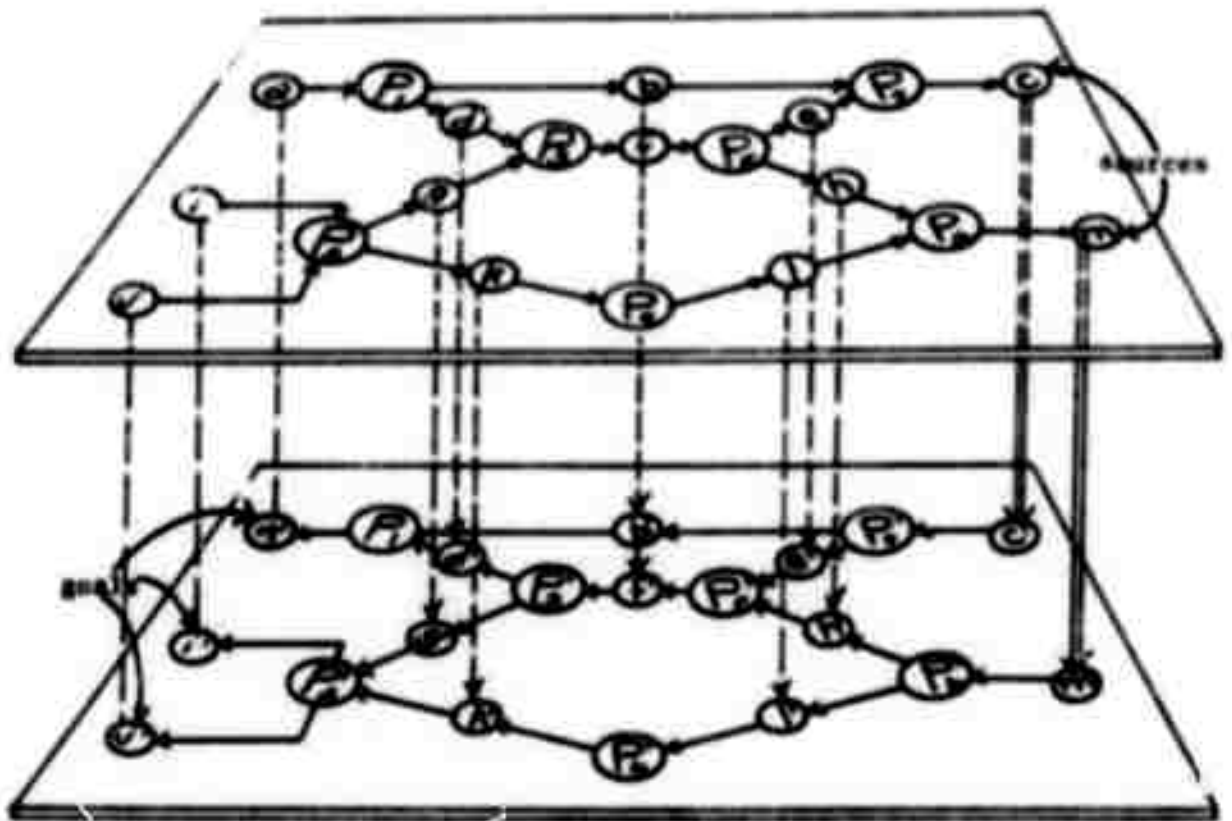
T is a contravariant functor

Fig. III-12

Whenever an ob like $(x \circ y + x) + x + 1$, containing formal variables, is to be evaluated, it is necessary that the environment determines the value of each variable. The realization of the ob is a covariant functor (fig. III-13).

III.4.3 The environment as an inherited attribute.

In the previous example the environment was considered as being global relatively to the expression, and as being accessible by any variable. We may also consider the environment as an inheri- ted attribute.[a] The evaluation may then be viewed as having two phases: the binding and the evaluation proper (fig. III-14).

In the example of section I.2.1,

$(2 + 3 \to x ; 5 \to y ; x + y \to x ; y + 1 \to y ; 2 \times y - x)$,

the $\to x$ and $\to y$ may be considered as being operators on environments (fig. III-15).

---

a) To each node is associated an attribute: a local environment. As a result, to two occurences of a same variable in a computation may correspond two different local environments which may associate to these variable-nodes two different values.

58

Fig. III-13

binding

evaluation

proper

goal

Fig. III-16

value (synthetic attribute)

piece of environment (inherited attribute)
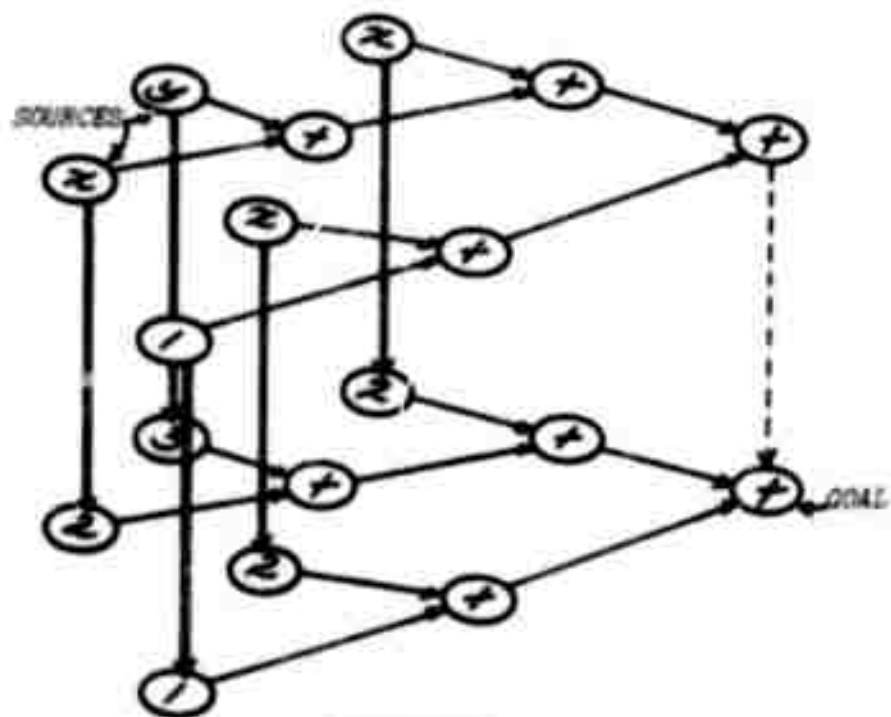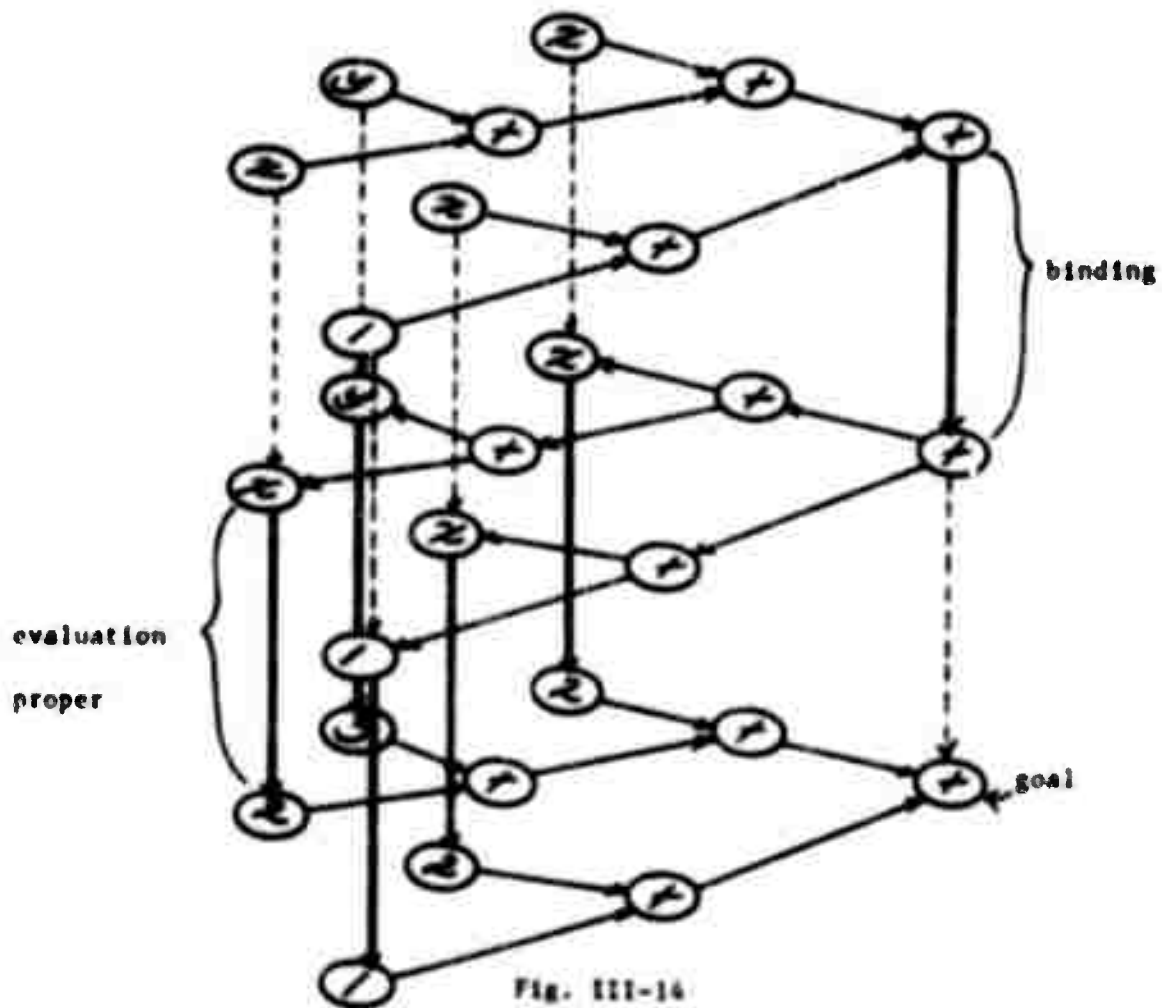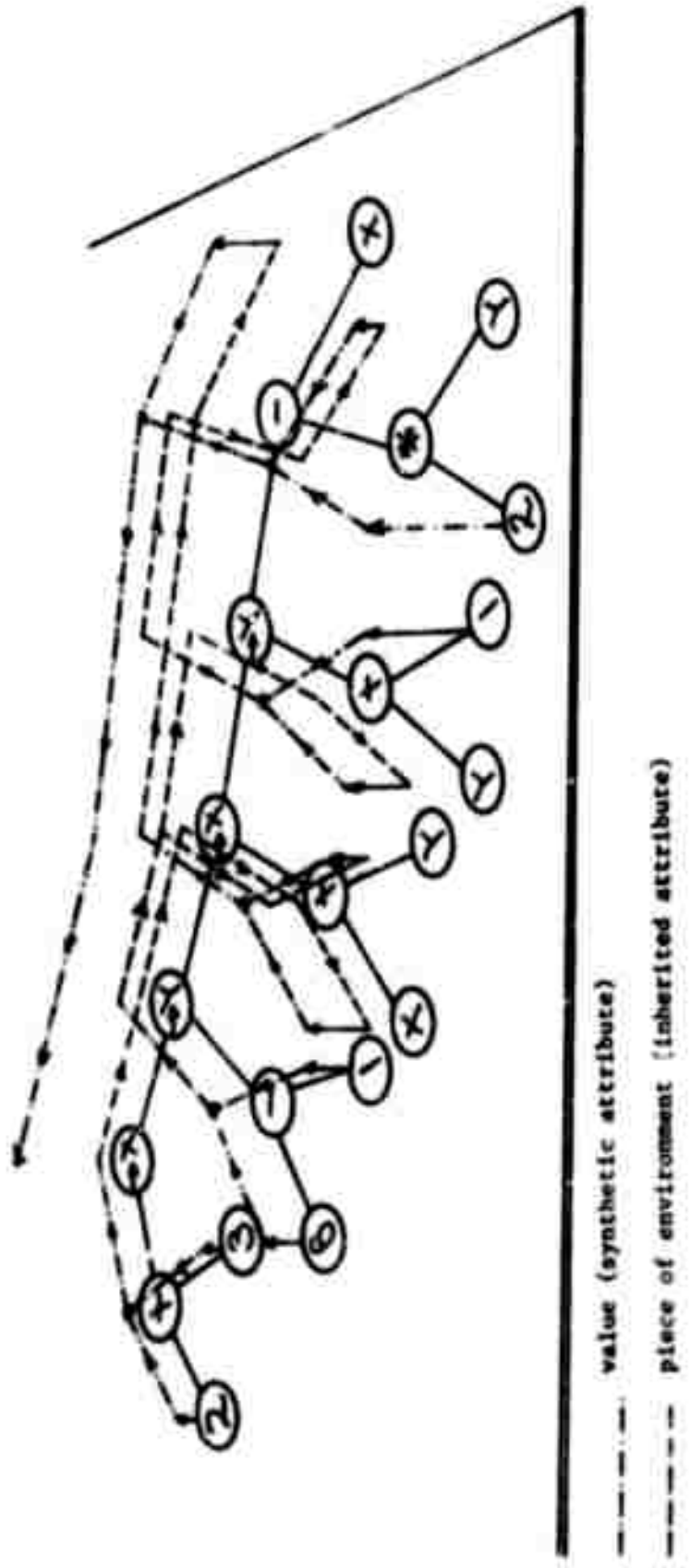
Fig. III-15

CHAPTER IV

SOME COMPUTATION MODELS

IV.1   Introduction

In this chapter, two different approaches of modelling
computations are discussed. In both of these approaches, the
models present in some way concurrency and distribution of
control.

In the first approach, the directed graph is taken as
a model of computation [6,10,10a,16,21,23] . In section IV.2
we will discuss the "computation model with data flow sequencing"
due to Adams [1]. Such a representation is of interest for us
since a DCPL program may appear to be a data flow model after
the binding has been performed.

In the second approach, a functional computation is
considered as an expression; the evaluation of such an expression
may be carried out by some abstract machine. The lambda-calculus
provides a machine whose elementary operations are replacement
and substitution (section IV.3).

Curry's combinators allow to use a much simpler machine,
using replacement only. Strikingly enough, variables are not used
at all in this representation. As a result of the simplicity
of the machine and of the extreme locality of control, the repre-
sentation of an expression may appear rather complex (section IV.4).

The second part of this chapter (sect. IV.3 and IV.4)

contains some technicalities. Some readers might prefer to

skip it.

IV.2 The Adams' computation model with data flow sequencing.[a]

The directed graph is used as a model of the computation.

"The nodes of the graph represent computation steps and the edges
represent transmission paths for data and control. An edge may
be thought of as a queue of data produced by one node and waiting
to be consumed by another. A computation step may be initiated
whenever each edge directed into that node of the graph contains
the amount of data required for the node to execute properly."

Fig. IV-1 displays some nodes which are activated whenever

there is an input value on each input edge. Then, output values

to be put on the output edges are computed according to a function f

associated with the node. "φ" is a notational device: placing "φ"

on an edge means placing nothing on this edge.

Fig. IV-2 displays some nodes having a more involved behavior.

Relatively to a node, an input edge may be in any one of two status:

it is either locked or unlocked. In fig. IV-2, the status of input

edges is given, for instance, as ULL , which means that port 1

is unlocked and that ports 2 and 3 are locked. The computation

step of a node is initiated whenever there is an input value on

each unlocked input edge. Output values are computed as before

according to an associated function f; moreover, a new status

for each input edge is determined according to another function g

associated with the node. The blocking capabilities of these nodes

permit the computations to be determinate.

---

a) We have taken the freedom to quote some passages of,
and to reproduce some figures from Adams [1].
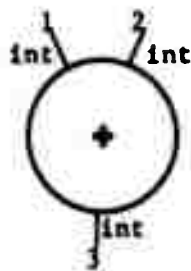
$$f: V_3 \leftarrow V_1 + V_2;$$

Fig. IV-1a
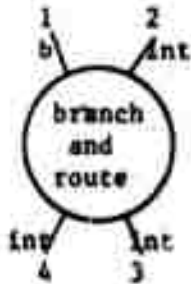
$$f: V_3 \leftarrow \underline{if}\ V_1 = \underline{false}\ \underline{then}\ V_2\ \underline{else}\ \varphi;$$
$$V_4 \leftarrow \underline{if}\ V_1 = \underline{true}\ \underline{then}\ V_2\ \underline{else}\ \varphi;$$

Fig. IV-1b
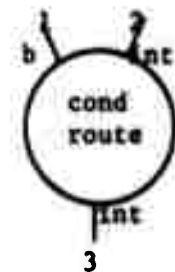
$$f: V_3 \leftarrow \underline{if}\ V_1 = \underline{true}\ \underline{then}\ V_2\ \underline{else}\ \varphi;$$

Fig. IV-1c

g: $ULL \rightarrow LUL$ $\underline{if}\ V_1 = \underline{true}$ f: $V_4 \leftarrow \varphi;$

$ULL \rightarrow LLU$ $\underline{if}\ V_1 = \underline{false}$ $V_4 \leftarrow \varphi;$

$LUL \rightarrow ULL$ $V_4 \leftarrow V_2;$

$LLU \rightarrow ULL$ $V_4 \leftarrow V_3;$

Fig. IV-2a
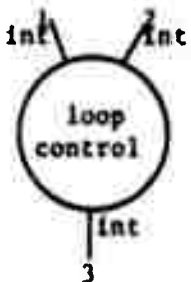
g: $UL \rightarrow LU$ f: $V_3 \leftarrow V_1;$

$LU \rightarrow LU$ $V_3 \leftarrow V_2;$

Fig. IV-2b   (from Adams [1])

Fig. IV-3 and IV-4 represent an iterative [1, p. 31] and

a recursive [1,p.32] factorial respectively. The latter figure

witnesses to the recursive character of a graph procedure:

> "When a node in a graph procedure represents a recursive
> call upon the procedure of which it is a part, a copy of the
> called graph procedure is created. Thus, during the execution
> of a graph program, an auxiliary graph referred to as the executing
> graph will be constructed. Initially the executing graph will
> consist of the main graph procedure G, with the initial data
> for the program placed on the edges of G. The initial data must
> be of the same type as the edges on which it is placed. Whenever
> a procedure node in the executing graph is ready for execution,
> a copy of the defining graph procedure will be created and added
> to the executing graph a) ; and when the procedure terminates,
> the created copy will be deleted. The executing graph can thus
> expand and contract dynamically during the execution of the program."

Remark. Such a situation may be regarded as an instance of the

realization of a contravariant functor (compare III-8 and IV-5).

In DCPL we will find the same notion of implementing in

space successive generations of a recursive procedure.

IV.3   Functional representation: the lambda-calculus b)

The discussion in this section and in the next one is

based on Curry [3]. The lambda-calculus provides a framework

in which functional expressions may be represented and evaluated.

---

a) Such a creation and addition of a graph procedure to
the executing graph is referred to in this thesis as an implementation
in space.

b) There are different lambda-calculi, each one having
its particular idea about what objects represent the same function.
However since in programming languages the emphasis is put to
the application of a function to an argument, and not to a function
itself, only one calculus is generally used (β-conversion lambda-calculus).

ITERATIVE
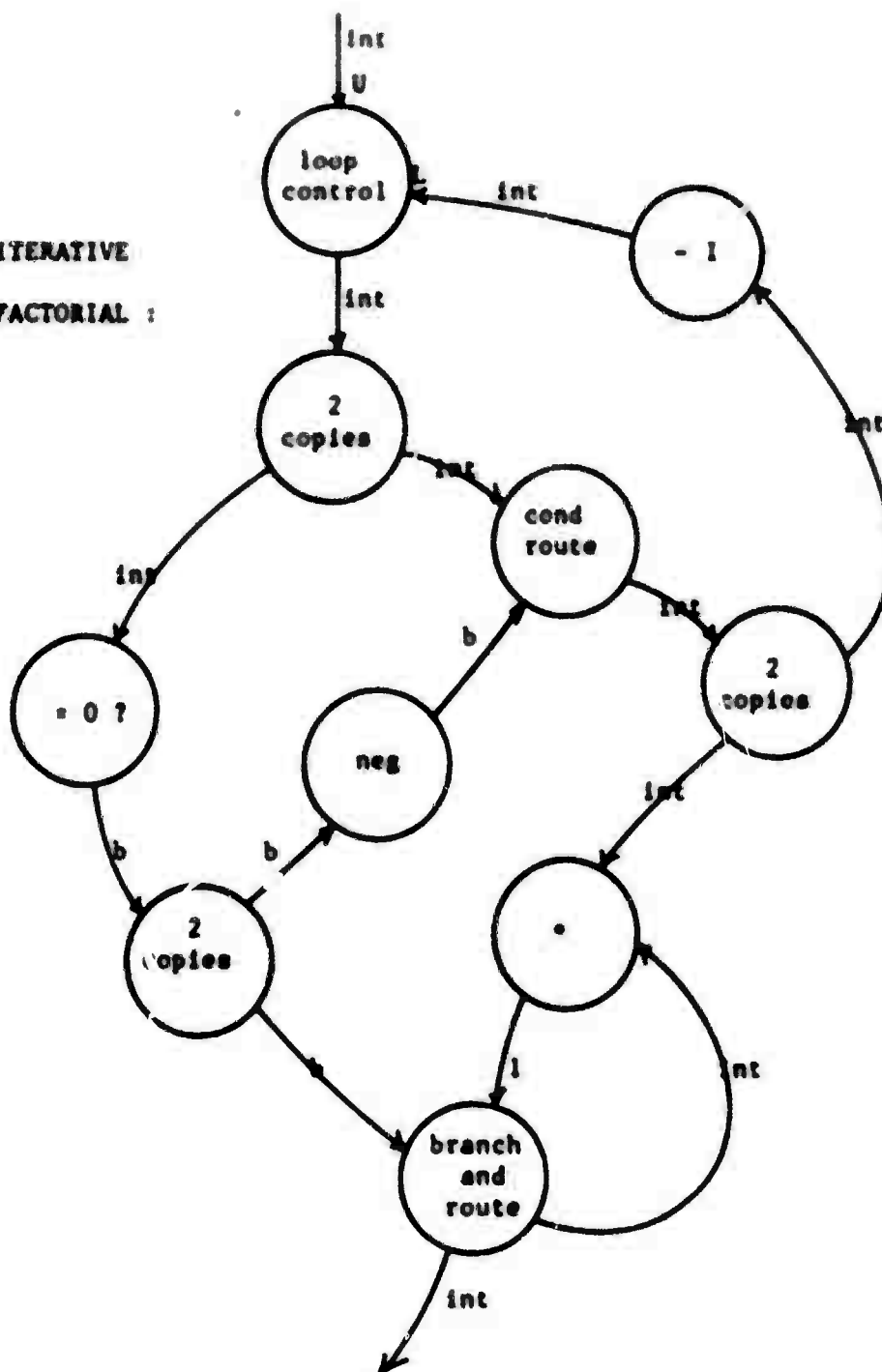
FACTORIAL :

Fig. IV-3 (Adams [1],p.31)
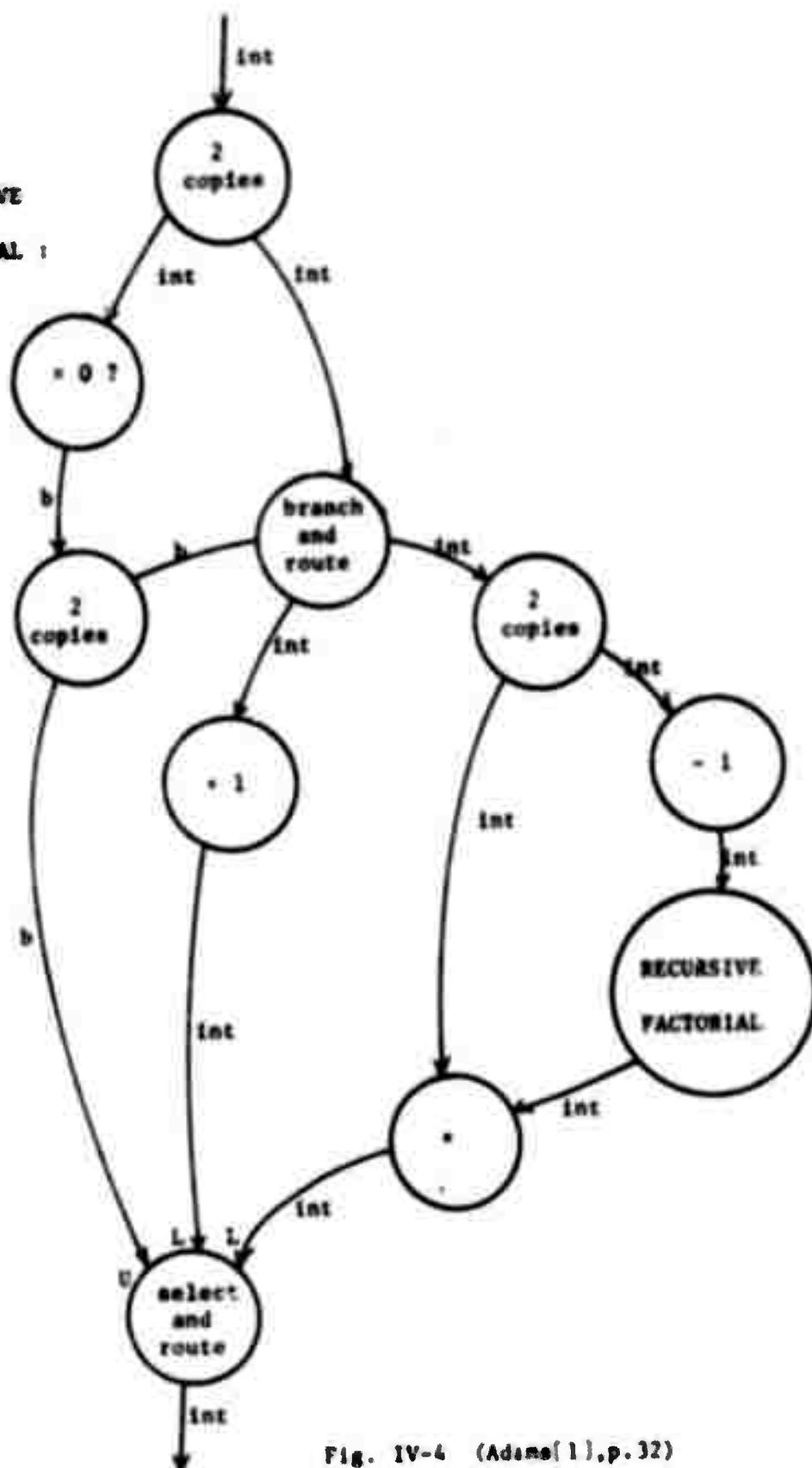
Fig. IV-4   (Adams[1],p.32)
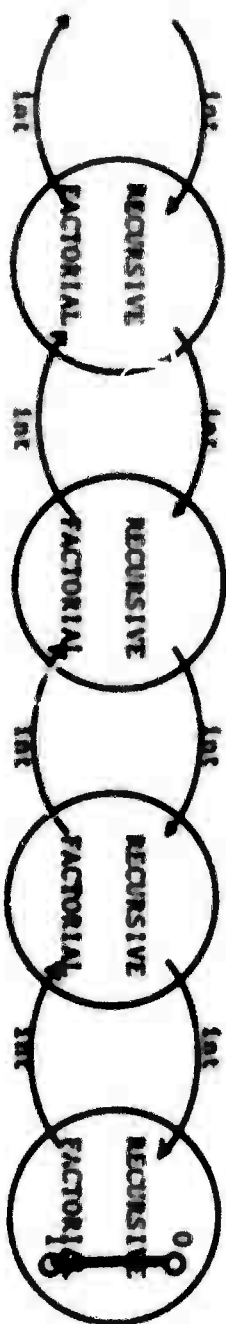
Fig. IV-5

Such evaluations present possibilities of concurrency and distri-
bution of control. Since Mc Carthy [18] , the lambda-calculus has
often been considered in connection with programming languages:
Landin [12,13,14], A. Evans [7], Morris [19].

IV.3.1 Variables in mathematics.

Let us consider with Curry the following mathematical statements:

(1) $(x + 1)^2 = x^2 + 2 x + 1$

(2) $\frac{d}{dx} x^2 = 2 x$

(3) $\int_0^3 x^2 \, dx = 9$

In statement (1) the variable x may be considered as
having an intuitive meaning: for any integer, for instance a,
we have the relation $(a + 1)^2 = a^2 + 2a + 1$.
We cannot interpret the use of the variable x in such a simple
way in (2) and (3): they do not enunciate any statement about
some object for which x stands. In fact, (2) and (3) state
some properties about a function, the square function. The use
of variables in these statements may be considered as only a
notational device. The use of variables may even be more explicit
with Church's Lambda notation (Church [2]); $\lambda x.x^2$ denotes then
the square function. If D and I_a^b denote respectively derivation
and integration between a and b, the statements (2) and (3) become:

(2') $D(\lambda x.x^2) = \lambda x.2x$

(3') $I_0^3(\lambda x.x^2) = 9$

The lambda notation allows us to consider a function, at least
conceptually, as an object: functions become part of the universe
of discourse, and statements about functions may be formulated.

IV.3.2 Functional representations in programming languages.

Functions appear naturally in programming languages whenever
some object is to be evaluated in some realization (see chapter III).
Generally statements about functions which are considered in
programming languages are very limited; with the exception, may be,
of some symbolic manipulation oriented programming languages,
functions are considered as far as they will be applied to some
arguments at some time. There is a notation to represent functions
and another notation for the application of a function to some
arguments, and there is a background machine which may evaluate
functional expressions. If such a background machine is to be
an abstract (or formal) one, a notation is not sufficient, it is
necessary to be able to represent a functional expression. By
representation we mean:

1. Any function may be represented as an object, better
   as an ob.

2. There is a combination called application which allows
   to represent in an explicit way the ob obtained by
   combining, with the application, a function to its arguments.

3. There is an abstract machine which can evaluate the
   previous ob and produce the representation of the resul-
   ting object.

IV.3.3 An abstract reducing machine.

We are interested here in abstract machines performing reductions since these machines are particularly simple: when applied to some object, they look for any component corresponding to a given pattern and replace it by an associated component . A macro-processor, an interpreter using a simple precedence grammar (Wirth [28]) might be considered as reducing machines.

IV.3.3.1 Substitution versus replacement.

We present these notions pictorially: in fig. IV-6 an ob X is substituted for an atom x in an ob A; the result is an ob B.(We may notice that

  1. x is supposed to be an atom

  2. each occurrence of x is replaced by an occurrence of X.).

In fig. IV-7, a component X of an ob A is replaced by an ob Y; the result is an ob B. (Note:

  1. X is not necessarily an atom, but an ob

  2. Even if there are several occurrences of X in A, only the considered occurrence is replaced.).

IV.3.3.2 Reduction rules.

Let us consider two obs A and B and the reduction rule $A \rightarrow B$. Let X be an ob having A as a component. We say that X may be reduced in Y, and we note $X \triangleright Y$ if Y is obtained by replacing, in X, the component A by B. We note $X \doteq Y$ whenever $X \triangleright Y$ or $Y \triangleright X$ and $\geq$ and $=$ represent respectively the quasi-ordering (symmetric and transitive) and the equivalence generated by $\triangleright$ and $\doteq$.
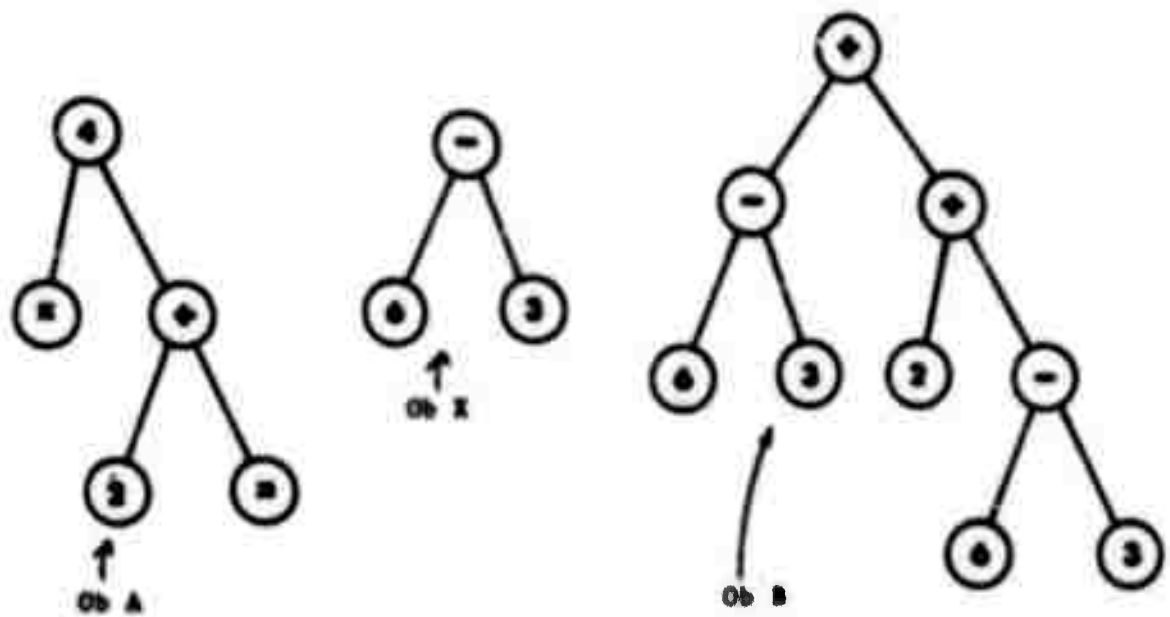
Fig. IV-6
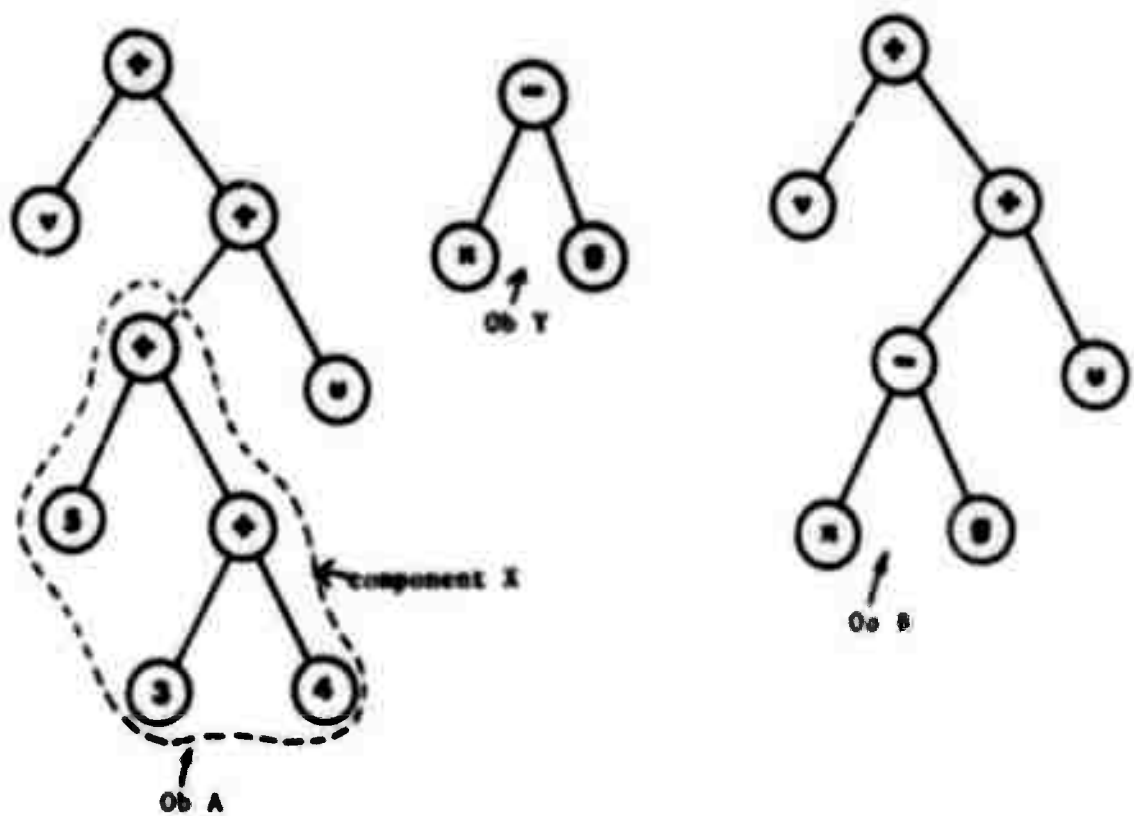
Fig. IV-7

The same definitions may be given with a set of reduction rules $\left\{ A_i \to B_i \right\}_{i \in I}$. In summary:

$X > Y \longleftrightarrow$ there is in the ob X a component A_i whose replacement by the corresponding B_i produces the ob Y

$X \doteq Y \longleftrightarrow$ either $X \triangleright Y$ or $Y \triangleright X$.

$X \geqslant Y \longleftrightarrow$ there is a sequence of objects X_0 , X_1 ,... X_n , $X_0 \equiv X$ and $X_n \equiv Y$ such that $X_0 \triangleright X_1 \ldots \triangleright X_n$ ($\equiv$ denotes the identity of obs)

$X = Y \longleftrightarrow$ there is a sequence of objects X_0 , X_1 ,... X_n such that $X_0 \equiv X$ and $X_n \equiv Y$ and $X_0 \doteq X_1 \doteq \ldots \doteq X_n$
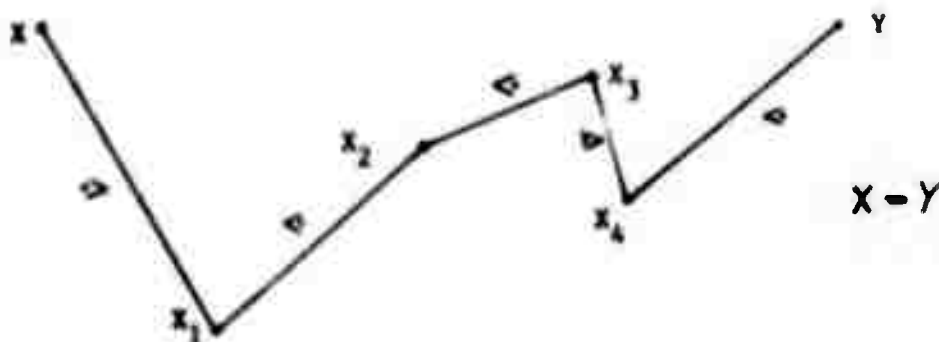
Fig. IV-8

IV.3.3.3 Reducing machine.

Whenever a set of reduction rules is given we can consider the following reduction process, to be applied to any ob X:
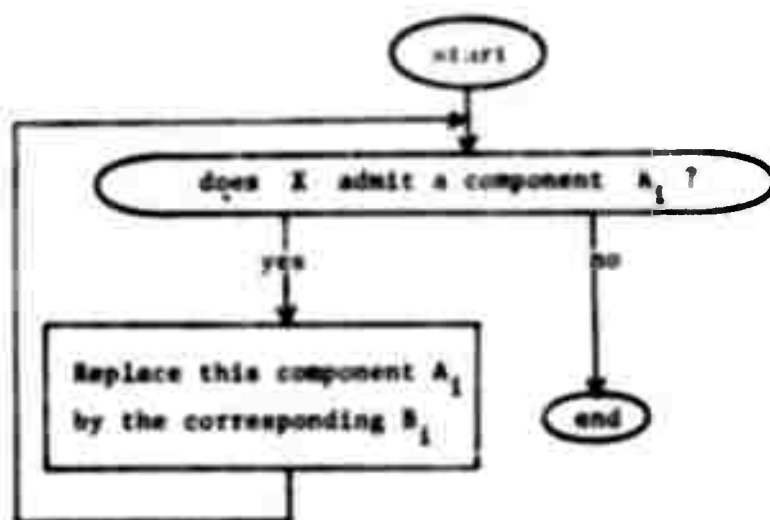
Fig. IV-9

Ob iously, such a process is not in general deterministic. Moreover, applied to an ob X it may or may not stop. If it does stop, producing an ob X_0 , there is no reduction which may be performed on X_0: X_0 is said to be a normal form of X.

We are interested in abstract reducing machines which are deterministic to some extent, namely in machines verifying the following condition (Church-Rosser): if $X = Y$ there is an ob Z such that $X \gtrless Z$ and $Y \gtrless Z$.
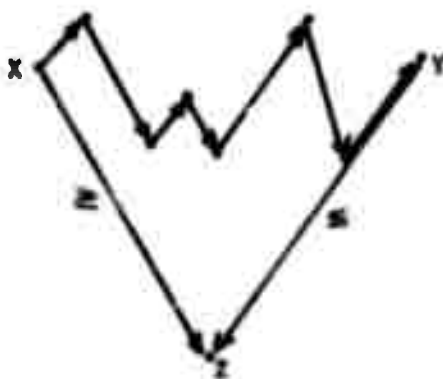
Fig. IV-10

Indeed such a condition guarantees that whenever the machine
stops when applied to an ob X, it produces the same normal form
(however in some simulation the machine may not stop). Such a normal
form may be considered as the "value" of X, the reduction machine
being therefore an evaluating machine.

IV.3.3.4 A trivial example.

Let us consider a trivial representation of simple computa-
tions without variables:

- to each value v in the universe corresponds a combinator of
  degree zero $\hat{v}$ and to each operator of degree n, opr, a combina-
  tor of degree n, $\widehat{opr}$. We call CL_o the class of obs which may be
  constructed with these combinators. For any operator opr and
  for any values v_1, v_2, ... v_n producing $opr(v_1, v_2, \ldots, v_n)$
  we have the reduction rule :

For instance if there is one operation, the addition, and
integers as values, the reduction rules will be of the form:

etc ...

Any sum will be trivially evaluated by our machine (fig. IV-11).
The Church-Rosser condition is here obviously verified; the machine
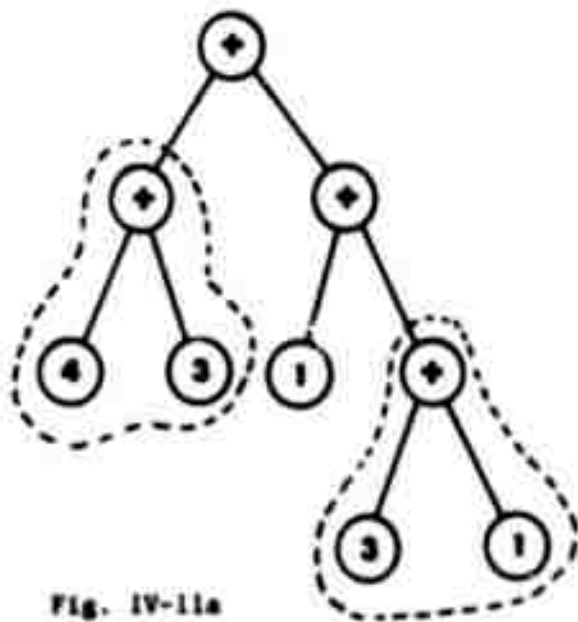
Fig. IV-11a
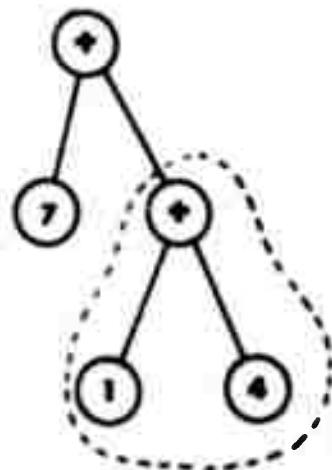
Fig. IV-11b
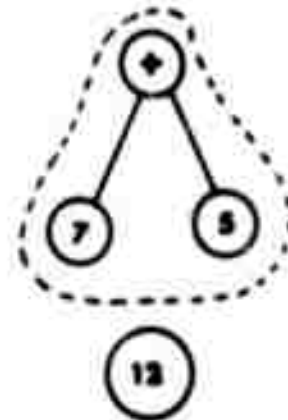
Fig. IV-11c

Fig. IV-11d
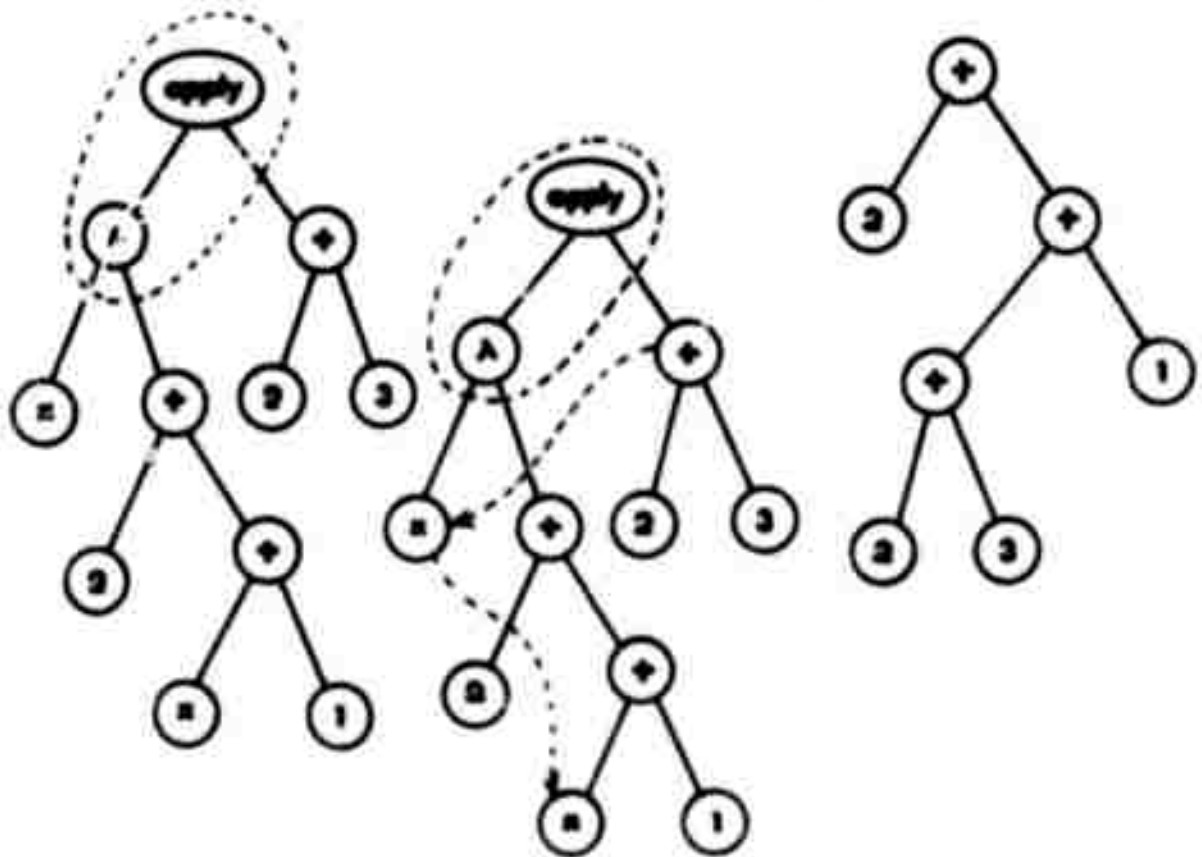
Fig. IV-11

is deterministic and has a distributed control. The result $\widehat{12}$,
as a combinator, is the normal form of the given expression and
may be considered as being the value of the expression. The
relation of equality we have defined corresponds to the tradi-
tional meaning of equality of sums.

IV.3.4 Evaluation of λ -expressions.

Let us consider, together with the previous combinators,
a new atom called a formal variable x. Any ob M containing x
may be considered as a function $CL_0 \rightarrow CL_0$ (CL_0 is the class of
obs obtained with the combinators associated to values and opera-
tions of the universe): indeed, to any ob A in CL_0 we may associate
the ob obtained by substituting A for x in M, which may be denoted
by $[A/x]M$ and the function $A \rightarrow [A/x]M$ may be denoted by $\lambda x.M$ ([2]).
The operation $M \rightarrow \lambda x.M$ is called abstraction relatively to the
formal variable x

We can represent such a function as an ob in the following
way:

1. There is a combination of degree 2, called abstraction
and denoted by a combinator λ. Whenever abstraction is applied
to two obs, the first one must be a formal variable. A written
notation for such a combination is $\lambda x.M$; the ob is

2. There is a combination of degree 2 called application,
and denoted by the combinator 'apply'. A written notation for

is MN .

3. There is an abstract reducing machine defined by the reduction rules generated by the following reduction scheme: Whenever M and N are two obs and x is a formal variable, $(\lambda x.M)N \rightarrow [N/x]M$, or graphically:
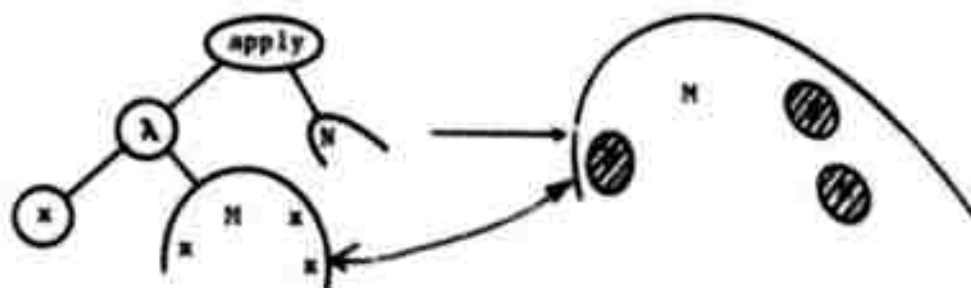
Fig. IV-12

It is proven that these reductions verify the Church-Rosser condition (Church-Rosser theorem, see Curry's [3,chapter 4])

We now have a machine which can evaluate expressions such as:

$(\lambda x.(2 \cdot (x + 1)))(2 + 3)$ (fig. IV-13)

or: $(\lambda x.x + 1)((\lambda x.2 + x) 3)$ (fig. IV-14).

In fact, very general computations may be expressed with such a model (see Landin [12]).

Remark 1: The reduction process may not stop when applied to some peculiar obs like: $Y \equiv \lambda f.[(\lambda h.f(hh))(\lambda h.f(hh))]$ called "paradoxical combinator" (see Morris [19]).

Remark 2. The reduction machine is not in a strict sense deterministic: we can only state that whenever the machine stops, it produces the same value; however some simulation may produce a value
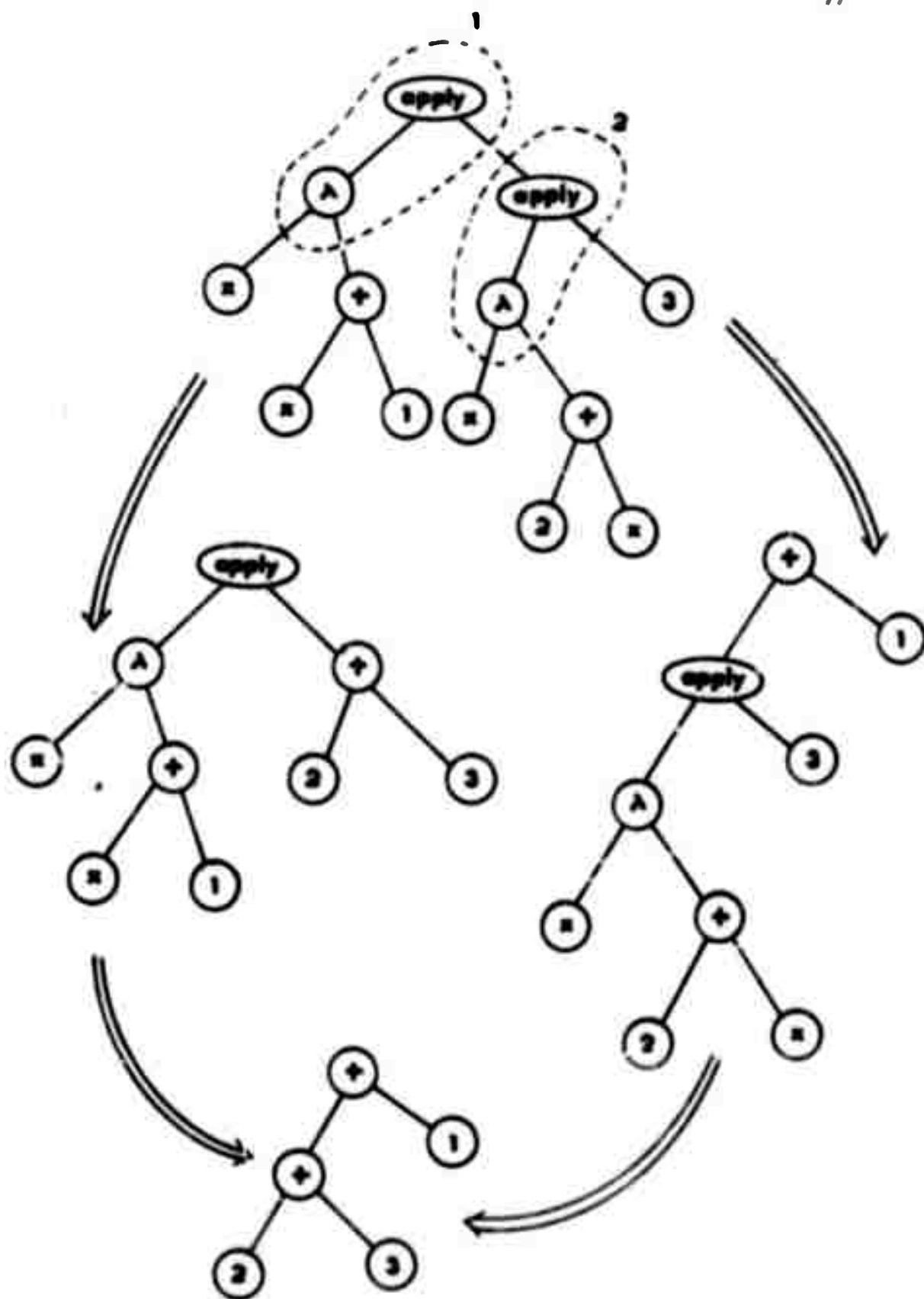
Fig. IV-14

and some other simulation may go for ever. Example: any expression
$(\lambda x.2)A$, A being any ob, may be reduced to 2 since x not occurring
in 2, $[A/x]2 = 2$. However if $A \cong Y$ (the paradoxical combinator),
the machine may undefinitely reduce Y. Therefore a reduction of
$(\lambda x.2)Y$ may not terminate.

The machine may be modified in order to be deterministic:
a reduction such as the one of
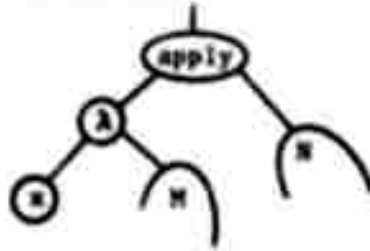
may only be applied when the argument N is not reducible any more.
Then $(\lambda x.2)Y$ (Y being the paradoxical combinator) becomes
a computation which does not terminate in any case.

In Mc Carthy [18] and Landin [12] a sequential machine
is used (the order of the reduction is uniquely determined).
The evaluation is therefore deterministic.

IV.4 Another functional representation: Curry's combinators.

In the previous section, an abstract machine evaluating
λ-expressions was supposed to be able to perform replacements
and substitutions. We consider a replacement as a simple and basic
transformation; however we view a substitution as a much more
complex transformation. The question then arises whether it is
possible to represent functional computations with an abstract
machine which is only able to perform replacements. This is possible
with Curry's combinators and moreover formal variables are not
needed any more in the representation. Such a property is concep-

tually important enough to legitimate this section of the thesis.
In fact the study of combinatory logic has been a point of departure
from which many ideas expressed in this dissertation have emerged.

IV.4.1 Notations and representations.

In this section (and in this section only), we suppose that
there is only one combination called application. Whenever F and X
are two obs, we represent the ob obtained by applying F to X by FX;
in a graphical notation:

Whenever parentheses are not used, association is to be
performed on the left (in this section only). So FXYZ ≡ (FX)YZ ≡ (FXY)Z

Any combination different from the application is represented
by an atom; for instance, if A denotes addition, A 2 3 represents
(A 2) 3 : A may be considered as an operation of degree 2, (A 2) as
an operation of degree 1 . Fig. IV-15 represents the ob:

$$A (A 2 (A 3 5)) 1$$

IV.4.2 The combinators K,I,♦ .

Let us suppose that together with the combinators associated
with values and operations we have the following combinators: K,I,♦.
To each of these combinators is associated a reduction scheme:

For any obs X,Y,Z,T we have:

Fig. IV-15

rule (K) :   K X Y → X

rule (I) :   I X → X

rule (0) :   0 X Y Z T → X(Y T)(Z T)
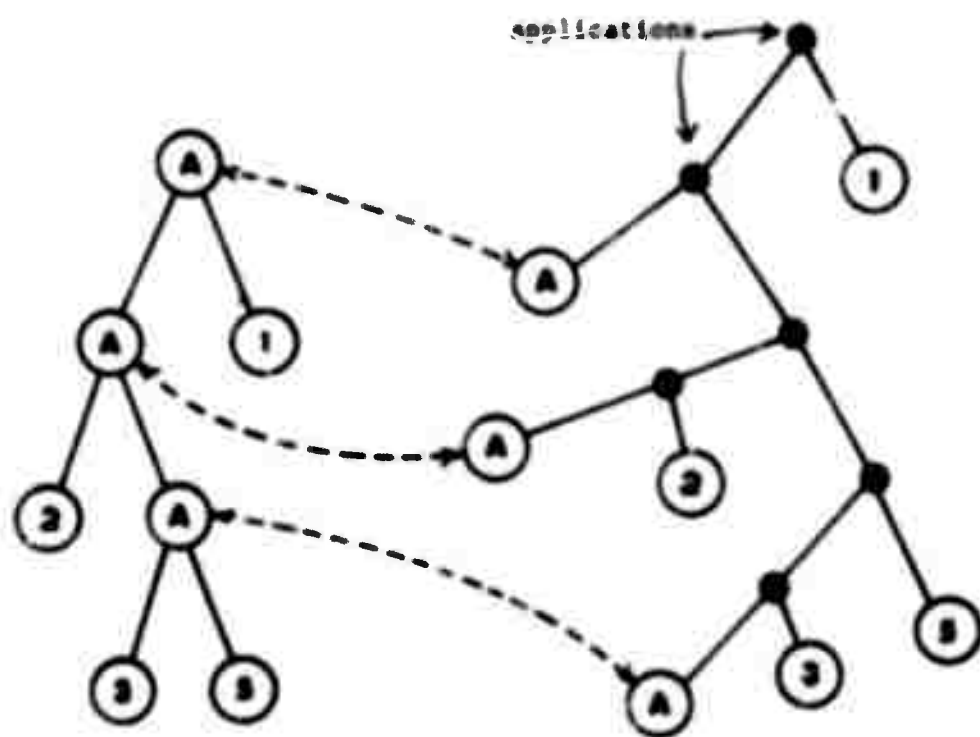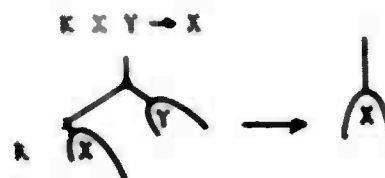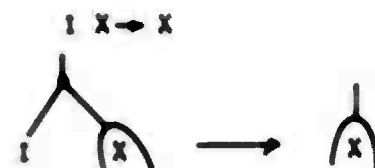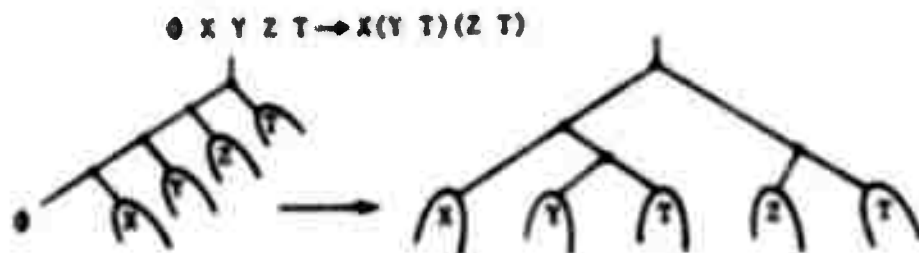
As we have seen in section IV.3 these rules define an abstract reducing machine.

IV.4.3 An example.

A and M denoting the addition and the multiplication, let us consider the ob:

$$F = 0 A (0 M I I) (0 M (K 2) I)$$

Let us reduce the ob F a , with 'a' standing for any integer:

F a = 0 A (0 M I I) (0 M (K 2) I) a

0 X Y Z T

(rule 0)

X (Y T) (Z T)

A (0 M I I a) (0 M (K 2) I a)

0 M I Z T 0 M K I T
0 X Y Z T 0 X Y Z T

rule 0 rule 0

X (Y T) (Z T) X (Y T) (Z T)

A (M (I a) (I a)) (M (K 2 a) (I a))

rule (I) rule (K)
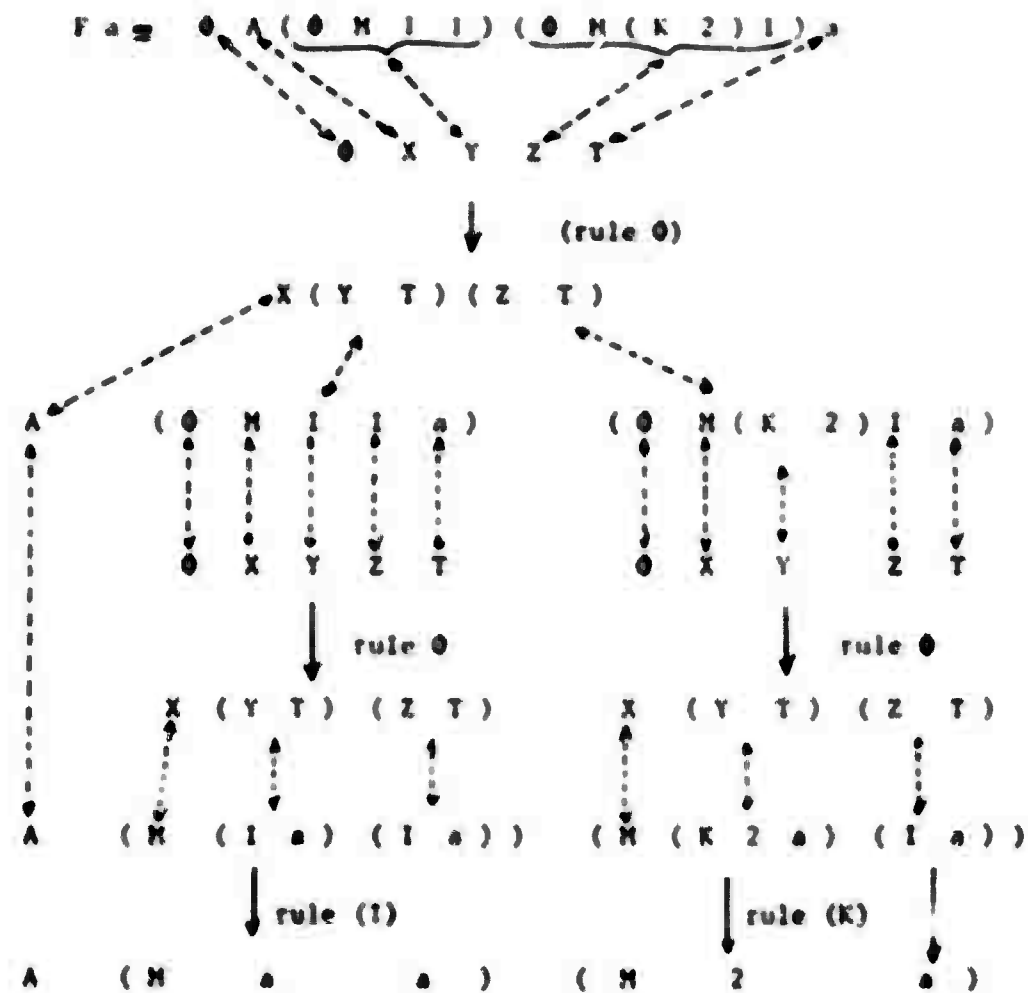
A (M a a) (M 2 a)

We have F a ▷ A (M a a) (M 2 a) . So, F is a function
which, applied to any integer a, produces A (M a a) (M 2 a) .
F is therefore an ob representing λ x . A (M x x) (M 2 x)
or, with usual notations, λ x.(x · x) · (2 · x) and F does not
contain any variable.

IV.4.4 An algorithm.

We may wonder if every function may be represented in this way: Let us consider an ob M which may contain a formal variable x. Is there an ob F which does not contain any variable, representing $\lambda x.M$? The answer is yes. In order to simplify the discussion, let us assume that M only contains operations of degree 2. Let us call [x]M the ob F we are looking for. We have the following algorithm:

> if M does not contain x then [x]M = K M
>
> > else if M is x then [x]M = I
> >
> > > else .
>
> M has the form:
>   <operator> <left operand> <right operand>
>
> and:
> [x]M = 0 <operator([x]<left operand)[x]<right operand>)

This algorithm, applied to $\lambda x.A(Mxx)(Mxx)$ produces in fact F.

IV.4.5 B,C,W,S,K.

If M contains several variables x, y, z , then $\lambda x\, y\, z\, .\, M$ is obtained by forming   [x] ([y] ([z] M)) .

We have particularized our discussion with operations of degree 2. However any function may be represented with:

B   (B):   B f g x = f(g x)

C   (C):   C f x y = f(y x)

W   (W):   W f x = f x x

or, alternatively, with:

S   (S) $S\ f\ g\ x = f\ x(g\ x)$

K   (K) $K\ c\ x = c$

With any of these combinators the abstract machine verifies the Church-Rosser condition. The remarks at the end of IV.3 are still valid here.

Remark. Each of the previous combinators may be considered as a lambda-expression. For instance, we can take:

$$B = \lambda f\ g\ x\ .\ f(g\ x)$$

$$C = \lambda f\ x\ y\ .\ f(y\ x)$$

$$W = \lambda f\ x\ .\ f\ x\ x$$

Then B,C,W may be considered as a "base", any lambda-expression being expressible as an ob constructed with B,C,W.

This discussion is the last of our preliminaries. We are now going to study DCPL.

PART TWO

D C P L

A DISTRIBUTED CONTROL PROGRAMMING LANGUAGE

CHAPTER V

P.C.P.L.

A DISTRIBUTED CONTROL PROGRAMMING LANGUAGE

V.1   The general frame.

V.1.1   A computation as a formal object.

In DCPL, any computation is considered as a formal object,
an ob (see section III.1). In the computing machine, such an ob
is actualized as a tree whose nodes, representing combinators,
are automata, and whose edges are directed channels of information,
directed from "son" to "father" (fig. V-1). A node may have zero,
one or several sons; the number of sons it has is called its degree
(fig. V-2).

The edges being directed channels of information, each node
may send a message up the tree; such a message is called a notice
(fig. V-3). Whenever a node receives a notice, it may either
pick it up or pass it along upward.

The tree is considered to be in some space in which each
node has an address. There is a communication system allowing any
node to send a message to any other node whose address it knows;
such a message is called a reply. A reply contains explicitly
the address of the addressee.

Whenever a node A wants some information from ancestor B,
A sends up a notice with its own address (fig. V-4); such a notice
is called a request. A request is a notice containing the address

Fig. V-1

Fig. V-2

Fig. V-3
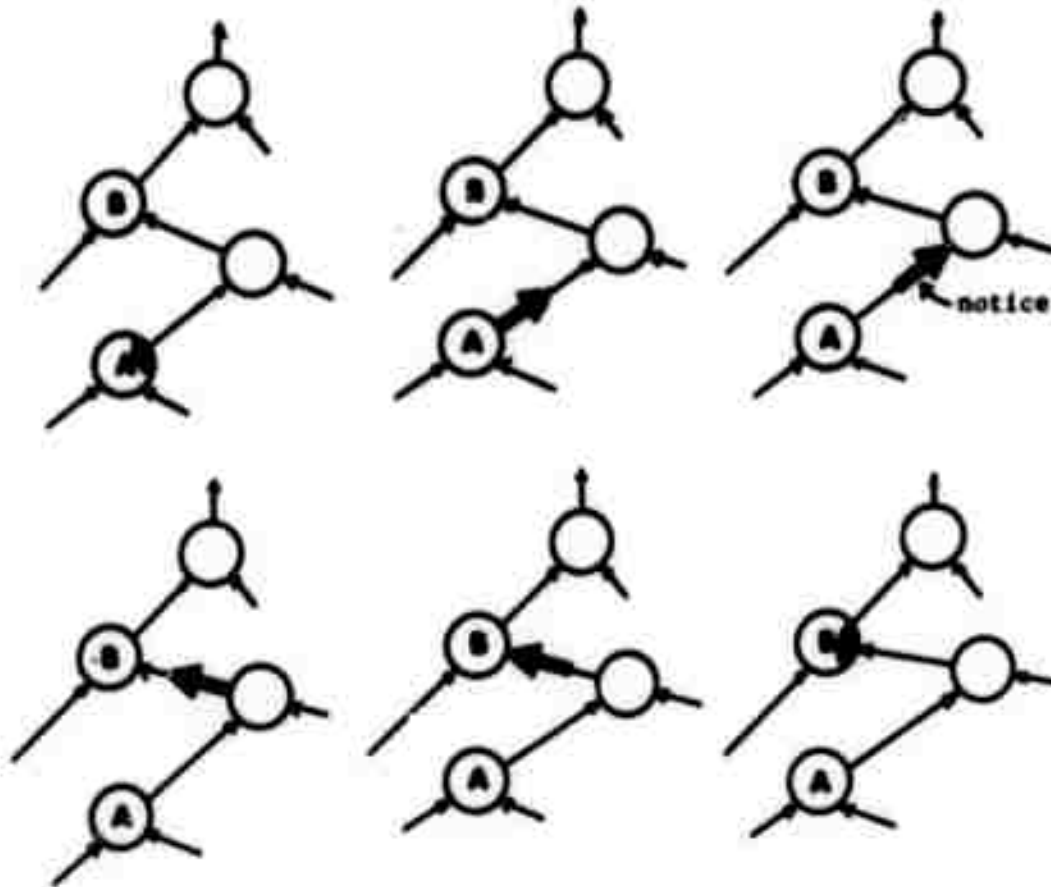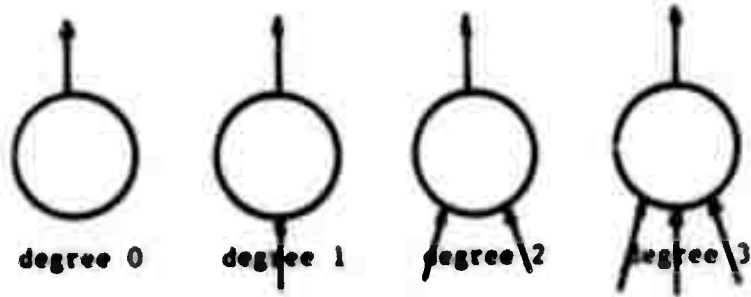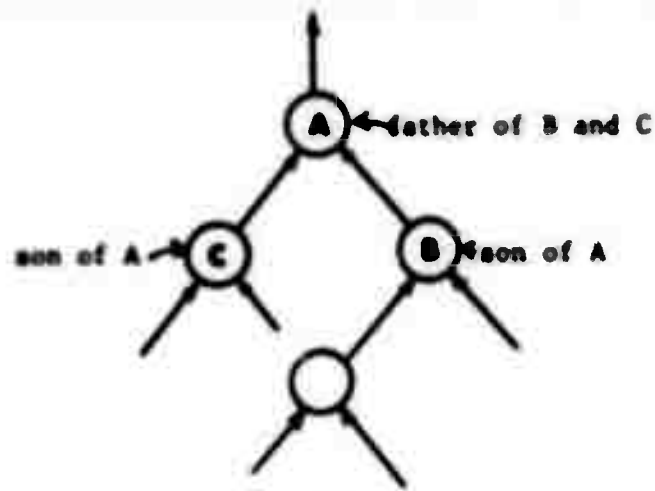
Fig. V-4

Fig. V-5

of the sender to which some reply is to be made. A notice which
is not a request is a simple notice. On receiving the request
sent by node A, the node B may send back the required information
(fig. V-5).

V.1.2 Values and operations: their representation.

As it has been already noted in the introduction (chapter 1),
we assume that there is a universe of values structured in classes
or types, and mappings from some classes to some possibly different
classes which are actualized by operations, and we leave the list
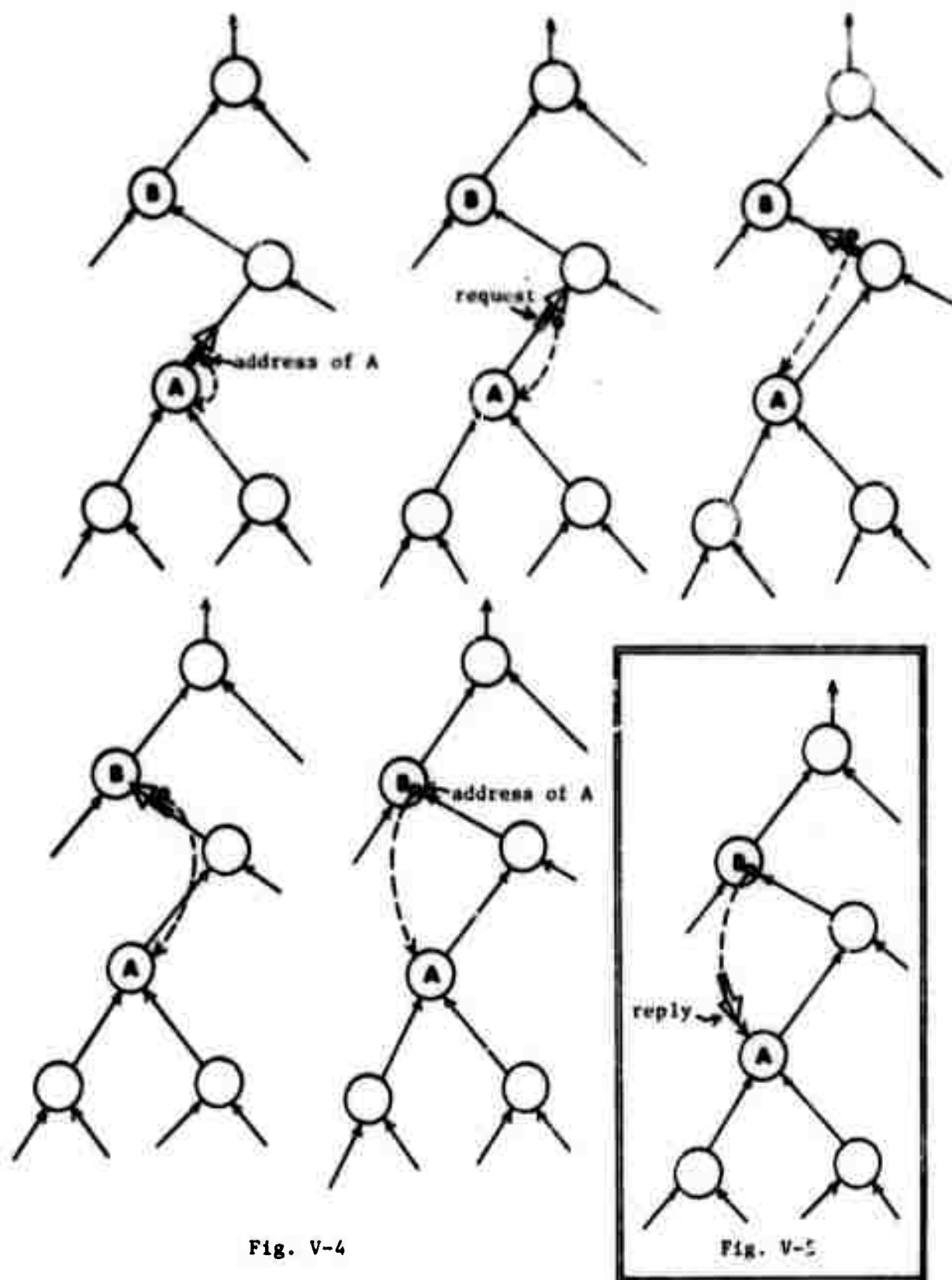of types and operations open-ended. We require however that:

1. to each value in the universe corresponds a specific
combinator, represented by a value-node. Spontaneously a value-node
sends up the tree a simple notice containing the value it represents;
the node then disappears (fig. I-3, chapter I).

2. to each operation in the universe corresponds a specific
operator represented by an operation-node, which has the degree
of the operation it represents. An operation-node waits until it
receives a value from each of its sons; it performs, then, the operation
it represents on these values.The result is sent up the tree, and
the operation-node disappears(fig. I-4). Fig. I-5, in the introduc-
tion, shows how a simple expression without variable may be evaluated.

3. to each ob corresponds a value in the universe,which
may be considered as a description of the ob. As a result, a descrip-
tion of an ob may be sent as a message from some part of a computation
to some other part, where the described ob may be built and imple-
mented in space, extending the tree (fig. V-6).

description
of an ob B

component b
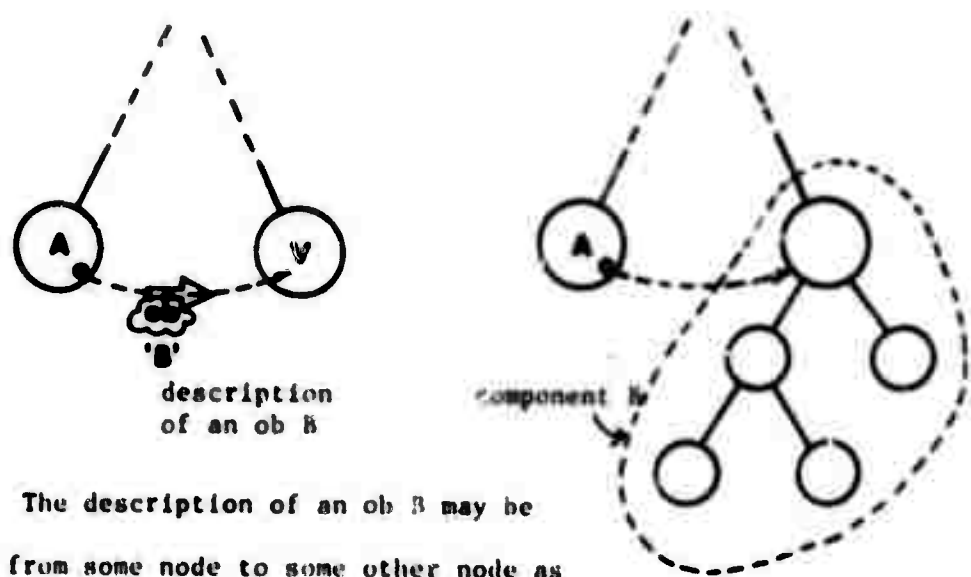
The description of an ob B may be
sent from some node to some other node as
a value. When received the ob.B may be implemented in space.

Fig. V-6

As a result of these behaviors, the tree representing a computation may grow and shrink during its life.[a]

V.1.3 The other combinators.

Together with the combinators associated to the values or operations, DCPL uses many other specific combinators:

Triadic combinators (degree 3):

'→'	(setdown)
'←'	(setup)
'*IF*' or '|'	(if)

Dyadic combinators (degree 2):

'*NEW*'	(new)
'*LAMBDA*'	(lambda)
'*MU*'	(mu)
'↑'	(sendup)
'↓'	(sendown)
'□→'	(trigger)
'*IMPL*'	(implement)
'o'	(compose)
'*OR*'	(or-event)
'*AND*'	(and-event)

---

a) Such a behavior is similar to the behavior of the executing graph used in the Adams' model (IV.2). Instead of graphs, the model described here manipulates trees.

<u>Degree zero combinators</u>:

$\emptyset$ (null ob)

a whole class of variables: X, Y, JOHN, VAR , ...

a whole class of alpha-variables: αX, αY, $\alpha JOHN$, αVAR , ...

The function and the behavior of each of these nodes will be progressively presented in the sequel.

V.2   <u>About the syntax</u>.

The tree-structured representation is fundamental in DCPL. However, it is useful to have a linear representation easier to manipulate. Rather than a straight parenthesization or a heavy syntactic apparatus, we prefer the use of a few replacement rules: programs gain a familiar appearance without any syntactic freezing. The following rules determine a one-to-one mapping between the tree-structured and the linear representation.

<u>Rule 1</u>. Parentheses may be used freely and no distinction is made between parentheses and brackets.

<u>Rule 2</u>: Any combinator of degree 2 may be infixed. Association is to be done to the right, unless explicitly specified through the use of parentheses. For instance $X+Y+Z$ is to be structured as $(X+(Y+Z))$. As a consequence, whenever two combinators of degree 2 are not ordered by parentheses, the latter is, on the tree structure, a descendant of the former: <u>NEW</u>   X .... † ....   has the following tree

structure: whereas (<u>NEW</u> X ...)† ... represents
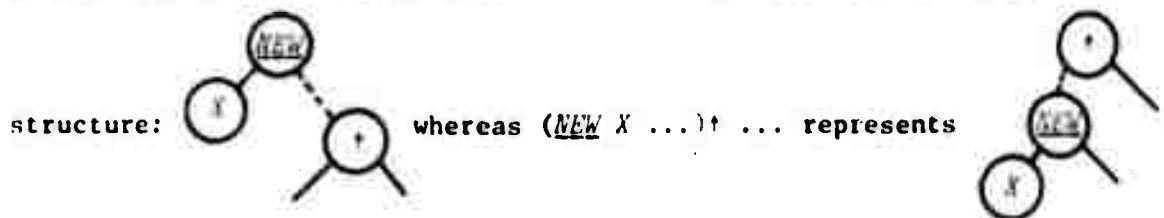
Rule 3. Much like in Algol where   Integer X,Y,Z;   is used instead of   Integer X; Integer Y; Integer Z; , one can factorize combinators:

NEW X,Y,Z   X+Y+Z   represents:
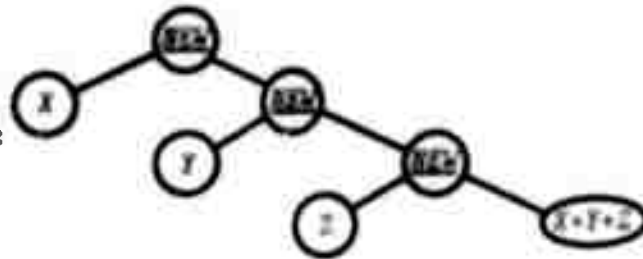
Rule 4. Combinators of degree 3 may be infixed between their first two arguments, a semi-colon separating the second argument from the third. Rule 2 is then applicable to such combinators. Some examples:

5+X;X+2   represents   and   PRED|EXPR1;EXPR2   represents

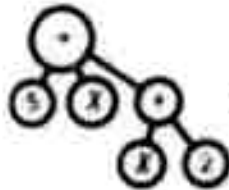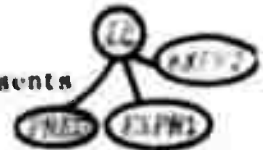The interest of this rule is sharpened when it is used repetitively on a nested structure:
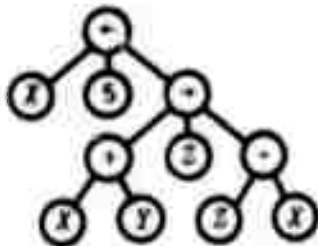
and
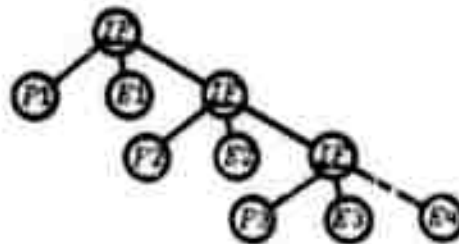
may be represented respectively by   X+5;(X+Y)+Z;Z-X   and by P1|E1;P2|E2;P3|E3;E4   or, with a non-necessary pair of parentheses, (X+5;(X+Y)+Z;Z-X)   and   (P1|E1;P2|E2;P3|E3;E4) .

Rule 5. As it will be discussed in the sequel, the application of a function to an argument or to a parameter is represented through the use of '↓' ('sendown') or '↑' ('sendup'). However, in order to conform to traditional notations, we allow expressions such as   F(A,B,C) , F(A,B,C;D,E) , F(;D,E) :
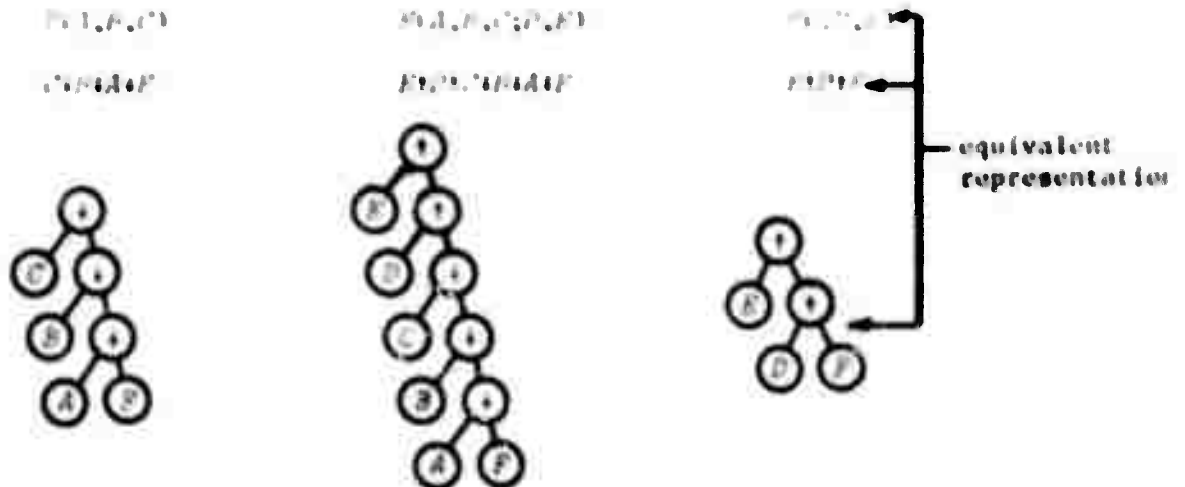
Remark. Application of rule 5 may be misleading: $F(A,B,C)$ is an ob; (A,B,C) is a notational convenience which has no meaning by itself. It does not represent any ob.

An example:

According to these rules, the expression

$$NEW \quad X,Y,Z \quad (X \leftarrow 5;(X+Y) \rightarrow Z;Z+X)+(Y+X+1;X)$$

is the following ob:
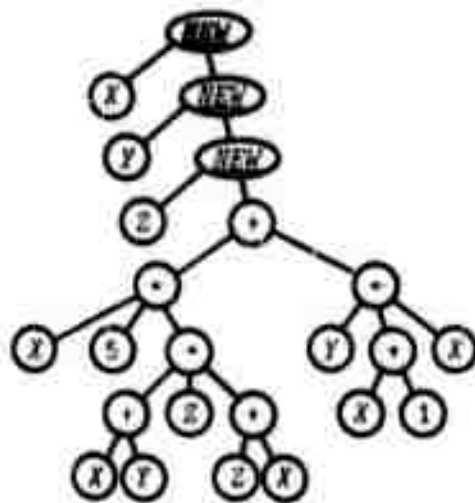
This is as much as is necessary to know about the "syntax" of DCPL.

V.3   The binding process.

In DCPL a system of mutually bound variables determines communication paths. The process which builds these paths is called the binding process. The binding process superimposes a graph structure to the tree structure which represents a computation.

A communication path leads from a "binder-node" to a variable-node. If (X) is a variable-node, (•X) a). (X•) . (USE X) .
(LAMBDA a) and (NU X) are possible corresponding binder-nodes.
Any variable-node sends spontaneously a request containing

---

a) Note: '•' is a combinator of degree 3, whose second argument must be a variable. However, the construct

is considered as being one node (•X) . In fact, in the

following list, the constructs on the left are considered to represent the constructs on the right:
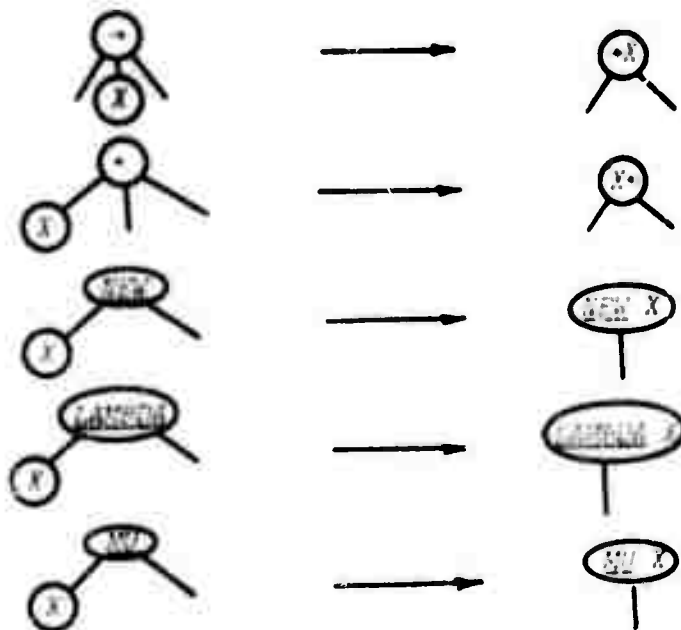
Its address and its name (for instance 'J'), up the tree (fig. V-7).
Such a request is picked up by the first corresponding binder-node
encountered. The variable-node is then bound by this binder-node:
a communication path leads from the binder-node to the variable-node.

A binder-node may bind several variable-nodes. Whenever
a binder-node receives an argument, it sends a copy of this argument
to each variable-node it binds (fig. V-8).

V.3.1 The binder-node setdown: $\cdot X$.
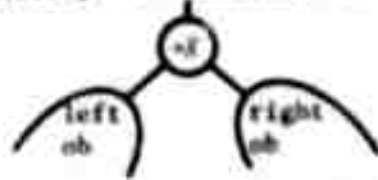
I being an identifier, $\cdot X$ combines two components:

$\cdot X$ may only pick up the requests reaching it from the right (fig. V-9).

On receiving a value simple notice on the left, setdown sends
a copy of this value to every variable-node it binds, on the right.
The scope of $\cdot X$ is therefore limited to the right component; $\cdot X$ may
be considered as assigning the value produced on the left to the
variable-nodes X in its scope (fig. V-10).

On receiving a value simple notice on the right, $\cdot X$ passes it
along up the tree and disappears (fig. V-11).

Example 1:

The evaluation of the expression $\cdot X \cdot I.X.$ is displayed in
fig. V-12.

Example 2:
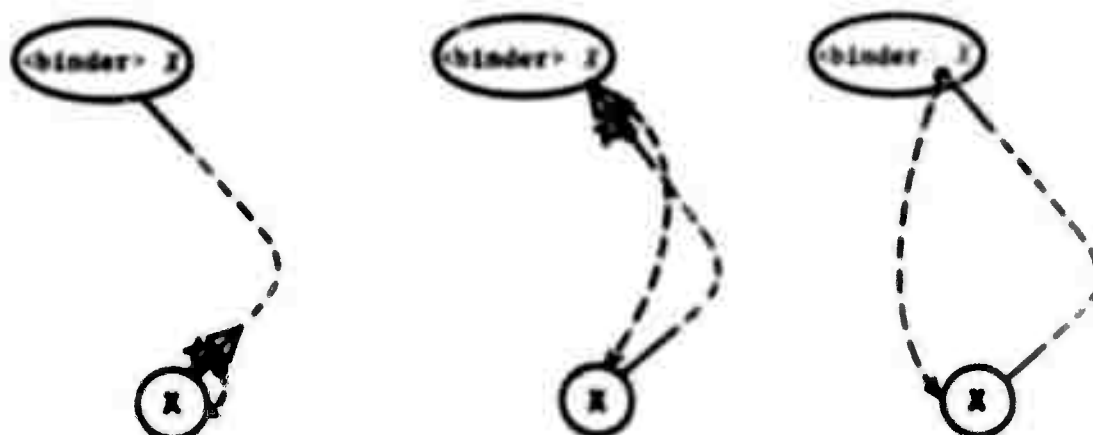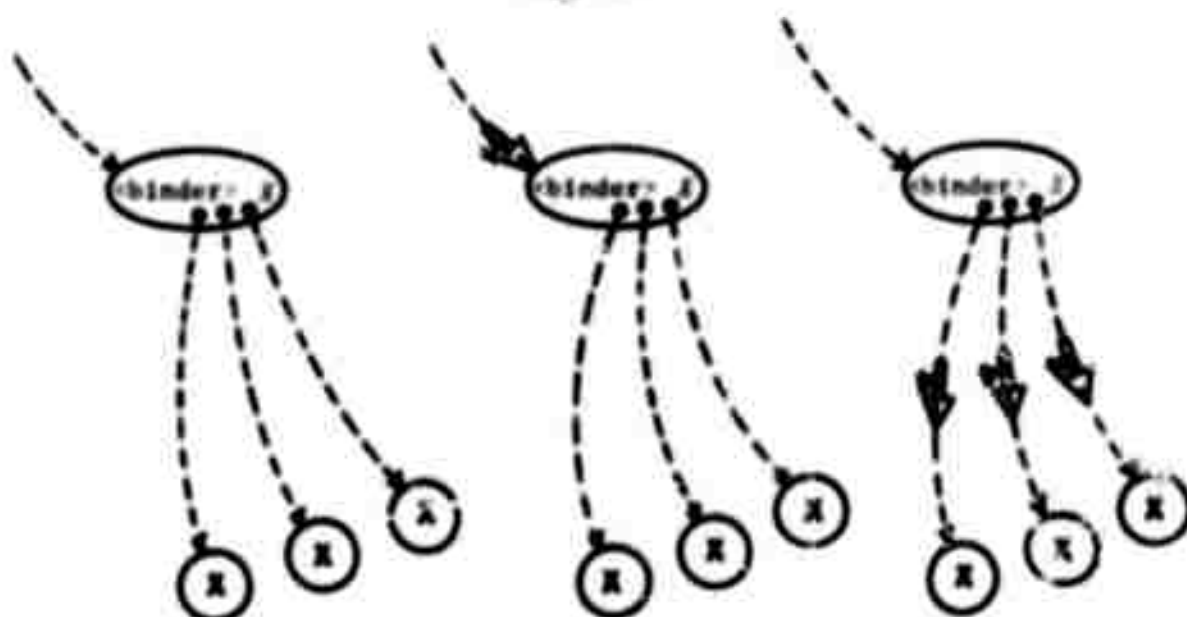
The evaluation of the expression $\cdot X \cdot I.X.I.$ is displayed
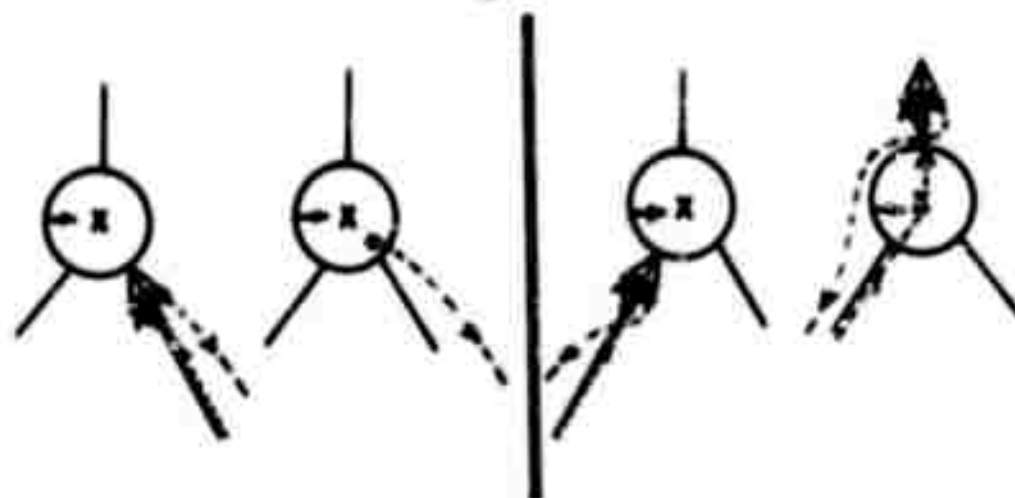in fig. V-13.

Fig. V-7

Fig. V-8

Fig. V-9
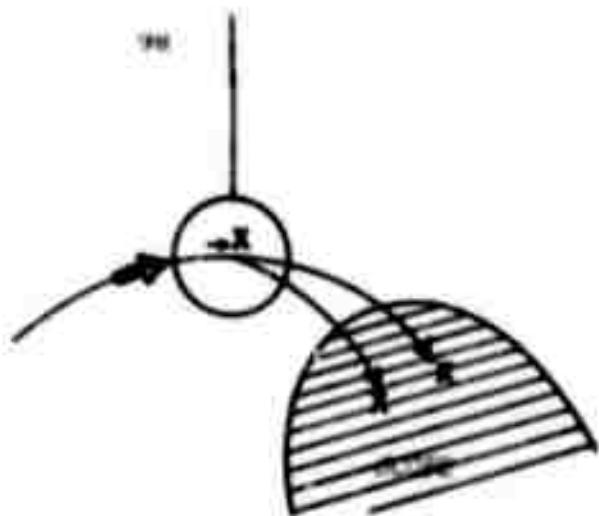
Fig. V-10
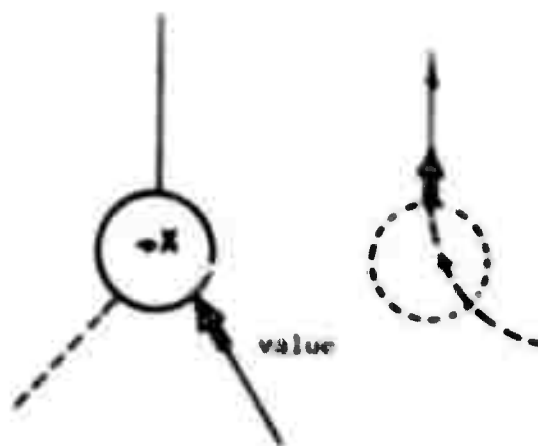
Fig. V-11
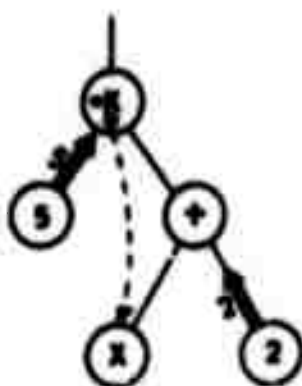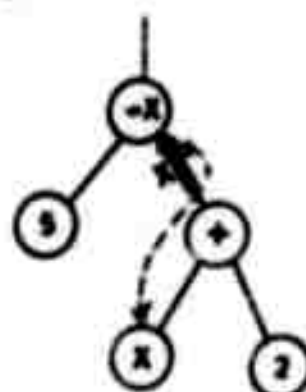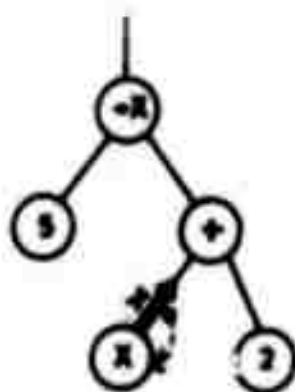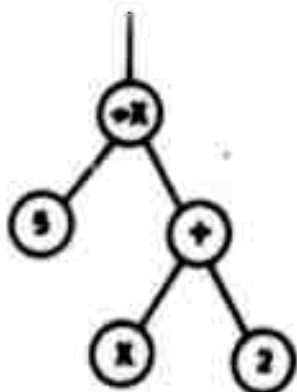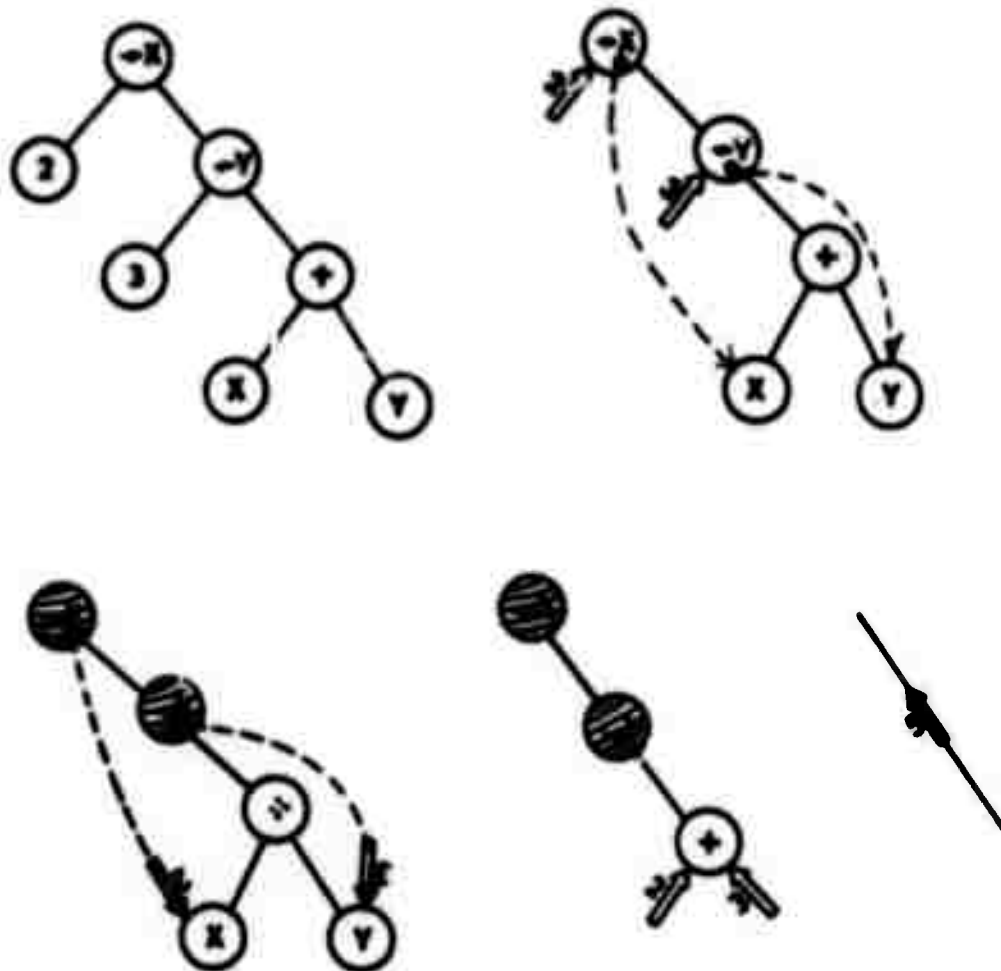
Fig. V-12

(?•X;?•Y;X•Y)

F'g. V-13

Example 3:

The evaluation of the expression

$$[(2+3)+X;5+Y;(X+Y)+X;(Y+1)+Y;(2+Y)-X]$$

has already been studied in the first chapter (fig. 1-6 and 1-11).

V.3.2 The binder-nodes setup '+', and '_NEW_'.

As already noted, the scope of $+X$ is limited to the right component. $X+$ allows to send a value in the right component and in the __environment__ as well (fig. V-14). In order to delimit exactly the scope of $X+$, _NEW_ X is used as a "top-binder": the scope of $X+$ is entirely under the corresponding _NEW_ X (fig. V-15).

$+X$ and $X+$ bind exactly in the same way any variable-node X in their right component. However, contrarily to $+X$, $X+$ sends up the tree a request with its name which is picked up by the first _NEW_ X the request encounters. As a reply, _NEW_ X sends to $X+$ the address of any variable-nodes whose requests it has received (fig. V-16).

Whenever $X+$ receives a value on the left, it sends a copy of it to each variable-node whose address it possesses (fig. V-17).

$X+$, as $+X$, disappears whenever it receives a value simple notice on the right. _NEW_ X disappears whenever it receives a value simple notice.

Example 1:

Fig. V-18 displays the evaluation of the expression:

$$\underline{NEW} \ V \ (2+U;V+U+1;U+V) \ + \ (3+U;U+V)$$

in the left argument of the addition, a value V is produced; V is used in the evaluation of both arguments.
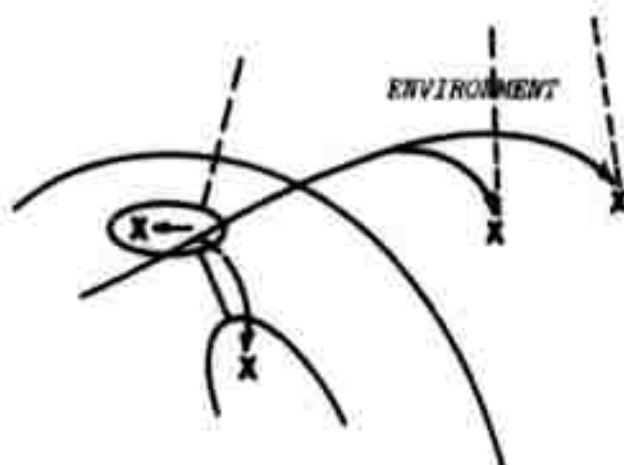
Fig. V-14
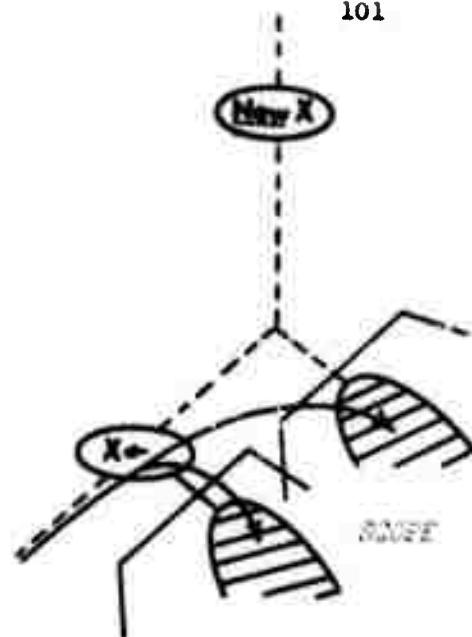
Fig. V-15

Fig. V-16

Fig. V-17

$\underline{NEW}$ V $(2{\rightarrow}U;V{\leftarrow}U{+}1;U{+}V)$ + $(3{\rightarrow}U;U{+}V)$
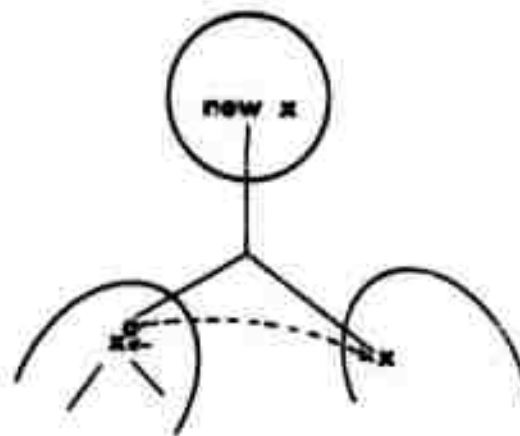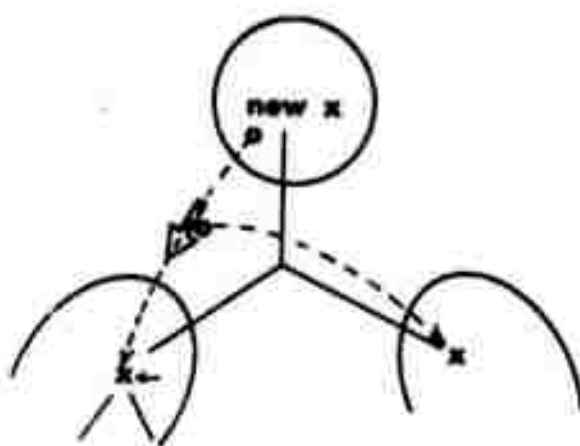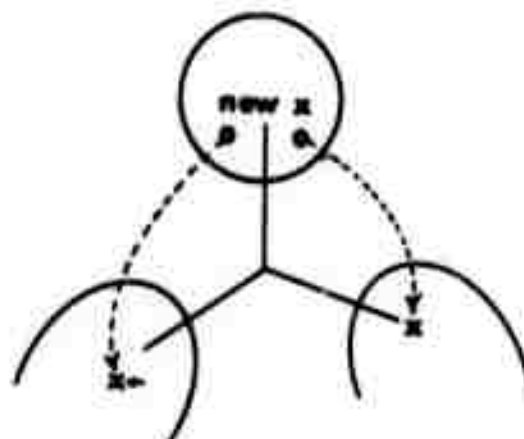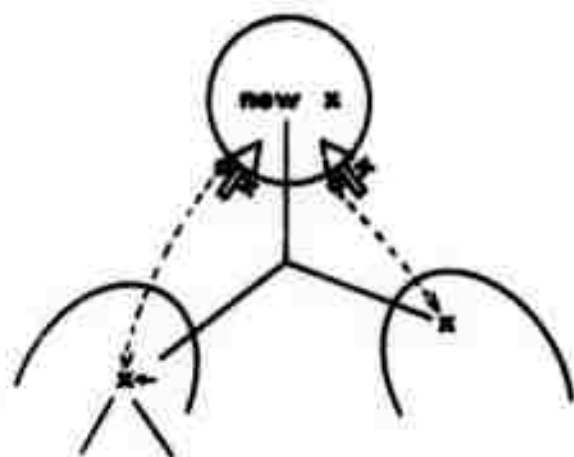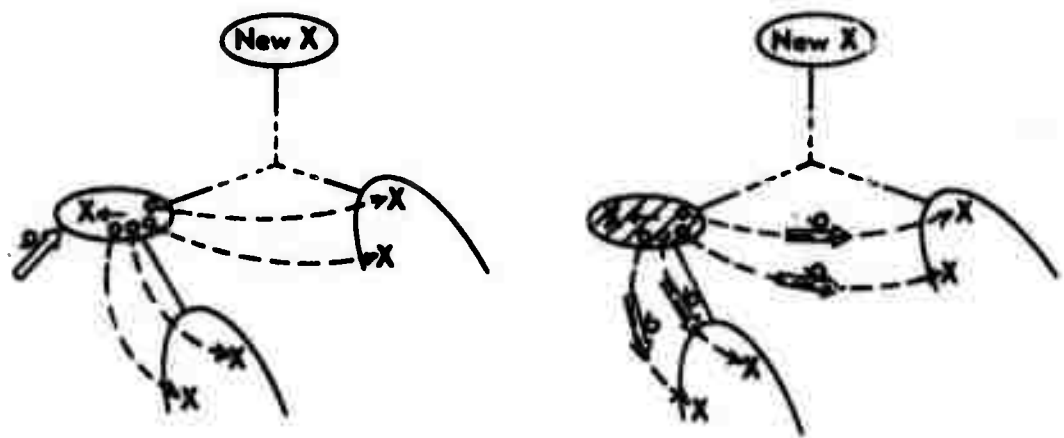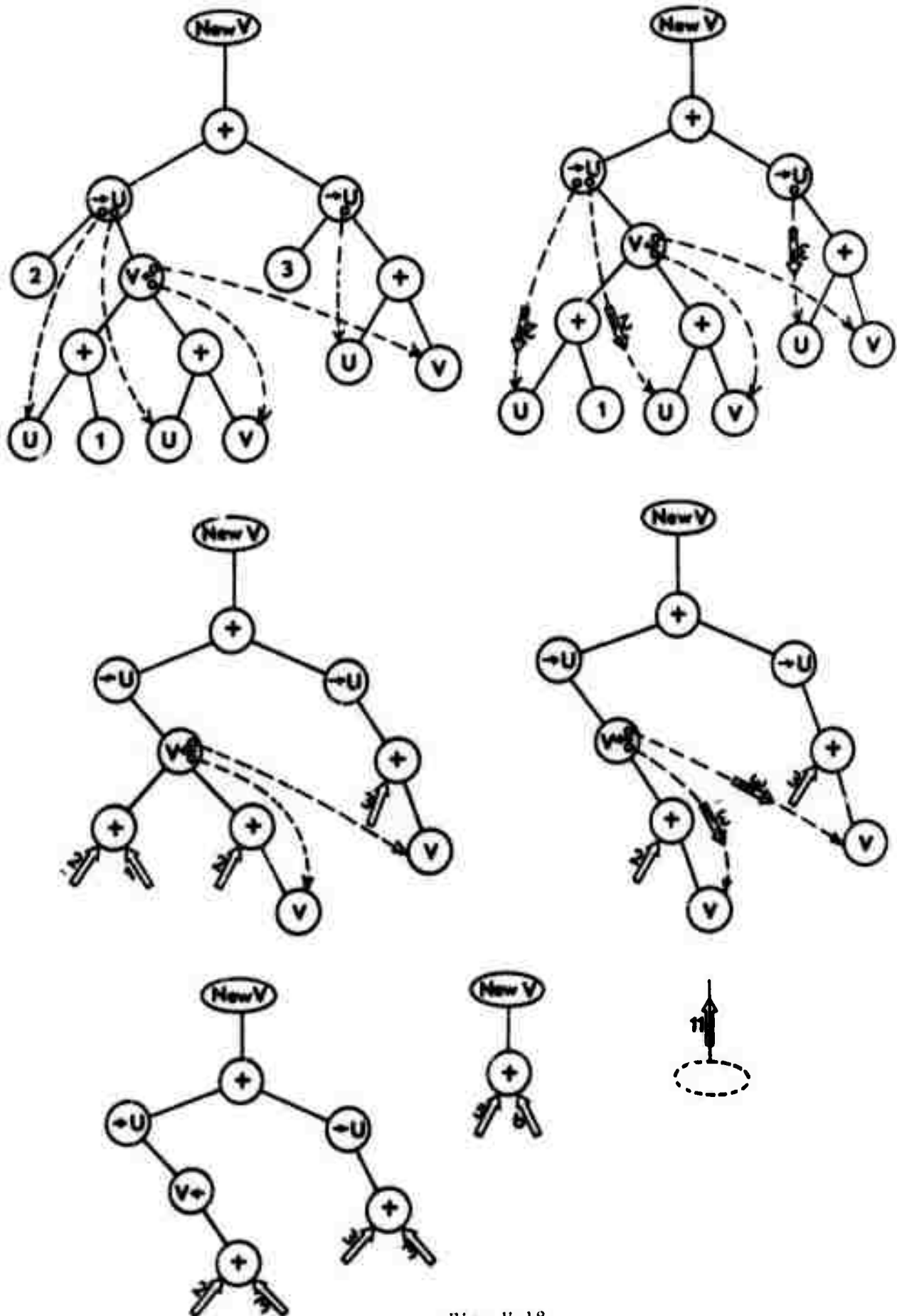
Fig. V-18

Example 2:

Fig. V-19 displays the evaluation of the expression:

$$\underline{NEW} \ U,V \quad (U \leftarrow 5;[U+V] \rightarrow Z;Z+U) \ + \ (V \leftarrow U+1;U) \ .$$

Such an expression is remarkable in that it cannot be evaluated by a sequential machine: the evaluation of the first argument requires the value of V which is produced in the second argument whose evaluation requires the value U which is produced in the first argument. In DCPL there is no deadlock since the two arguments are evaluated concurrently.

V.3.3 The binding of procedures.

A procedure is a computation ob which is intended for implementation and usage in various parts of a computation program. In order to ensure generality of utilization, the set of variables used inside a procedure must not conform to the set of variables used in the environments in which the procedure is to be implemented (please, see fig. V-20).

Whenever a variable in a procedure is to receive an argument from the environment, the variable is bound inside the procedure to a $\underline{LAMBDA}$ binder-node; the procedure itself is interfaced with its environment through a node sendown '$\downarrow$' (fig. V-21a).

Whenever a variable in a procedure is to send an argument to the environment, the variable is bound inside the procedure to a $\underline{MU}$ binder-node; the procedure itself is interfaced with its environment through a node sendup '$\uparrow$' (fig. V-21b).

$\underline{NEW}\ U,V\ (U{\leftarrow}5;[U{+}V]{\rightarrow}Z;Z{+}U)\ +\ (V{\leftarrow}U{+}1;U)$

Fig. V-19

106
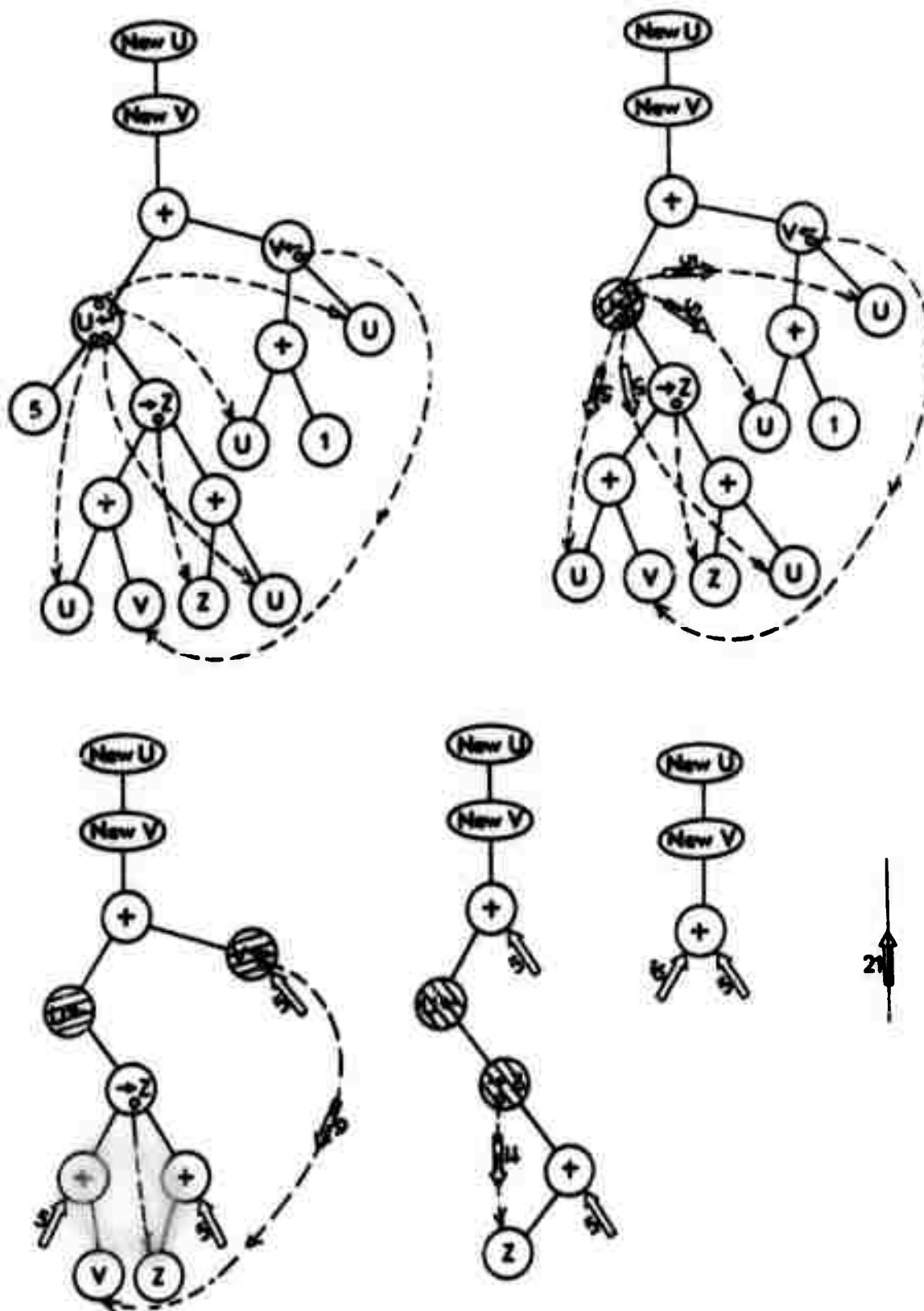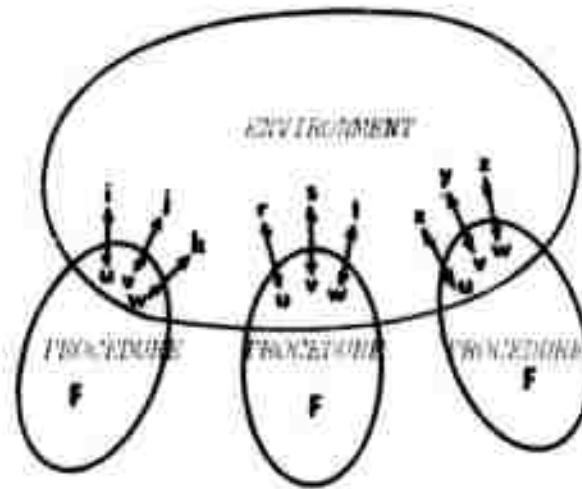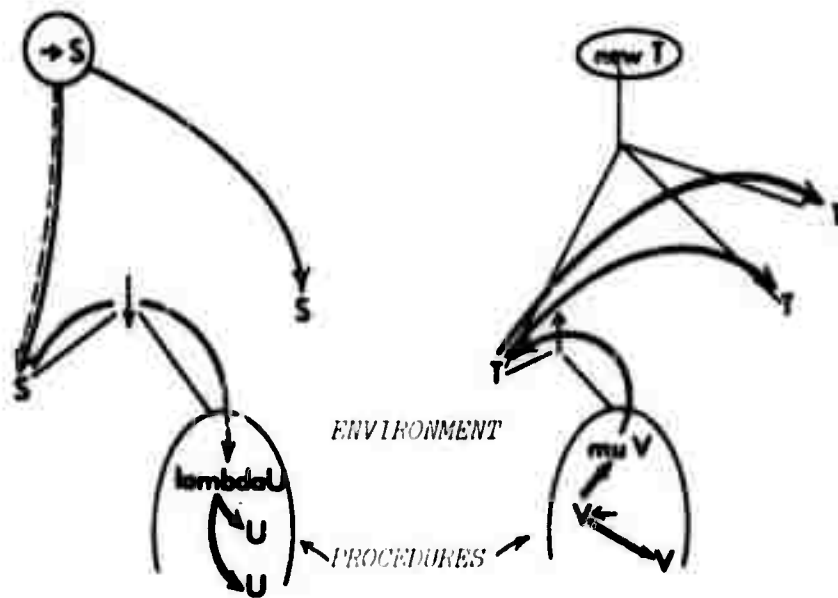
Fig. V-20

Fig. V-21a

Fig. V-21b

'_LAMBDA_' and sendown '↓'.

An example: _LAMBDA_ S (S+1)   is a procedure with one lambda-variable S.   [_LAMBDA_ S (S+1)] (3)   represents that procedure applied to one argument 3. According to the (syntax) replacement rule number 5, the computation tree is:
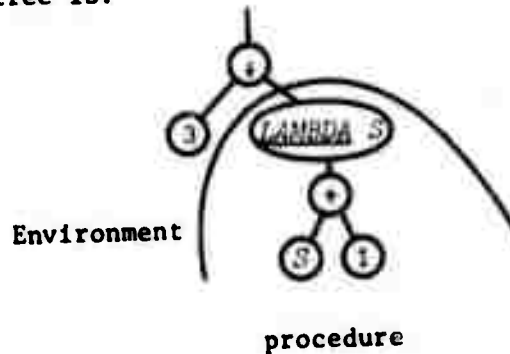
procedure

In fig. V-22a and V-22b the variable-node S is bound to the lambda-node: the former sends up the tree a request which is picked up by the latter. This binding occurs inside the procedure itself and may be performed before the implementation in space of the procedure, i.e., the physical embedding of the procedure in its environment. Then the procedure is bound to its environment: the lambda-node sends up the tree a request with its address which is picked up by the sendown node (fig. V-22c and V-22d). The argument may flow down, from the environment into the procedure, and reach the variable-node (remainder of figure V-22).

Whenever a procedure contains several lambda-variables, it is necessary to have the arguments matched with the lambda-nodes (fig. V-23). The matching is performed dynamically in the following way: when a lambda-node sends up the tree a request with its address, this request contains a count set to zero. This count is incremented by one whenever the demand encounters another lambda-node, and decremented by one whenever the demand encounters a sendown node

108

ENVIRONMENT

PROCEDURE

Fig. V-22a

ENVIRONMENT
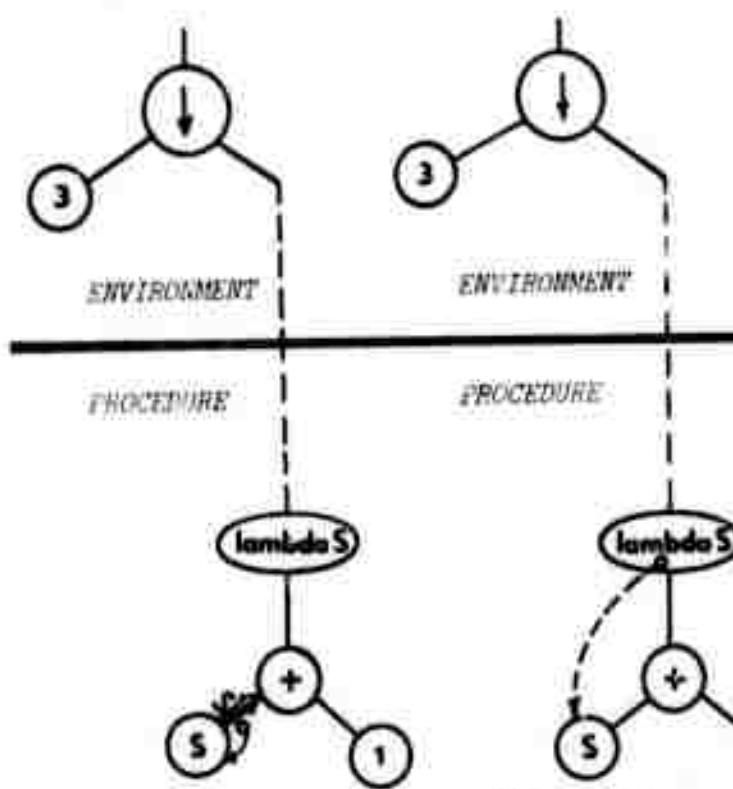
PROCEDURE

Fig. V-22b

Fig. V-22c

Fig. V-22d
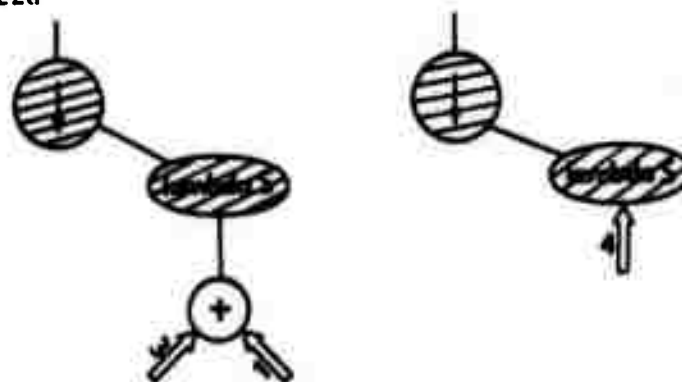
Fig. V-22

Fig. V-23

Fig. V-24

110
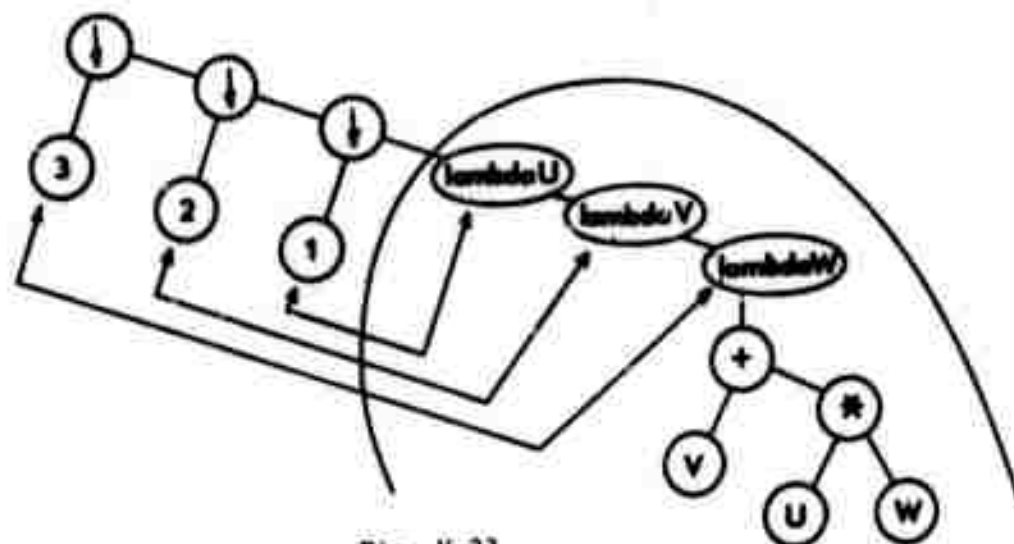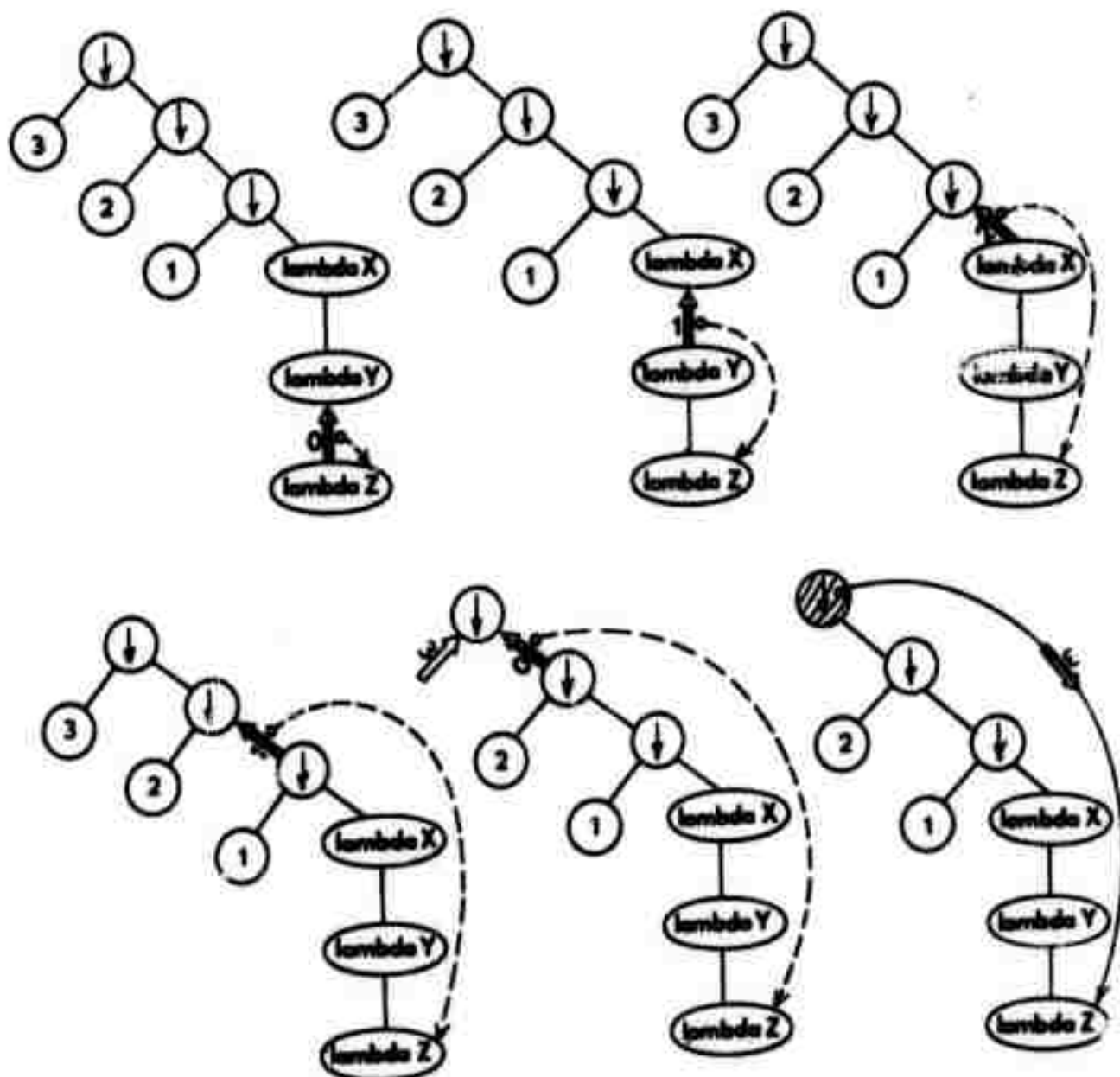
with a non-zero count. The demand is picked up by the first sendown
node which receives it with a count equal to zero (fig. V-24).

'_MU_' and sendup '↑'.

An example: ..... [_MU_ T ... T↑↑ ...] ↑:X) ...

environment
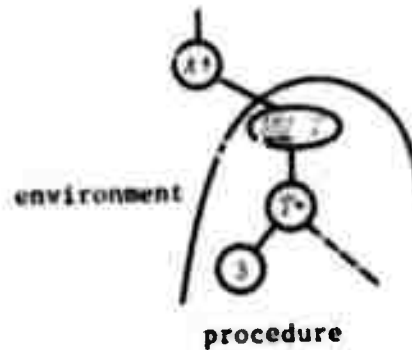
procedure

The binding inside the procedure and the binding of the
procedure to the environment are performed exactly as for lambda
and sendown. However, instead of receiving an argument, the variable
in the procedure receives the address in the environment to which
the produced value is to be sent (fig. V-25).

Let us consider again the example of fig. V-19:

$$\underline{NEW}\ U,V\ (U{\leftarrow}5;[U{+}V]{\cdot}Z;Z{\leftarrow}U)\ {+}\ (V{\cdot}U{+}1;U)$$

It may be expressed as

$$\underline{NEW}\ U,V\quad F(V;U)\ {+}\ G(U;V)$$

with $F = \underline{MU}\ U\quad \underline{LAMBDA}\ V\ (U{\leftarrow}5;[U{+}V]{\cdot}Z;Z{\leftarrow}U)$

and $G = \underline{LAMBDA}\ U\ \underline{MU}\ V\ (V{\cdot}U{+}1;U)$.   In order to make it clear that

U and V are bound in F and G, we can take

$$F = \underline{MU}\ R\quad \underline{LAMBDA}\ S\ (R{\leftarrow}5;[R{+}S]{\cdot}T;T{\leftarrow}R)$$

and $G = \underline{LAMBDA}\ R\ \underline{MU}\ S\ (S{\cdot}R{+}1;R)$ (fig. V-26).
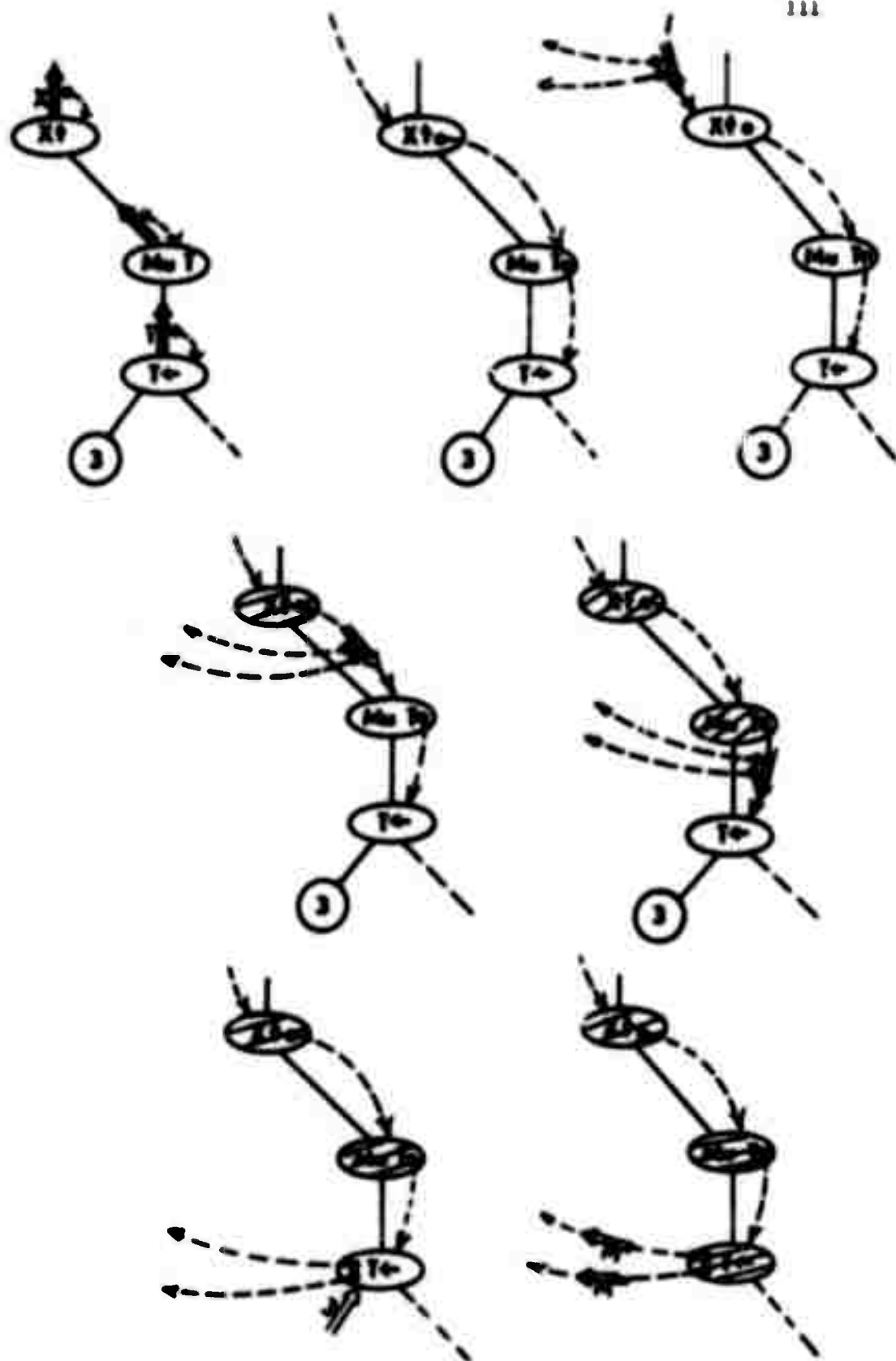
Fig. V-25
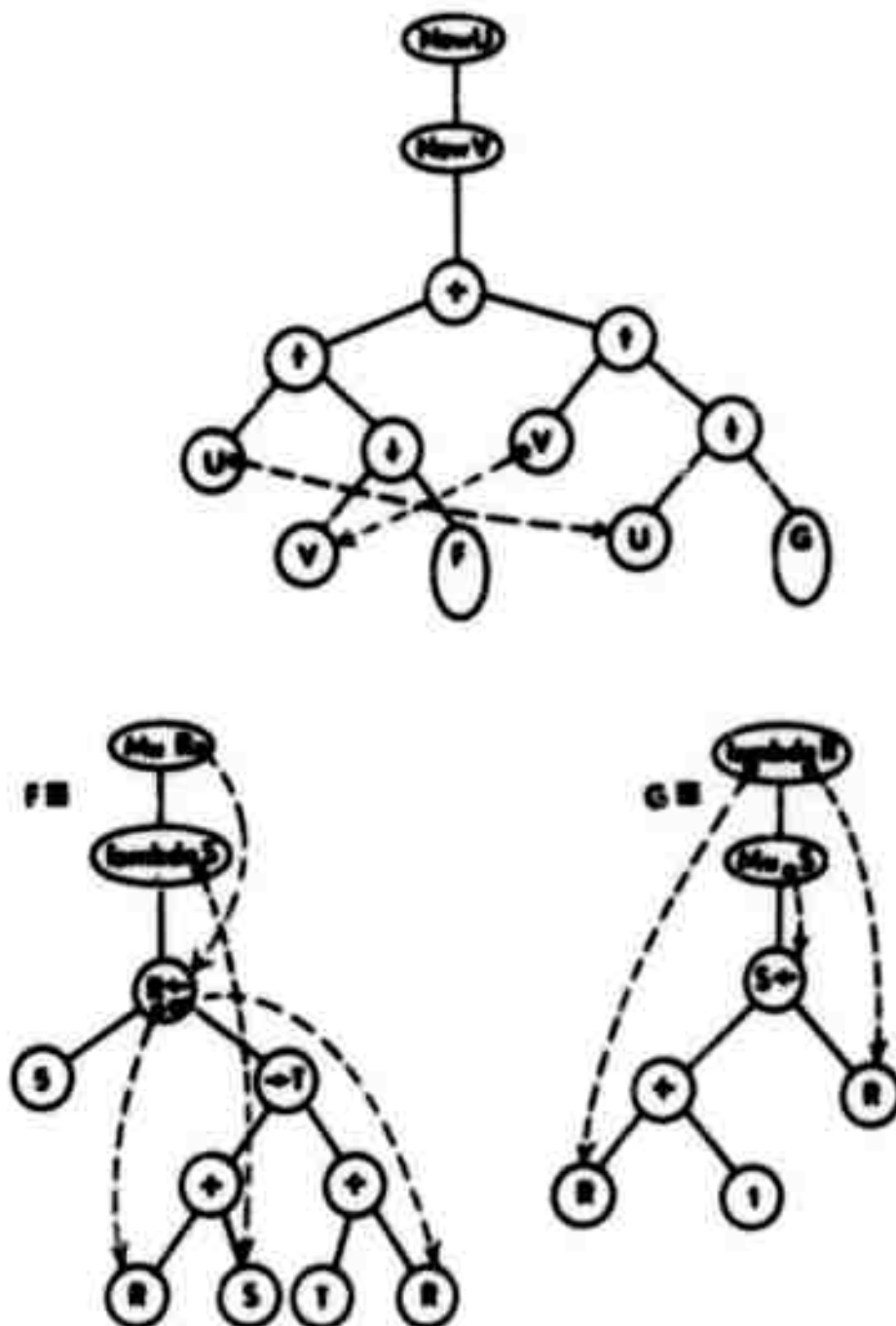
Fig. V-26a
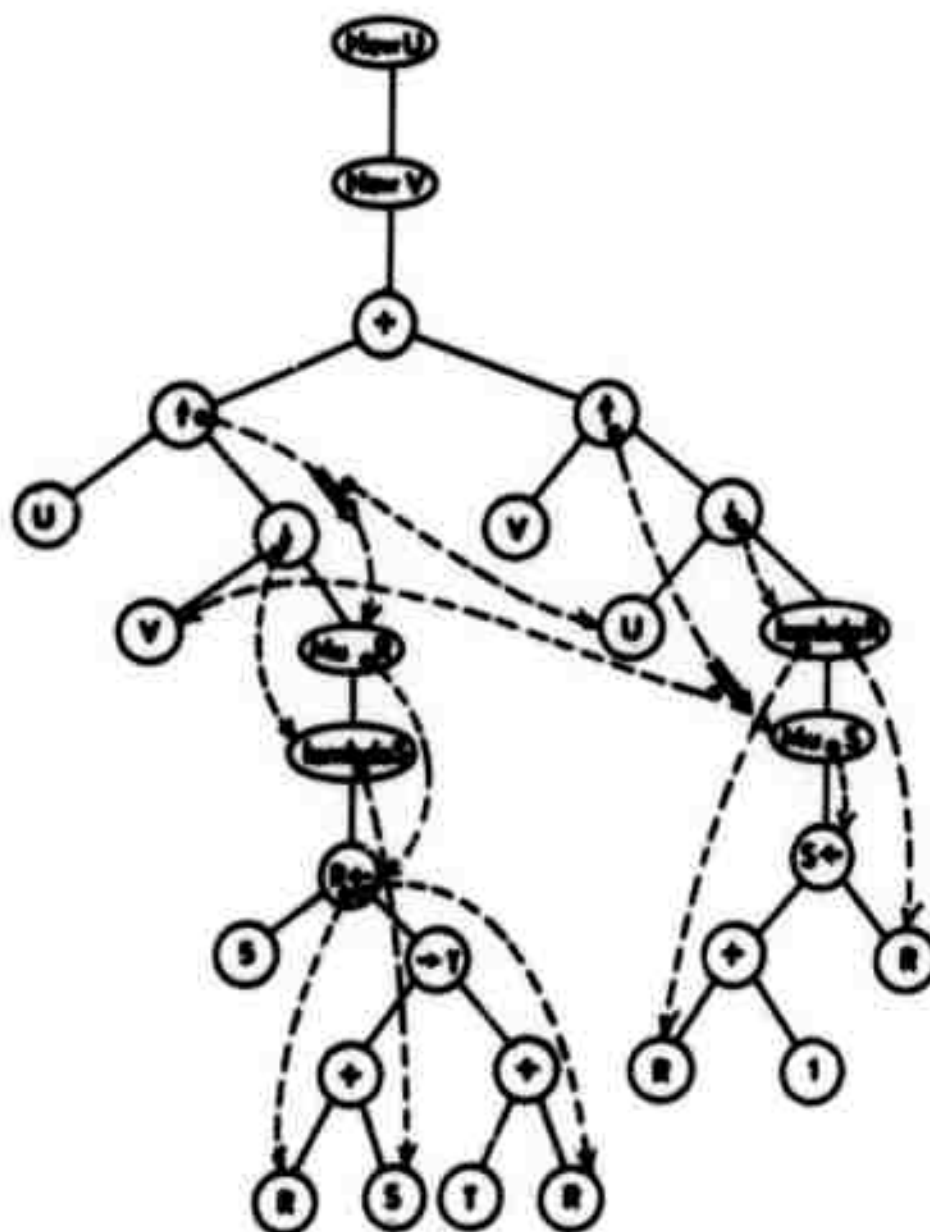
Fig. V-26b

V.4   Procedures as arguments.

V.4.1 Pseudo-values.

As stated in section V.1.2, any ob has its description as a value in the universe of values. Such a value is called a pseudo-value: "_LAMBDA_ S ($S+1$)" is a pseudo-value representing the ob _LAMBDA_ S ($S+1$).

Whenever a variable receives a pseudo-value as an argument, it implements the ob described by the pseudo-value at its own place. As a result a procedure may be received as an argument and implemented if the corresponding pseudo-value is given beforehand (fig. V-27).

Let us however consider the following example:

.... _LAMBDA_ S,T ($S+T$) $\rightarrow F$ .... $F(5) \rightarrow G$ .... $G(4)$ ....

In this example the procedure _LAMBDA_ S,T ($S+T$) is first implemented in the environment A and receives the argument 5; then the resulting procedure is to be sent and implemented in the place of G in the environment B. This may only occur if the ob [_LAMBDA_ S,T ($S+T$)](5) is able to produce a pseudo-value representing itself. We will see in the sequel how we may implement such a pattern of behavior.

V.4.2 Pseudo-argument.

There is a special message 0 called pseudo-argument: whenever a variable X receives a pseudo-argument, it sends up the tree the pseudo-value representing itself, "X" (fig. V-29).

Whenever the combinator representing an operation receives the proper number of input values, it computes the resulting value and sends it up the tree. If the same combinator receives at least
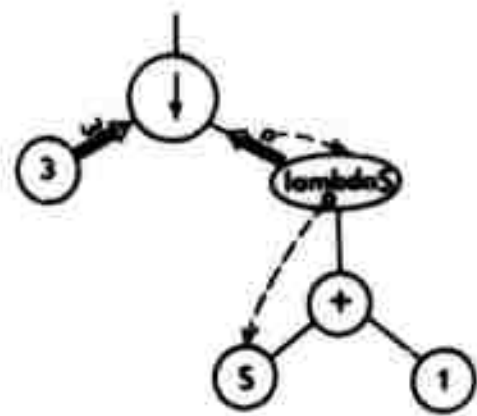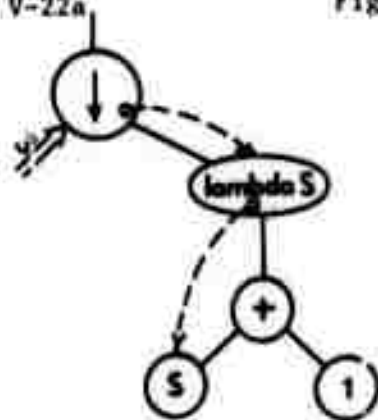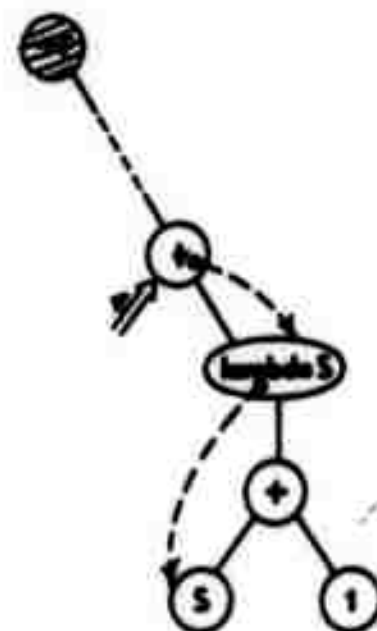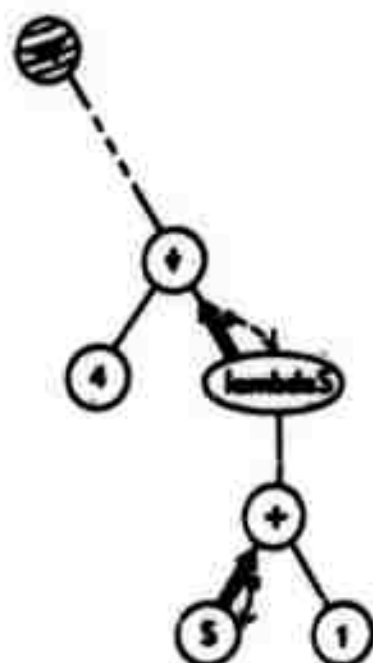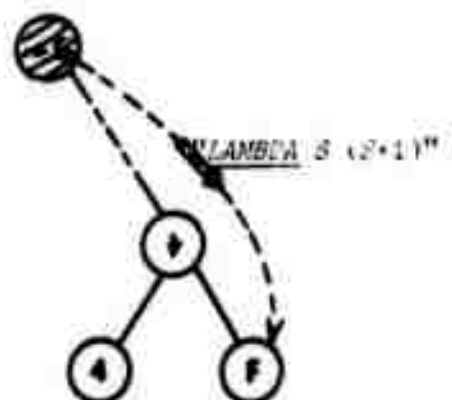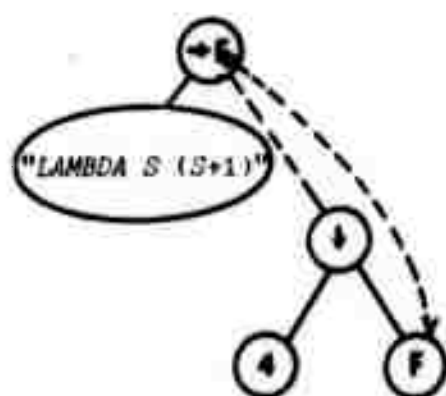
Fig. V-27
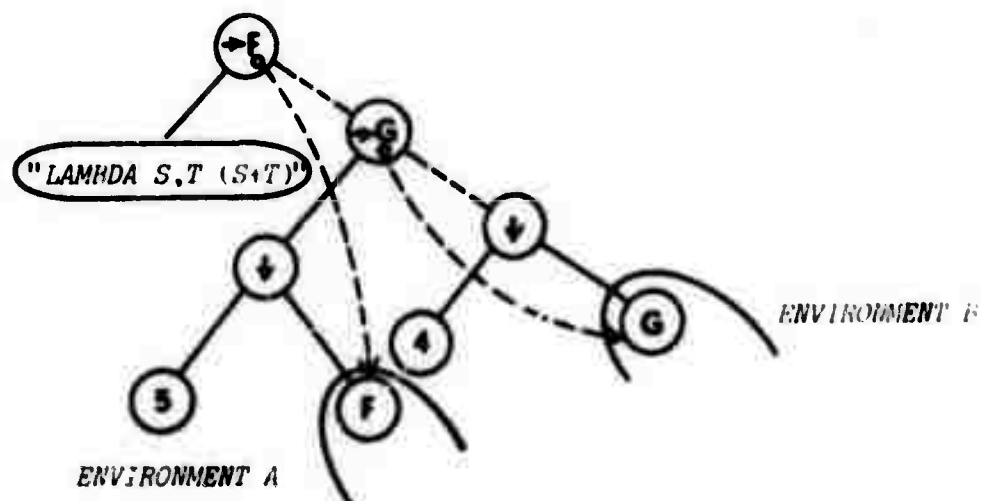
Fig. V-28
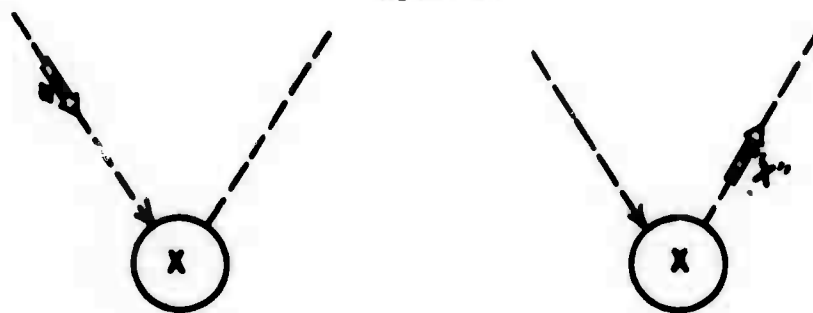
Fig. V-29
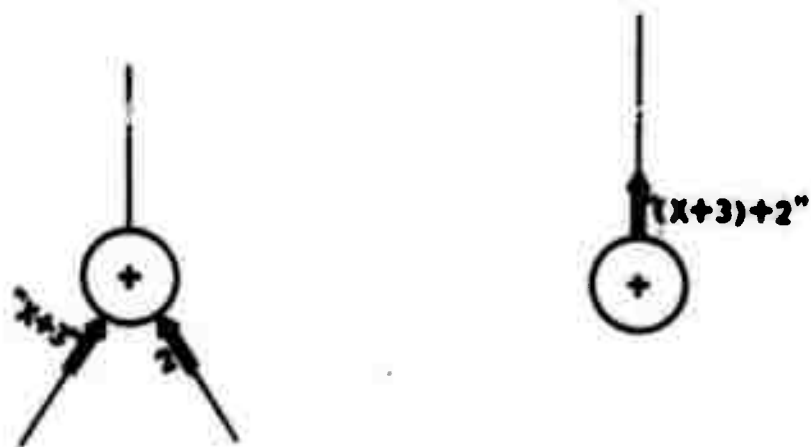
Fig. V-30

one pseudo-value representing an ob, it produces a pseudo-value
which is sent up the tree: for instance, in fig. V-30 the combinator
'+' receives a pseudo-value "$X+3$" and a value 2; it sends up the tree
the pseudo-value "$(X+3)+2$".

An ob may have some of its variables receiving a value and some
other variables receiving a pseudo-value. As a result, this ob is
partly evaluated into an ob whose associated pseudo-value is produced
and sent up the tree (fig. V-31).

V.4.3 Procedural arguments.

We are now able to handle the example of fig. V-28: whenever
the node $+G$ receives from the left a request sent by a lambda-node,
it sends as a reply a pseudo-argument (fig. V-32). As a result the
ob for which $F(5)$ stands may be partially evaluated, the correspon-
ding pseudo-value is produced and sent to the variable G (fig. V-33).

The problem of having procedures as arguments is not yet
completely solved: we have seen how a combinator representing an
operation reacts whenever it receives a pseudo-value; however we
do not know yet how other combinators (for instance $+, +, \ldots$) are
to react. Let us study the two following examples:

Example 1: $(\,(\,\text{LAMBDA}\ X\ (\,(\,\text{LAMBDA}\ Y\ (Y+1)\,)+G;G(X))\,)+F;F(2)\,)$
                                                            (fig. V-34).

Example 2: $(\,(\,\text{LAMBDA}\ X\ (\,(\,\text{LAMBDA}\ Y\ (Y+X)\,)+G;G(X))\,)+F;F(2)\,)$
                                                            (fig. V-35).

In example 1, the component $(\,(\,\text{LAMBDA}\ Y\ (Y+1)\,)+G;G(X))$
may be handled as previously (fig. V-36):
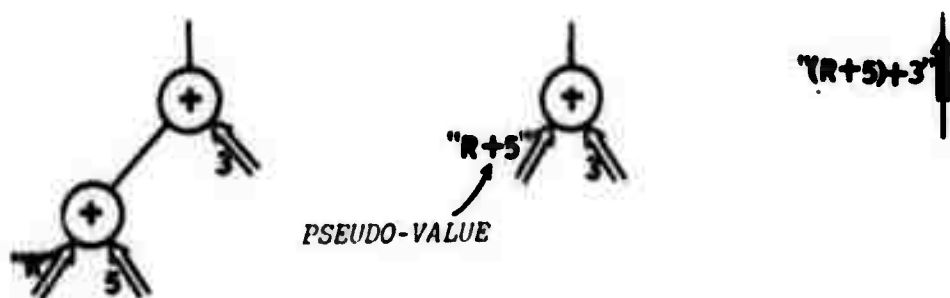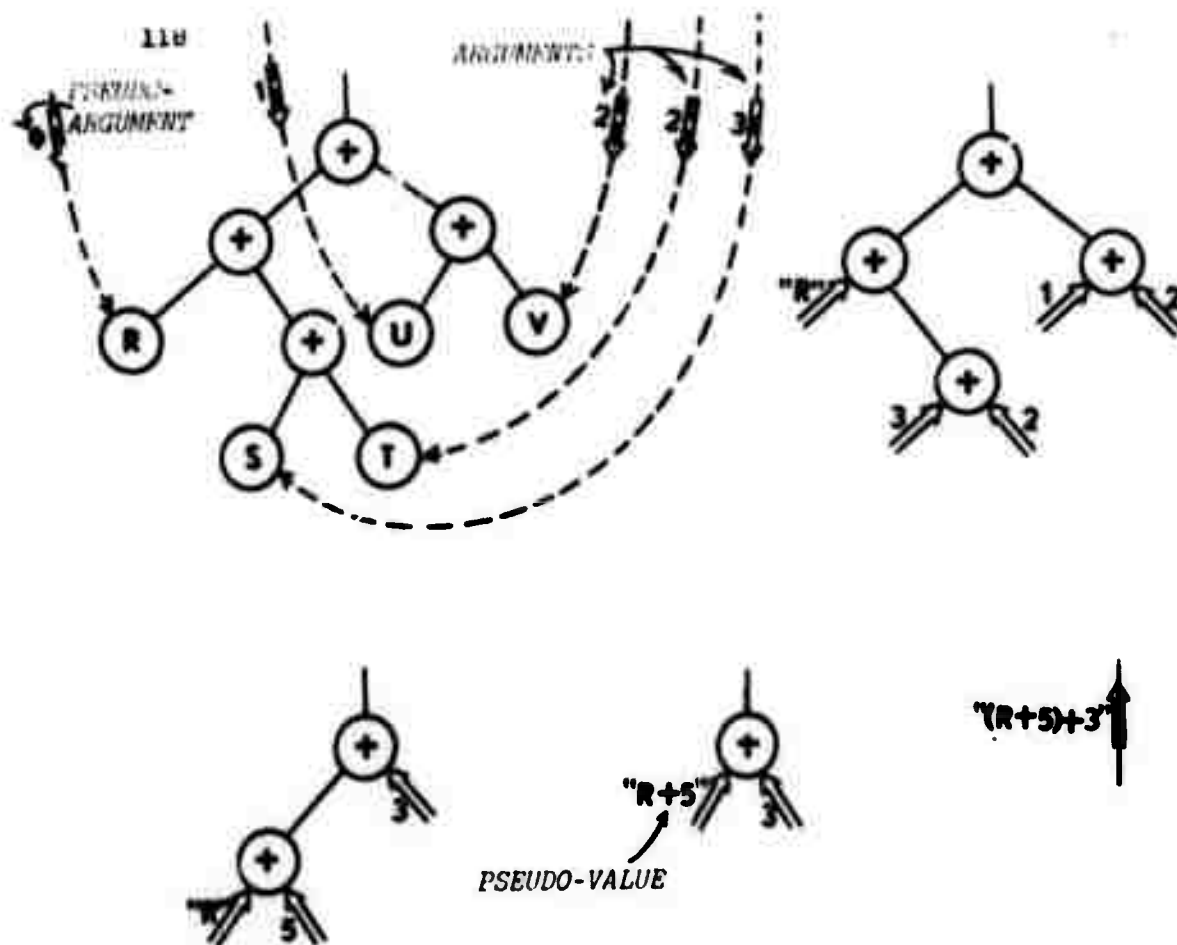
Fig. V-31

Fig. V-32

Fig. V-33

120
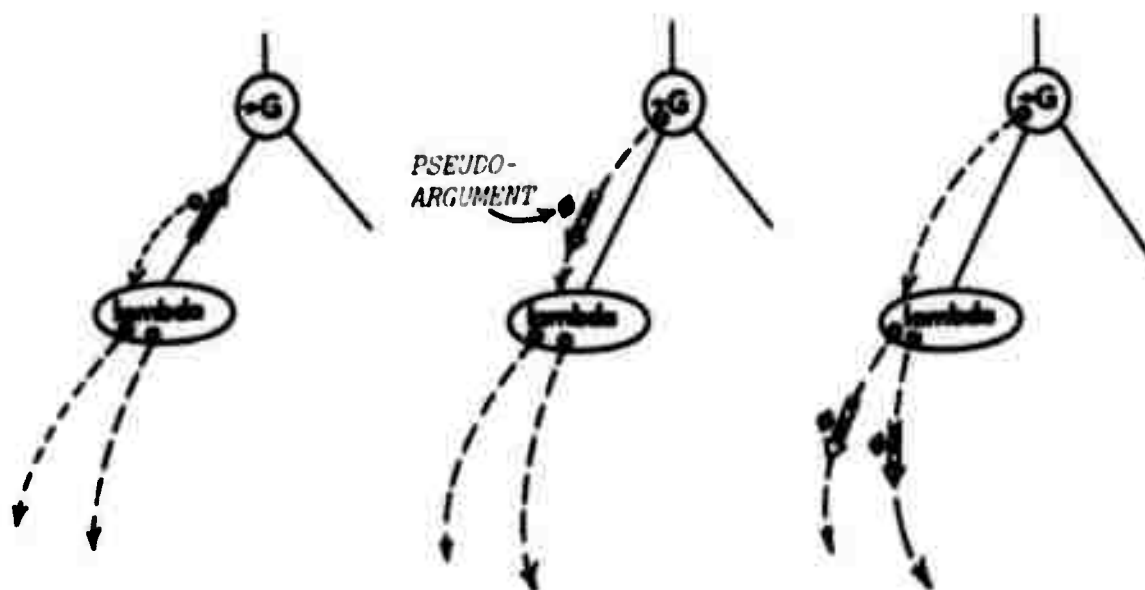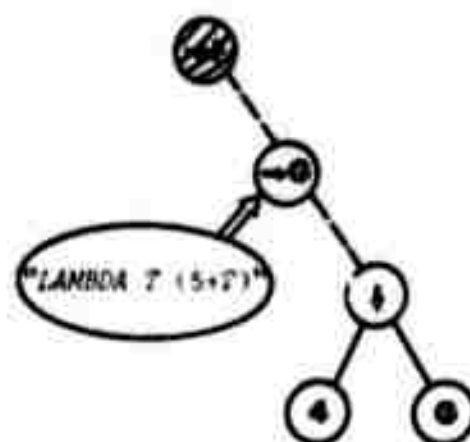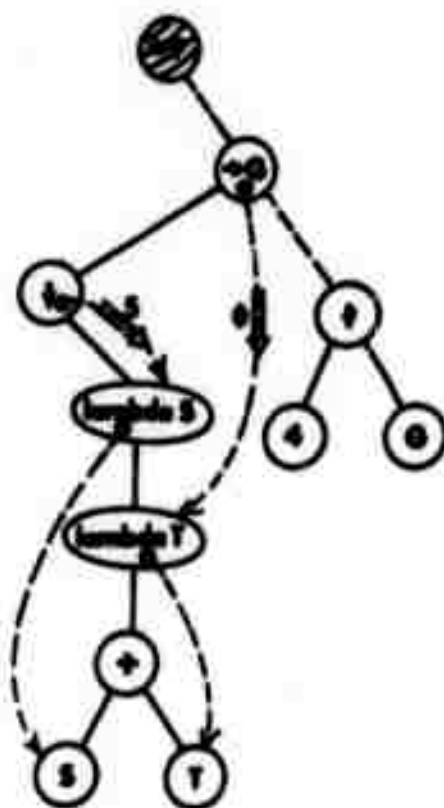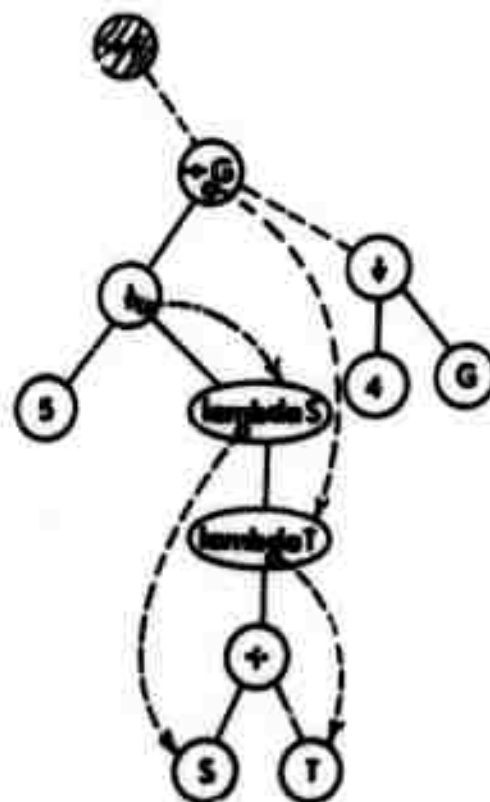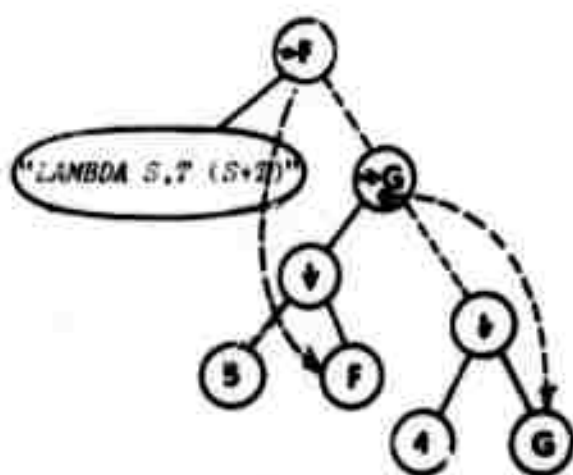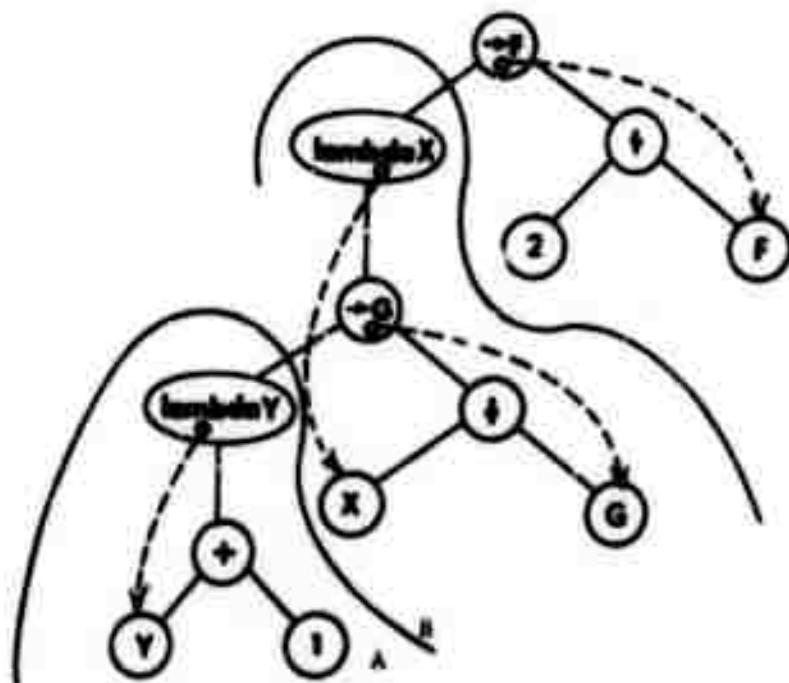
Fig. V-34
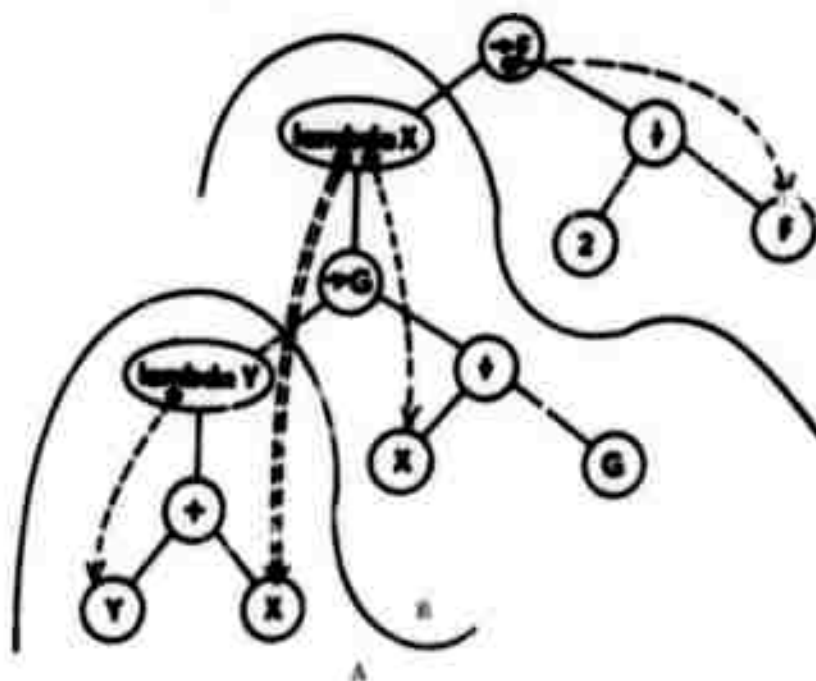
Fig. V-35

1. →G sends to the lambda-node *LAMBDA* Y a pseudo-argument φ.

2. on receiving the pseudo-value "*LAMBDA* Y (Y+1)", →G passes it along to the variable-node G where the procedure is to be implemented.

The variable-node X (in G(X)) receives a pseudo-argument sent by →F via *LAMBDA* X ; it sends up the tree the pseudo-value "X" which is received, on the left, by the node sendown,↓ . On receiving this pseudo-value, sendown sends a pseudo-argument to the procedure on the right, and waits for a pseudo-value from this procedure (fig. V-37). Then ↓ combines the two pseudo-values into a pseudo-value it sends up the tree. The completion of the evaluation of example 1 is displayed on fig. V-38.

In example 2 the procedure *LAMBDA* Y (Y+X) cannot be transmitted and implemented in the place of G for X is externally bound: it is indeed our policy to implement an argument only after its complete evaluation; we require (see fig. V-35) that the implementation of B in the place of F occurs before the implementation of X in the place of G .

As a result →G must have here another behavior: on receiving a pseudo-value on the left it sends a pseudo-argument to each variable-node it binds on the right. On receiving a pseudo-value or its right →G combines the two pseudo-values and sends the resulting pseudo-value up the tree (fig. V-39).

In summary, in the following situation
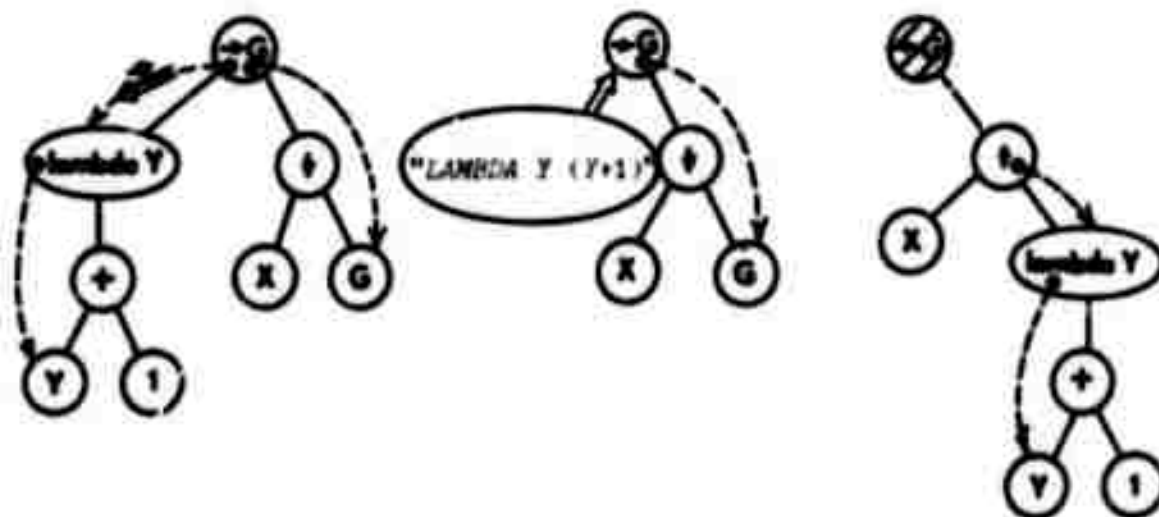
Fig. V-36
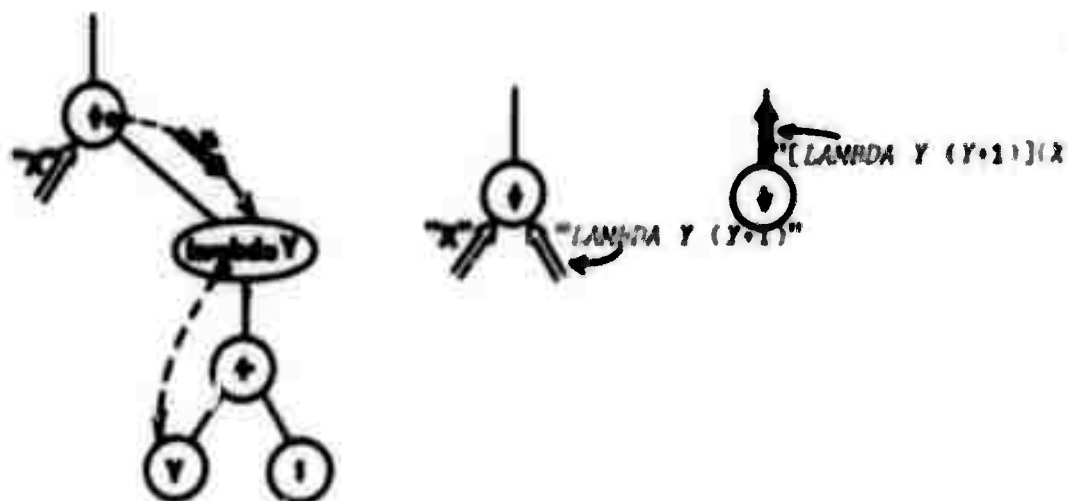
Fig. V-37
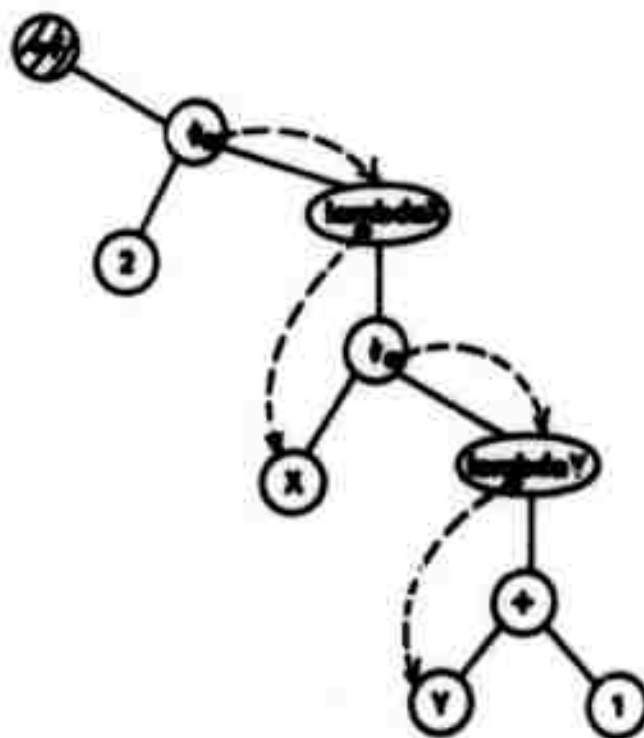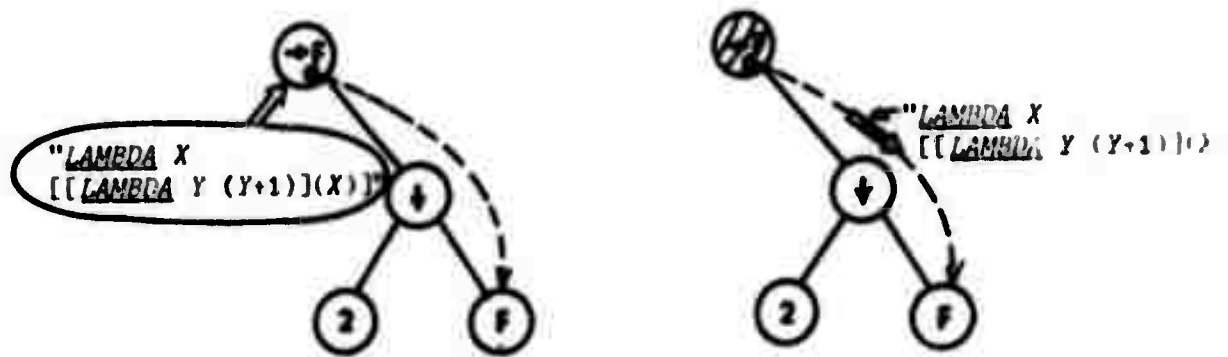
Fig. V-38
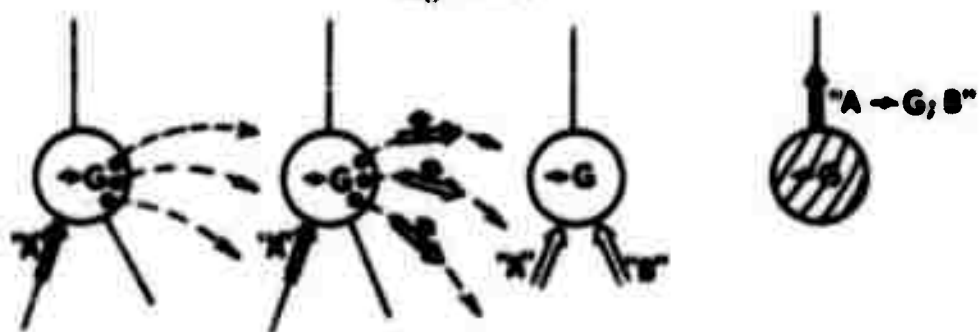
Fig. V-39

the ob A may or may not be to receive some argument from the
environment.

In the latter situation (fig. V-40) and if A does receive
these arguments, A produces a pseudo-value "A", after having received
a pseudo-argument ϕ from $\rightarrow G$ (fig. V-41). This pseudo-value "A" is
sent by $\rightarrow G$ to the variable-node G where A is implemented.

If A receives at least one pseudo-argument from the environment
(fig. V-42), the evaluation of A is not completed and "A" is not
to be sent and implemented in the place of G . Instead, on receiving
the pseudo-value "A", $\rightarrow G$ sends to the variable node G a pseudo-argu-
ment. On receiving the pseudo-value "B", $\rightarrow G$ sends up the tree the
pseudo-value "$A \rightarrow G:B$" (fig. V-39).It is therefore necessary that $\rightarrow G$
is able to distinguish between these two situations.

We implement such a behavior in the following way:

Whenever a variable-node sends up the tree a request with its
own address, such a request contains an integer called count, equal
to zero at the beginning. While the request flows up the tree, the
count is incremented by one every time the request reaches on the left
one of the combinators setdown '$\rightarrow$' , setup '$\rightarrow$' , or sendown '$\downarrow$' .

Such a count is associated to pseudo-arguments and pseudo-values
as shown in the following table:

the ob λ may or may not be to receive some argument from the environment.

In the latter situation (fig. V-40) and if λ does receive these arguments, λ produces a pseudo-value "λ", after having received a pseudo-argument θ from $\neg G$ (fig. V-41). This pseudo-value "λ" is sent by $\neg G$ to the variable-node G where λ is implemented.

If λ receives at least one pseudo-argument from the environment (fig. V-42), the evaluation of λ is not completed and "λ" is not to be sent and implemented in the place of G . Instead, on receiving the pseudo-value "λ", $\neg G$ sends to the variable node G a pseudo-argument. On receiving the pseudo-value "θ", $\neg G$ sends up the tree the pseudo-value "$\lambda \neg G.\theta$" (fig. V-39).It is therefore necessary that $\neg G$ is able to distinguish between these two situations.

We implement such a behavior in the following way:

Whenever a variable-node sends up the tree a request with its own address, such a request contains an integer called count, equal to zero at the beginning. While the request flows up the tree, the count is incremented by one every time the request reaches on the left one of the combinators setdown '⋆' , setup '∘' , or sendown '↓' .

Such a count is associated to pseudo-arguments and pseudo-values as shown in the following table:

EVENT   ACTION

(situation of example 1)

(situation of example 2)

Fig. V-40

Fig. V-41

Fig. V-42

<u>Remark</u>.

Whenever a combinator ⊛S receives from the left a pseudo-value with a count of zero, it is assured (by these behaviors) that the left ob has not received a pseudo-argument from above ⊛S. As a result, the pseudo-value received may be implemented (see example 1).

The behaviors of ⊛ and ⊹ are similar to the one described for ⊛.

V.5   <u>Conditional expressions</u>.

'<u>IF</u>' (or '|') is a combinator of degree three:

The ob "predicate" is assumed to produce a logical value: *FALSE* or *TRUE* . In the former case the ob "if-true-ob" is deleted from the computation tree; in the latter case it is the ob "if-false-ob" which is deleted from the computation tree (fig. V-43a and V-43b).

As shown in section V.3, any ob may exchange information with its environment; as a result, messages can freely flow down from the environment into the obs "predicate", "if-true-ob", "if-false-ob" which may be considered as being evaluated concurrently.

However, it is important to prevent an ob which eventually would be cancelled from sending messages to its environment. For this reason, the messages which are sent to the environment by the obs "if-true-ob" and "if-false-ob" are picked up and kept in two different packages at the '<u>IF</u>'-combinator level (fig. V-44a). When the value of the ob "predicate" reaches the combinator, one of the two packages is destroyed

128

IF-TRUE-OB   IF-FALSE-OB

IF FALSE-OB

Fig. V-43a

true

IF-TRUE-OB   IF-FALSE-OB

IF-TRUE-OB

Fig. V-43b

ENVIRONMENT

picked up and kept outgo
messages

PREDICATE

IF-FALSE-OB

IF-TRUE-OB

Fig. V-44a

ENVIRONMENT

true

IF-TRUE-OB

Fig. V-44b

ENVIRONMENT

false
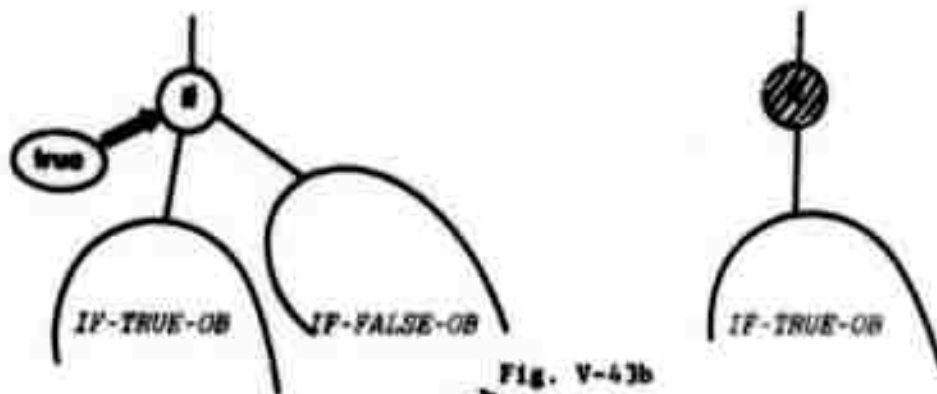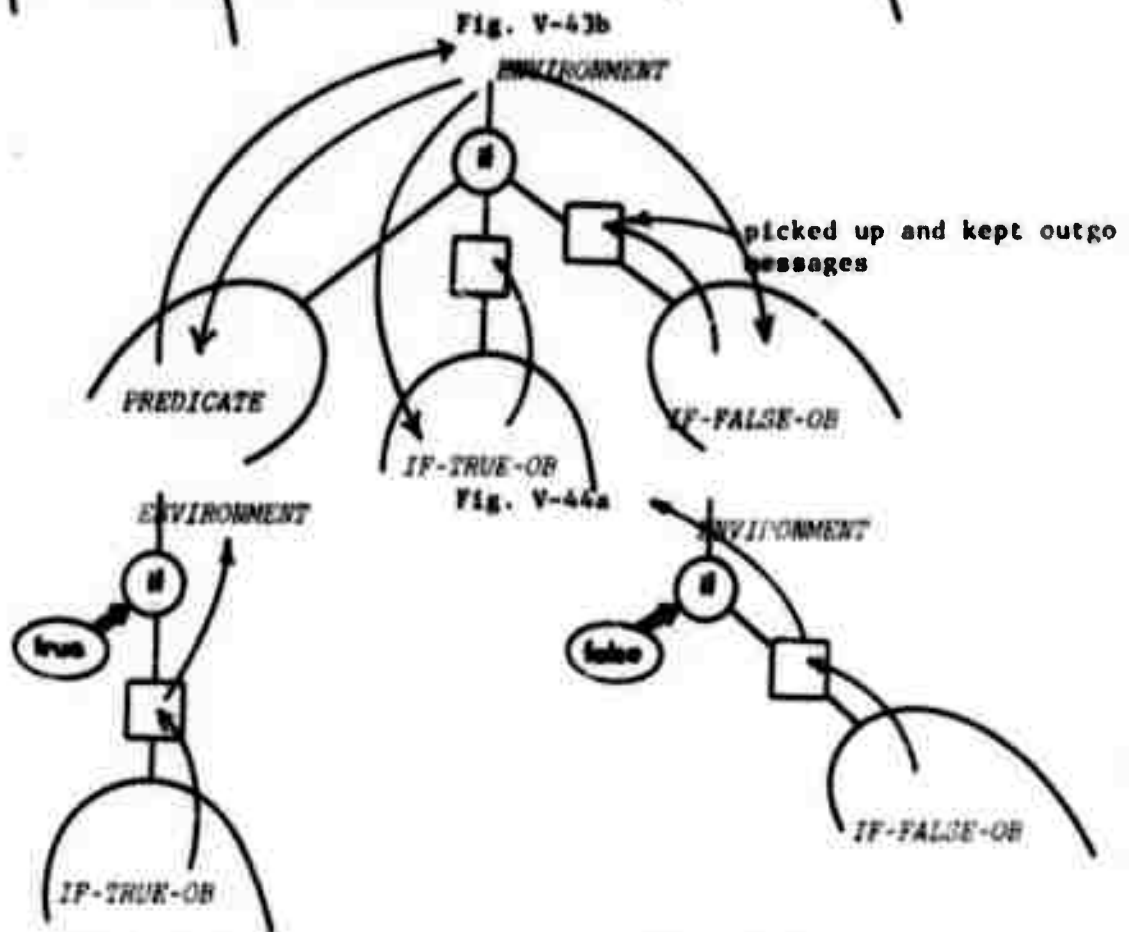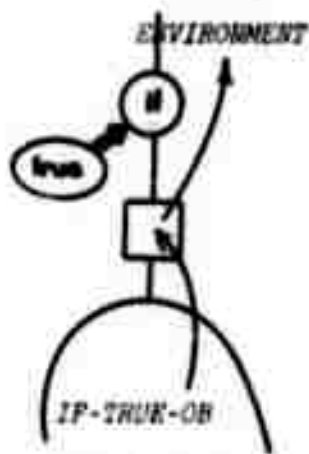
IF-FALSE-OB
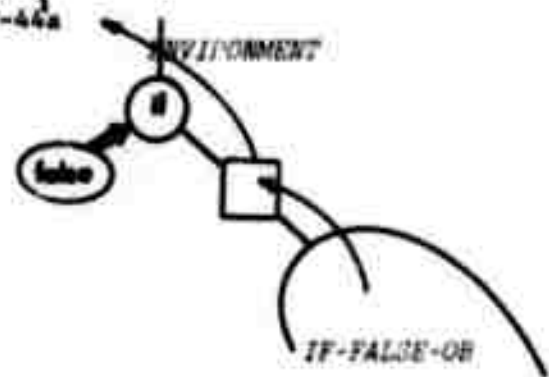
Fig. V-44c

and the messages contained in the other one are freed (fig. V-44b
and V-44c).

V.6   DCPL as a system-oriented programming language.

In the previous section of this chapter, we have seen how
simple DCPL programs may be considered as expressions which may be
evaluated. This section sustains our claim that DCPL is a system-
oriented programming language: a computation is considered a system
of asynchronously cooperating "independent" programs (coroutines)
linked by paths of information along which messages are sent.

V.6.1 Asynchronous events and sequential processes.

Sequential processes which are triggered by the occurrence
of some asynchronous event can be embedded in DCPL. Whenever some
actions are to be synchronized[a] in some way, such sequential processes
may be used (these processes may be very small and perform just one
elementary action).

Any value may be considered as an event whenever we are
interested in knowing whether the value has been received or not,
disregarding the value itself.

We may operate on events with the operators '$\underline{AND}$' and '$\underline{OR}$',
which are not to be confused with the logical operators '$\wedge$' and '$\vee$'.
The nodes which correspond to these operators consider any value they
receive as an event. An AND-node produces an event whenever it re-
ceives an event from both of its sons. An OR-node produces an event
whenever it receives an event from one of its two sons (fig. V-45).
Thus it is possible to have an expression producing an event (fig. V-46).

---

a) The word "synchronize" is here a poor choice. We mean that these
actions must be safeguarded from one another as they would be with sema-
phores (Dijkstra [5]).
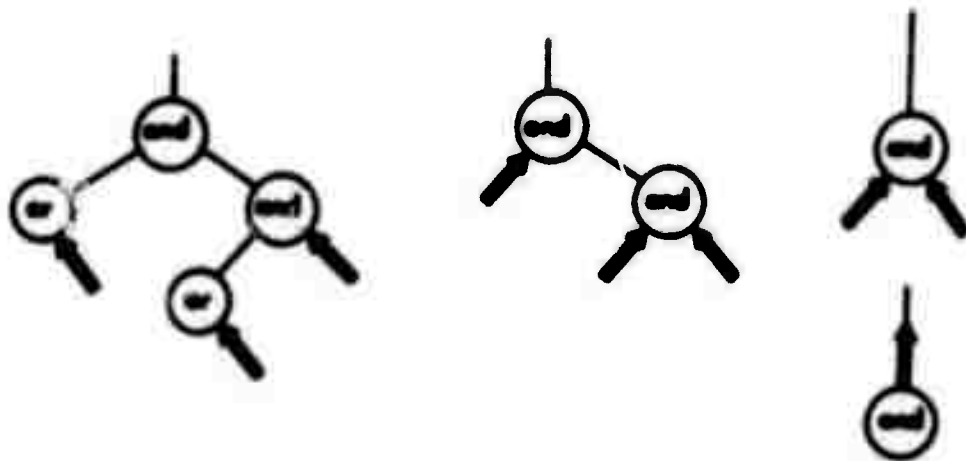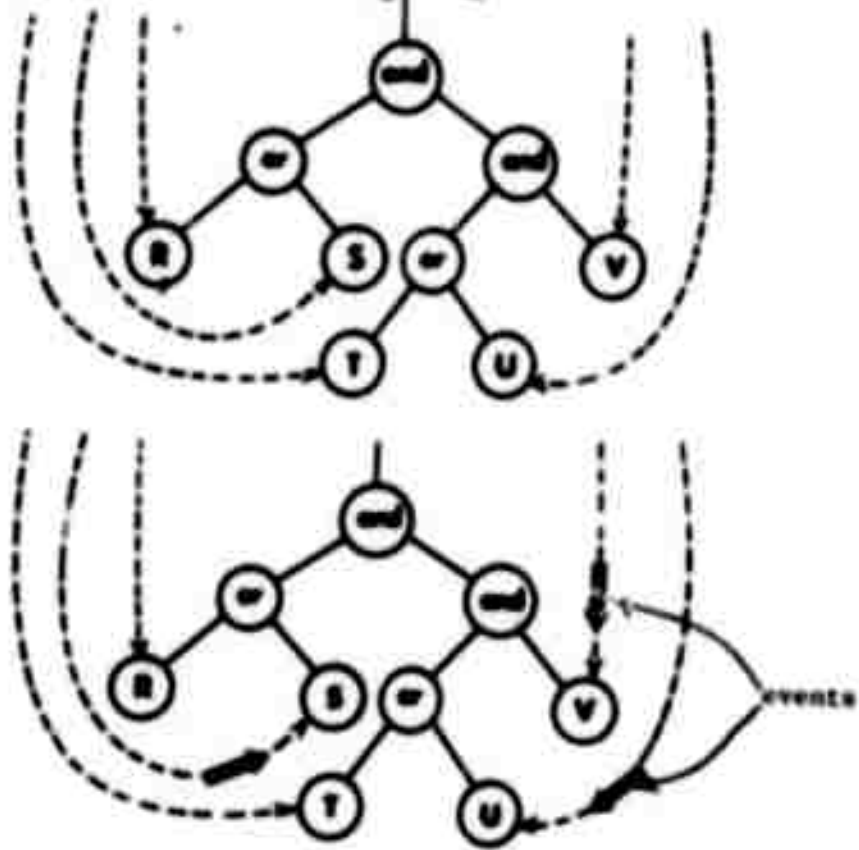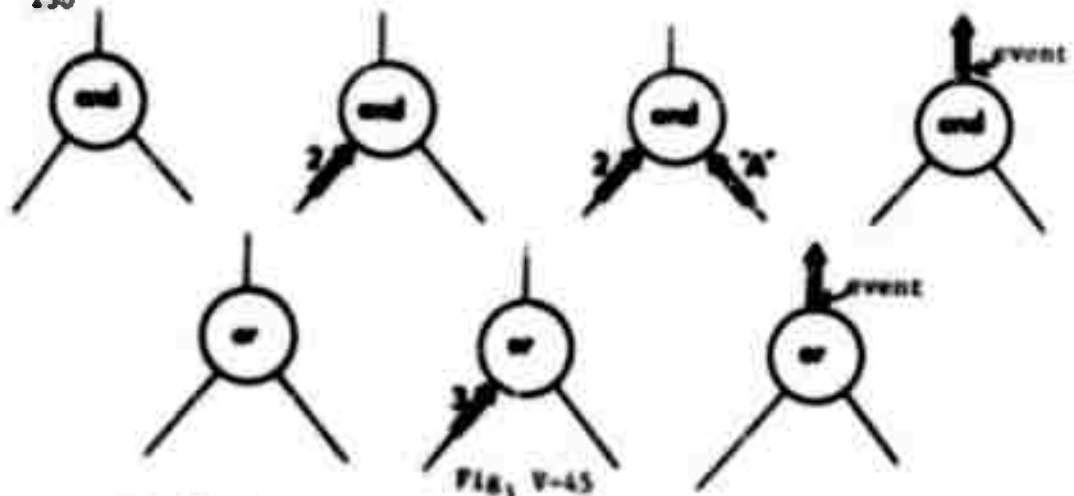
Fig. V-45

Fig. V-46

Such an expression may be used to trigger a sequential process with the combinator trigger, '□→':
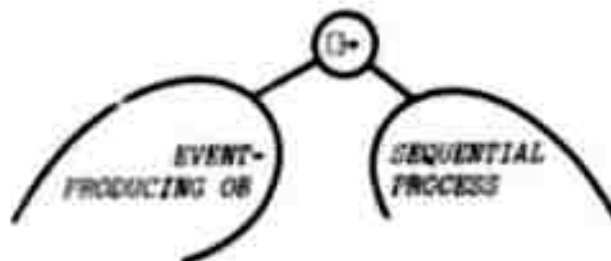
Whenever the trigger-node receives an event from the left, it triggers the sequential process on the right (fig. V-47).

A sequential process is considered as a sequence of statements. Whenever such a sequential process is triggered, the statements are executed sequentially. A sequential process may be connected to its DCPL environment by communication paths (fig. V-48). Some statements may, when executed, send some values on these paths; some other statements may only be executed after they have received a value from such a path. If the control reaches such a statement before the arrival of the required value, the control stays pending in this statement until the value does arrive: the servicing of the communication paths in a sequential process may be viewed as some kind of input/output operations.

A sequential process may be DCPL-like; in this case the binding of variables is performed as already described (each variable-node sends a request which is picked up by the corresponding binder, etc ...). However it may be useful to embed in DCPL some subset of a usual sequential programming language; the programs in these languages being considered as sequential processes, the DCPL environment serves as a host system. In this case a special binding process

132

trigger-node

sequential
process

event →

sequential
process

control
signal

sequential
process

Fig. V-47

ENVIRONMENT

information-path
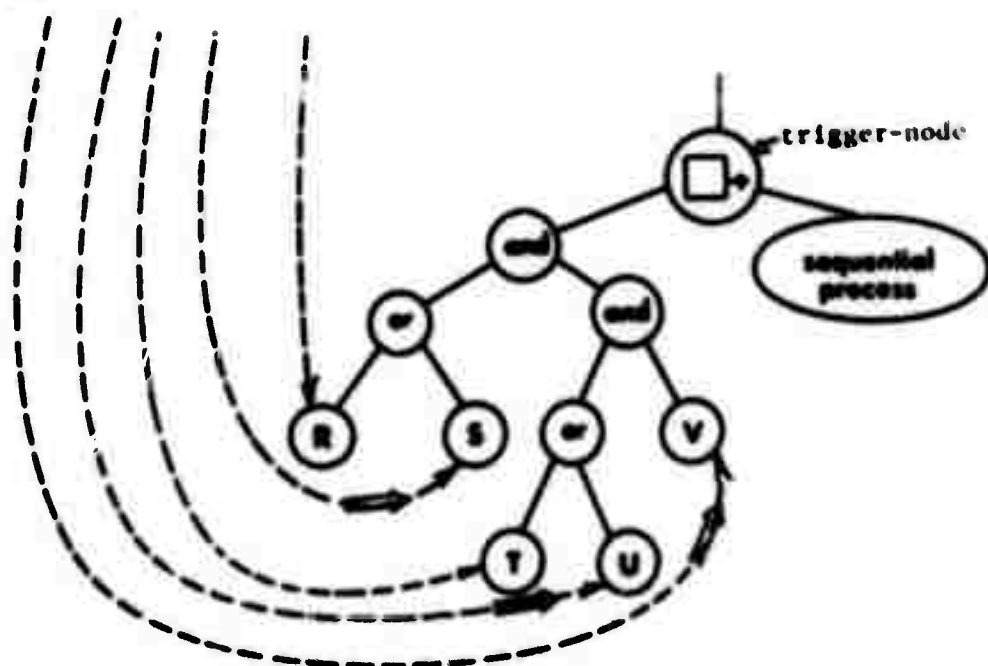
sequential
process

Fig. V-48
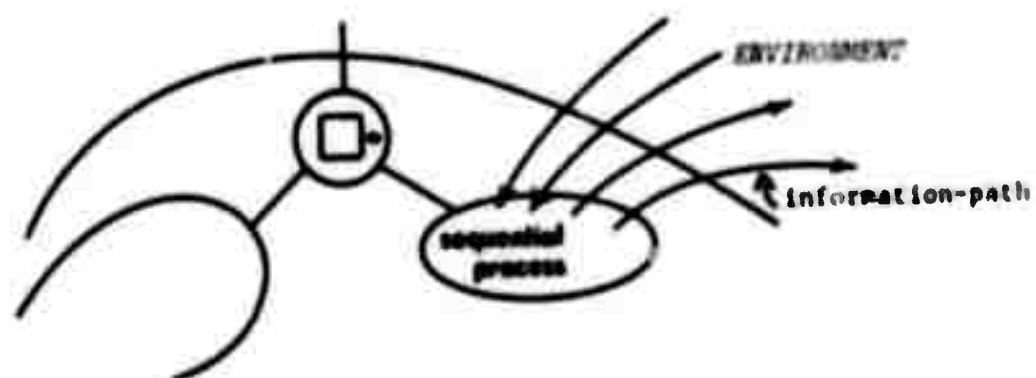
must be worked out in order to bind the sequential processes to their
environments.

V.6.2 Cells.

Simple or structured cells may be used in DCPL. A cell is
declared with its type: *INTEGER X* , *LOGICAL Y* etc ... . Each cell
is implemented as a node in a computation tree (fig. 7-49). The
scope of a cell declaration is limited to the portion of the subtree
under the declaration node which is not superseded by another cell
declaration with the same name. In order to assure that no attempt
of retrieving a value from a cell will occur before the value is
stored, these actions are to be ordered and synchronised. For this
reason, in DCPL a cell can only be accessed from a sequential process.
There is an assignment operator ':=' to assign a value to a cell:
the value is retrieved whenever the name of the cell is referred to
in an expression.

V.6.3 Recursive procedures.

Let us try to implement a recursive procedure like the
factorial function. For instance we may suggest for n! something like

$$(\underline{LAMBDA}\ X((X=0)|1;X \cdot FACT(X-1))^{a)})$$

or more specifically we may suggest for 3!

$$("(\underline{LAMBDA}\ X ((X=0)|1;X \cdot FACT(X-1)\ ))" \cdot FACT;FACT(3))$$

In this program, several occurrences of the ob assigned to *FACT*
are to be embedded one in another (fig. V-50). At some point the

---

a) '*IF*' and '|' denote the same combination.

Fig. V-49

Fig. V-50a

Fig. V-50b

embedded generations
of FACT

deleted
nb

Fig. V-10c
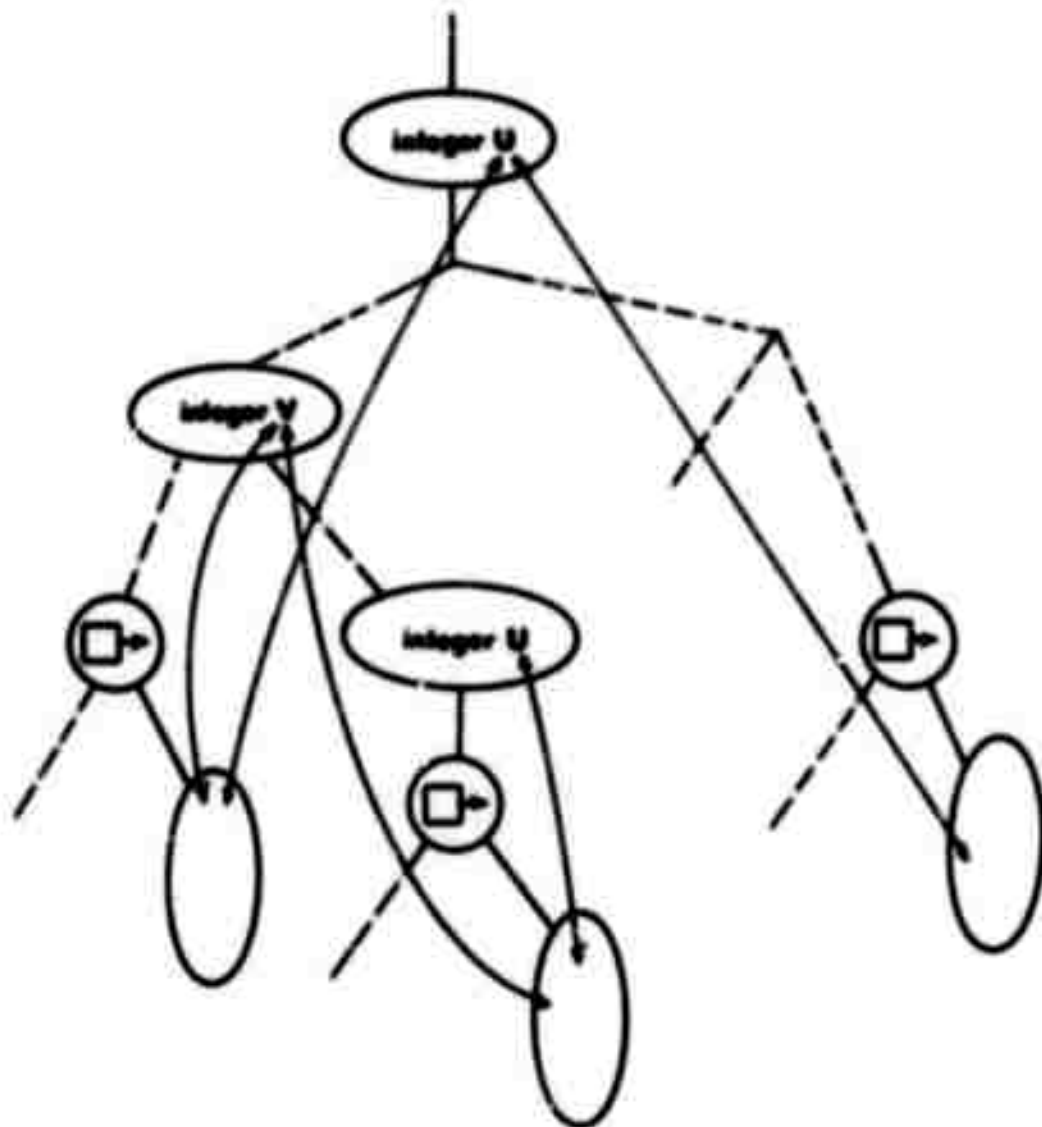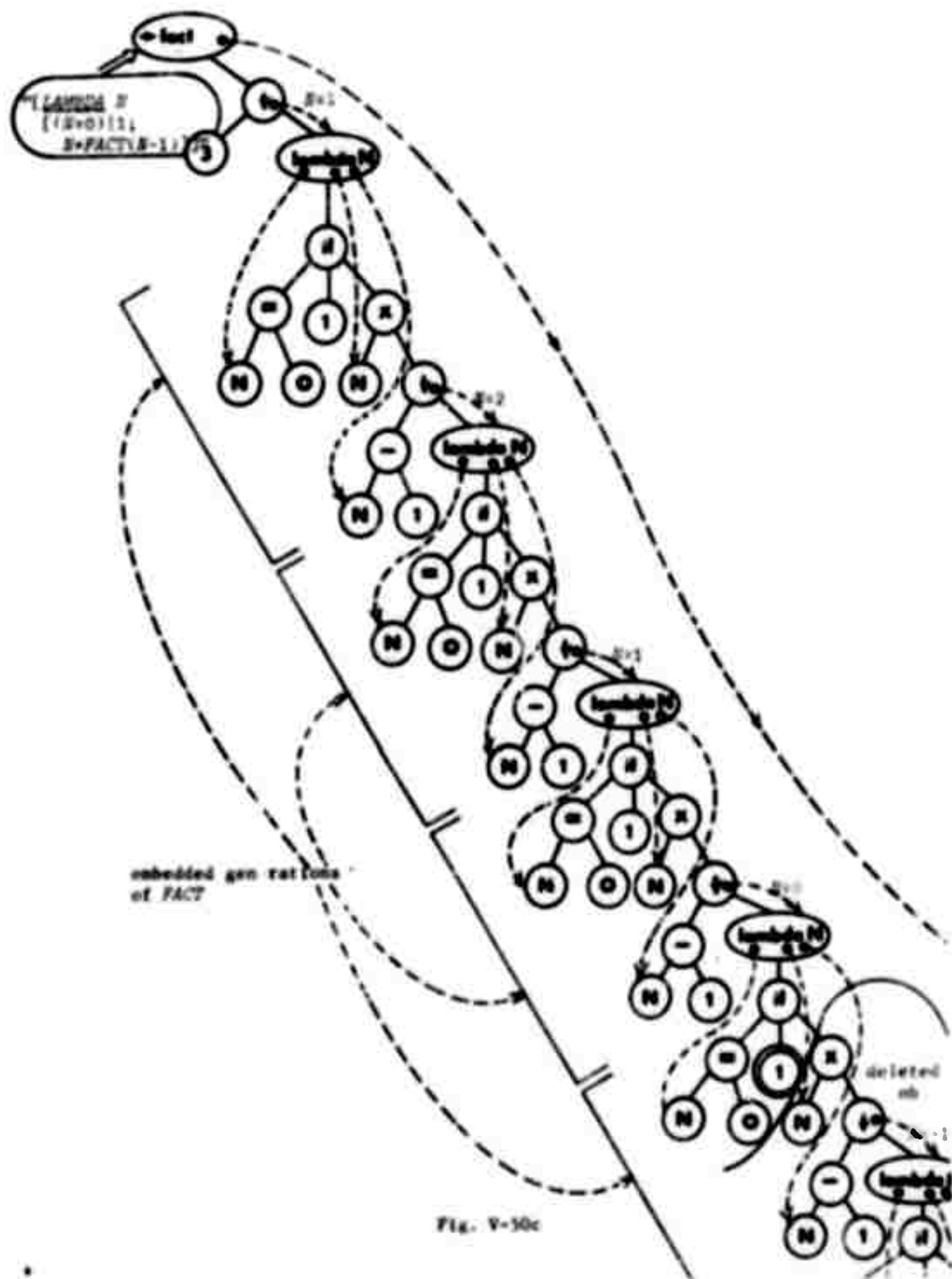
variable N will be locally equal to zero and the right subtree
$N*FACT(N-1)$ of the corresponding 'IF' combinator will be disregarded.

However such a program presents a major shortcoming: the
implementations of the different occurrences of *FACT* are in no way
synchronized with the computation itself; thousands of generations
of *FACT* may be needlessly implemented.

With the combinator '*IMPL*' (implement) we can master the imple-
mentation of a procedure. '*IMPL*' stops any request reaching it on
the right. Whenever it receives an event on the left, it frees the
stopped requests. The variable-nodes on the right may then receive
their arguments (fig. V-51).

In the program

("(*LAMBDA* X ((X=0)|1;X*(X *IMPL* FACT)(X-1)))"*FACT; FACT(1))

the implementation of *FACT* is synchronized with the availability
of the variable X. In the following program an implementation of
FACT is only performed if the corresponding test produces a *FALSE* value

("(*LAMBDA* X *LET* T (((X=0)|TRUE;(T-1|FALSE)))|1;X*(T *IMPL* FACT)(X-1)))"*FACT;
                                                                        FACT(1))

Some other programs would permit a fixed amount of "look-ahead".

V.6.4 Paths of information: alpha-variables.

Alpha-variables (as for instance αf, αT, $\alpha^{\prime}WME$ etc...)
allow the construction of alpha-paths along which an indeterminate
number of arguments may be sent. As far as ob-construction and
binding are concerned, alpha-variables have exactly the same behavior
as variables (there is one exception, see the "cyclic behavior"
subsection). However, passing an alpha-value never results in the

"LAMBDA N
((N=0)1:N=(N IMPL FACT)(N-1

Fig. V-51

deletion of a node; as a result, alpha-variables allow a computation tree to carry out repetitively a same computation.

An alpha-variable may successively receive arguments which it passes along, up the tree, as alpha-values (fig. V-52).

An operation-node may successively receive an alpha-value on each incoming path and produces each time a resulting alpha-value (fig. V-53). If an operation-node has an incoming alpha-path and an incoming simple-path, it uses the value received on the simple path iteratively with the successive values received on the alpha-path (fig. V-54).

Cyclic behavior.

Iterative and cyclic behaviors may be modeled with alpha-paths. In particular a setdown-node may bind a setup-node constituting a recirculating path. For instance, the program

$$NY \ a \!\!\!/ \quad (1 \rightarrow n\!\!\!/; \ a\!\!\!/ \rightarrow n\!\!\!/; \ a\!\!\!/ \rightarrow n\!\!\!/ + 1; \ \phi)$$

is an infinite loop sending out on the path $a\!\!\!/$ the successive integers starting at 1 (fig. V-55).

Interlinked coroutines.

Alpha-paths may interconnect coroutines. In the following example, the procedure PROC1 sends messages to procedures PROC2 and PROC3:

$$NEW \ a\!\!\!/ \ \dots \ PROC1(;a\!\!\!/) \ \dots \ PROC2(a\!\!\!/) \ \dots \ PROC3(a\!\!\!/) \qquad (fig. \ V-56).$$

In order to associate together procedures it is useful to have a particular dyadic combinator compose '∘' ; whenever compose receives a value from one of its two sons, it produces this value and deletes the other son. Our previous example may be written:

140
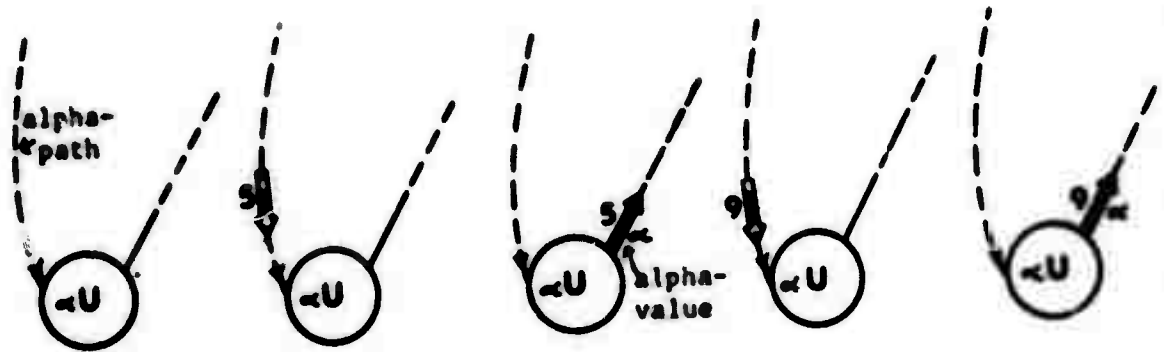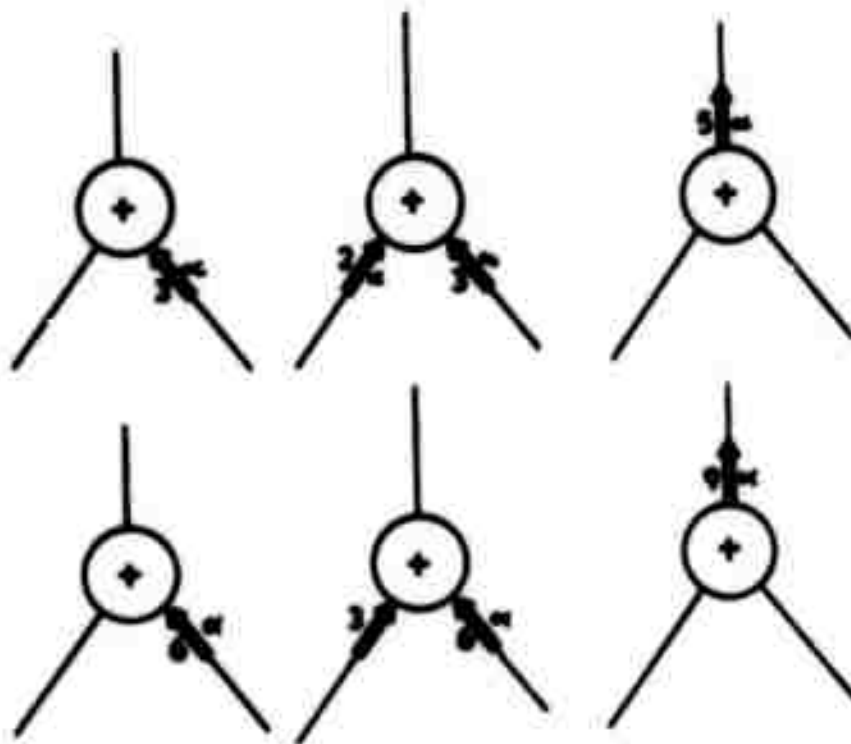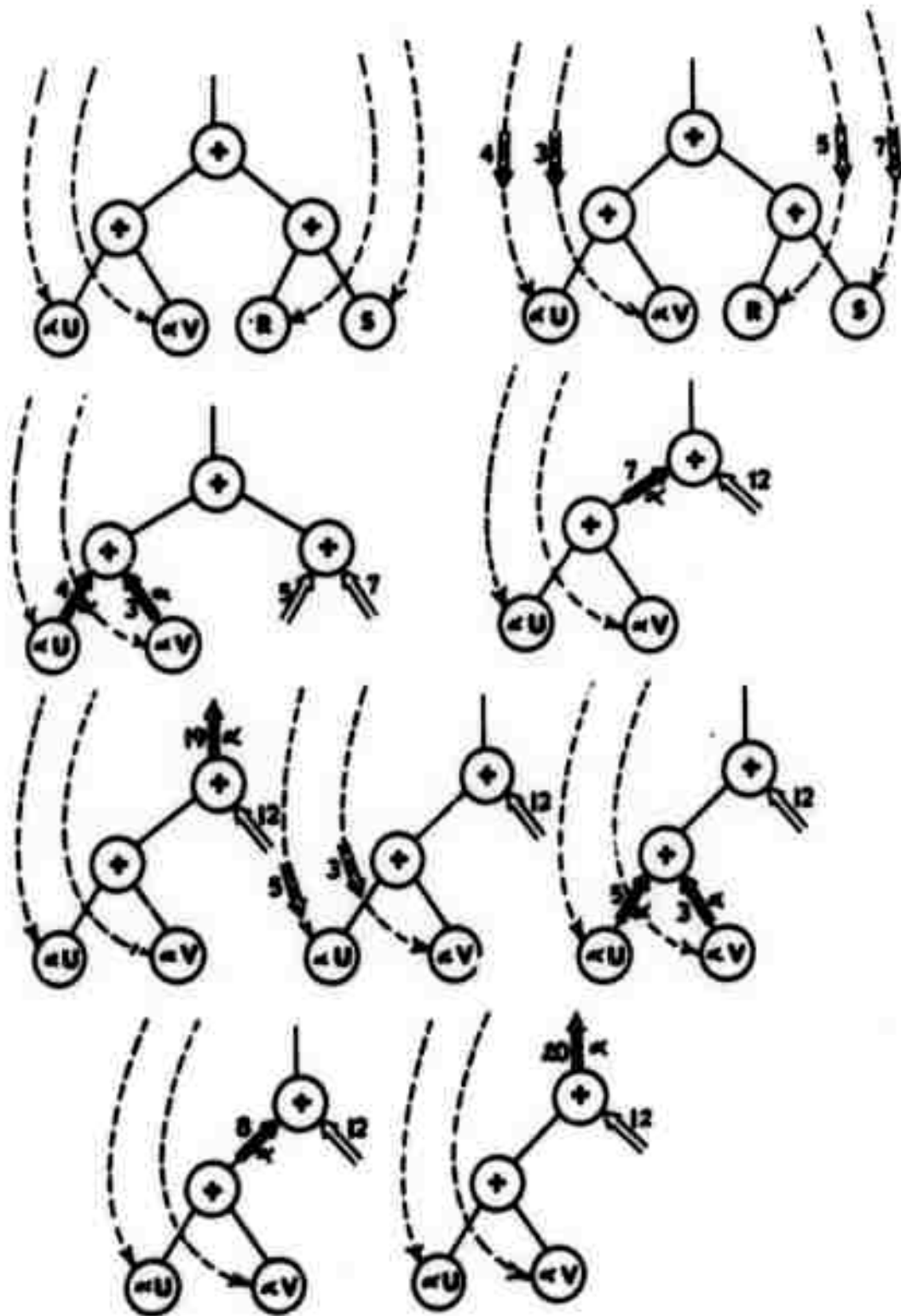
Fig. V-52

Fig. V-53

Fig. V-54
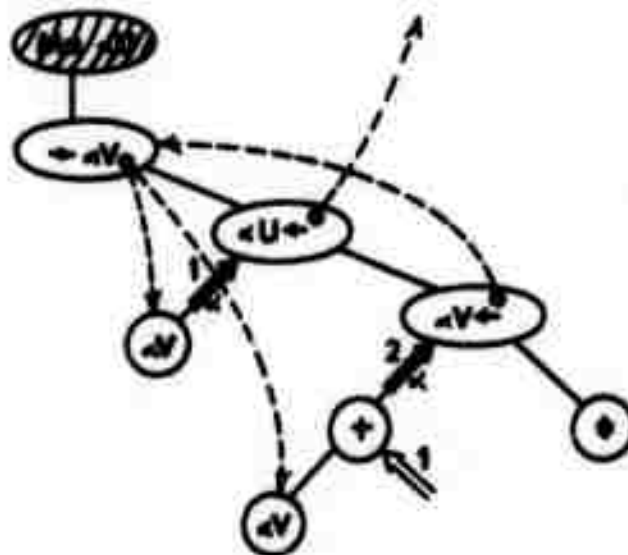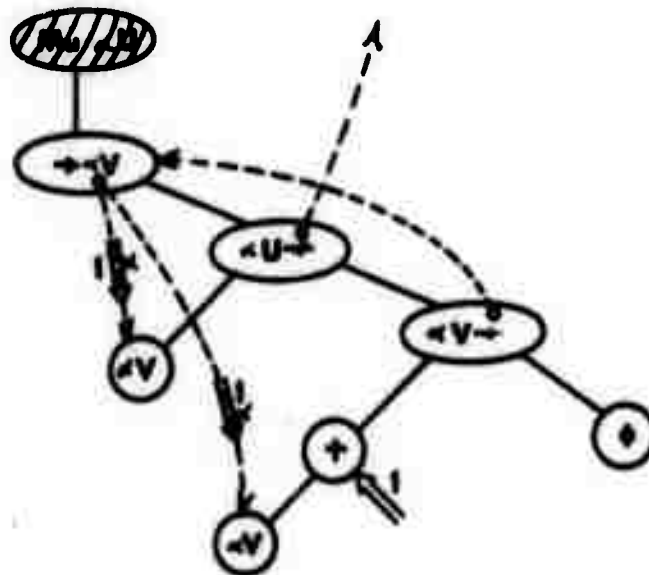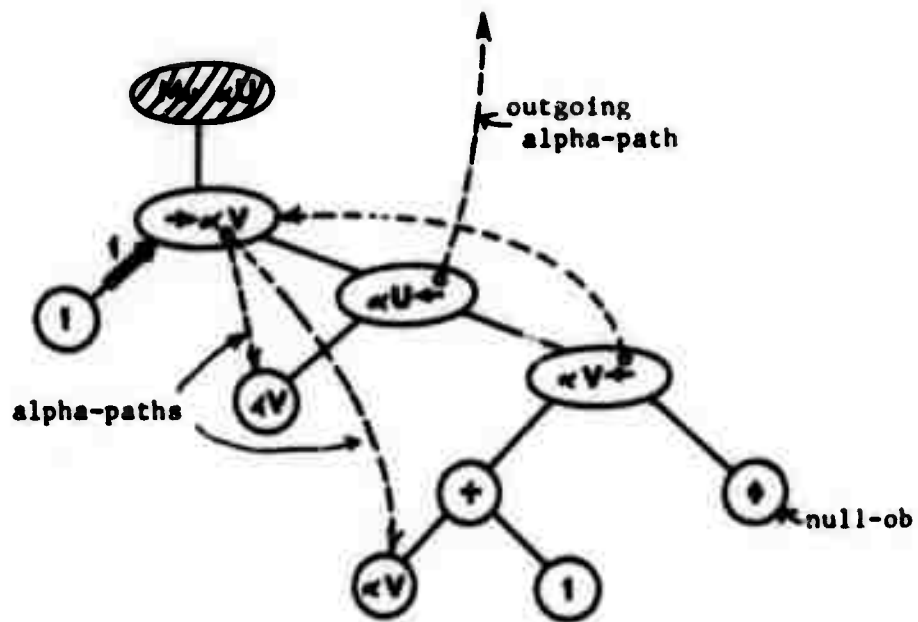
Fig. V-55a,b,c
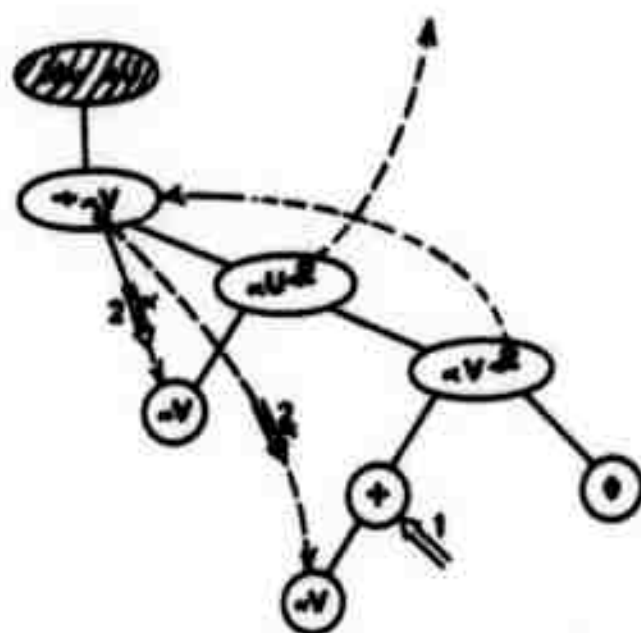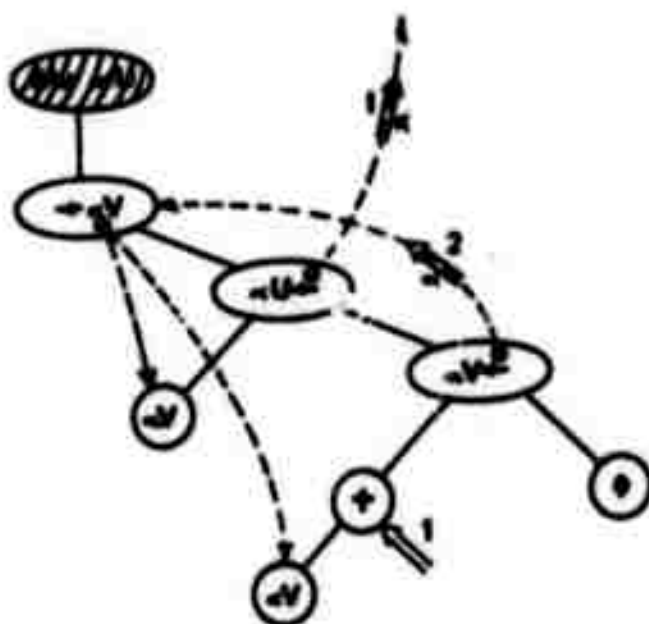
Fig. V-55d,e
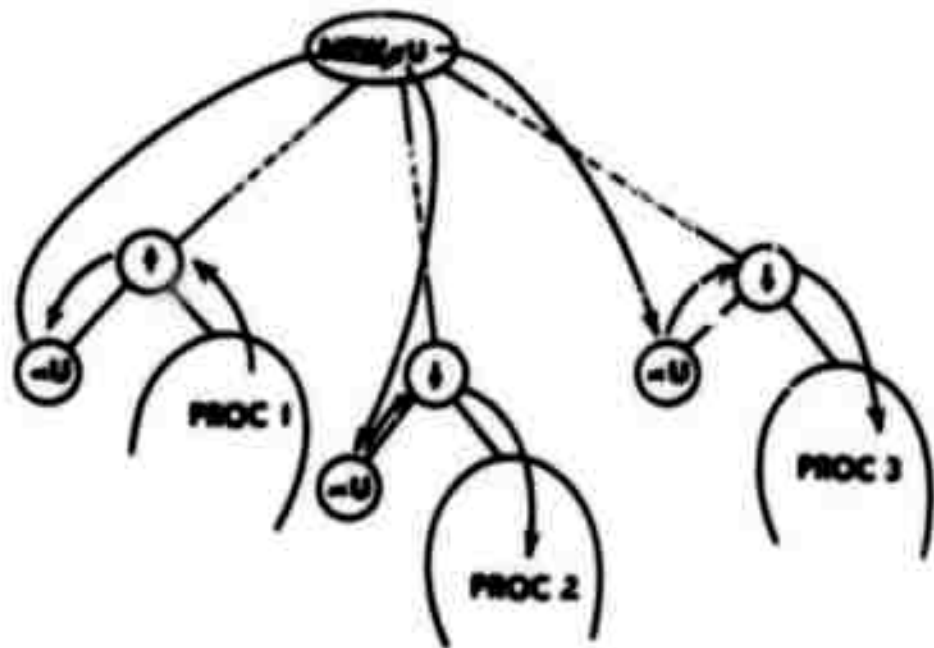
Fig. V-56
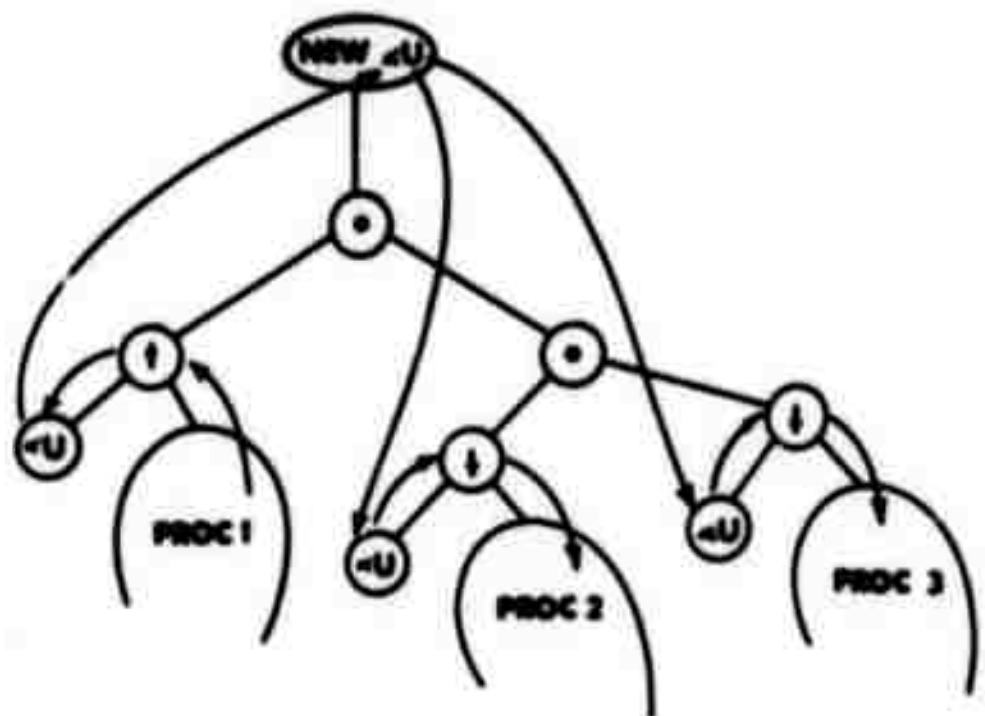
Fig. V-57

[*LET* αU   *PROC1*(;αU)·*PROC2*(αU)·*PROC3*(αU)]   (fig. V-57).

In the following example, any of the procedures *PROCU*,*PROXY*, *PROCV* can send information which is duplicated and sent to the two other procedures:

[*LET* αU,αY,αW   *PROCU*(αY,αW;αU)·*PROCV*(αU,αW;αY)·*PROCW*(αU,αY;αW)]
(fig. V-58).

Whenever a procedure is interfaced with its environment, an alpha-variable in the environment may be bound with another alpha-variable or with a simple variable.

In the former case the path in the environment is connected to the path in the procedure; any number of arguments may use these paths.

In the latter case the procedure must be recursive, the path in the environment is connected successively to each generation of the recursion[a] (fig. V-59). If the path goes into the procedure the incoming arguments are queued on reaching a sendown, constituting a First-In-First-Out queue. Whenever such a sendown receives a request from a lambda-node, the first element in the queue is popped out and sent as a reply to the lambda-node (fig. V-60).

Queue declaration.

It is possible to declare the end of a path as a queue: αU *QUEUE* Y; the scope of such a declaration is the subtree under it. Whenever Y is referred to in a sequential process, the first of the queue is popped out; it may then be forwarded to the environment of the sequential process (fig. V-61).

---

a) each generation receives one argument from the path.
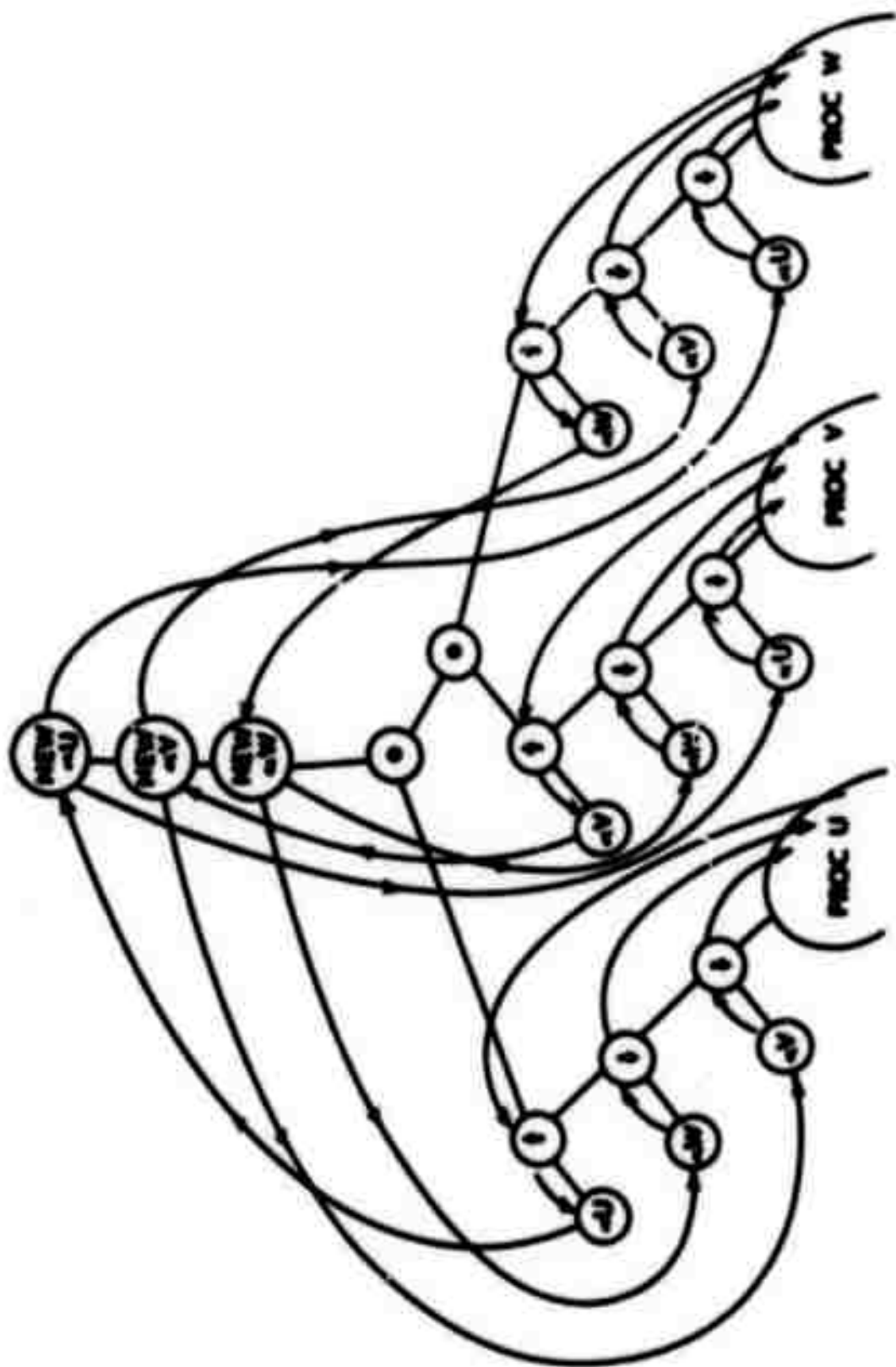
FIG. V-38
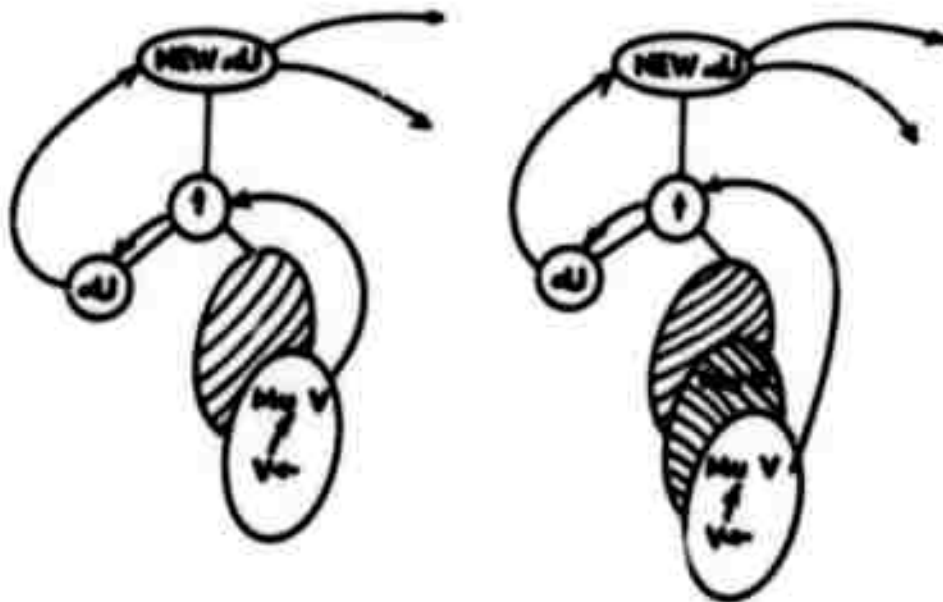
Fig. V-59
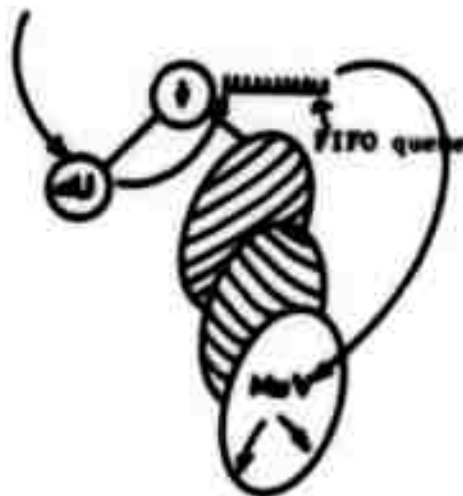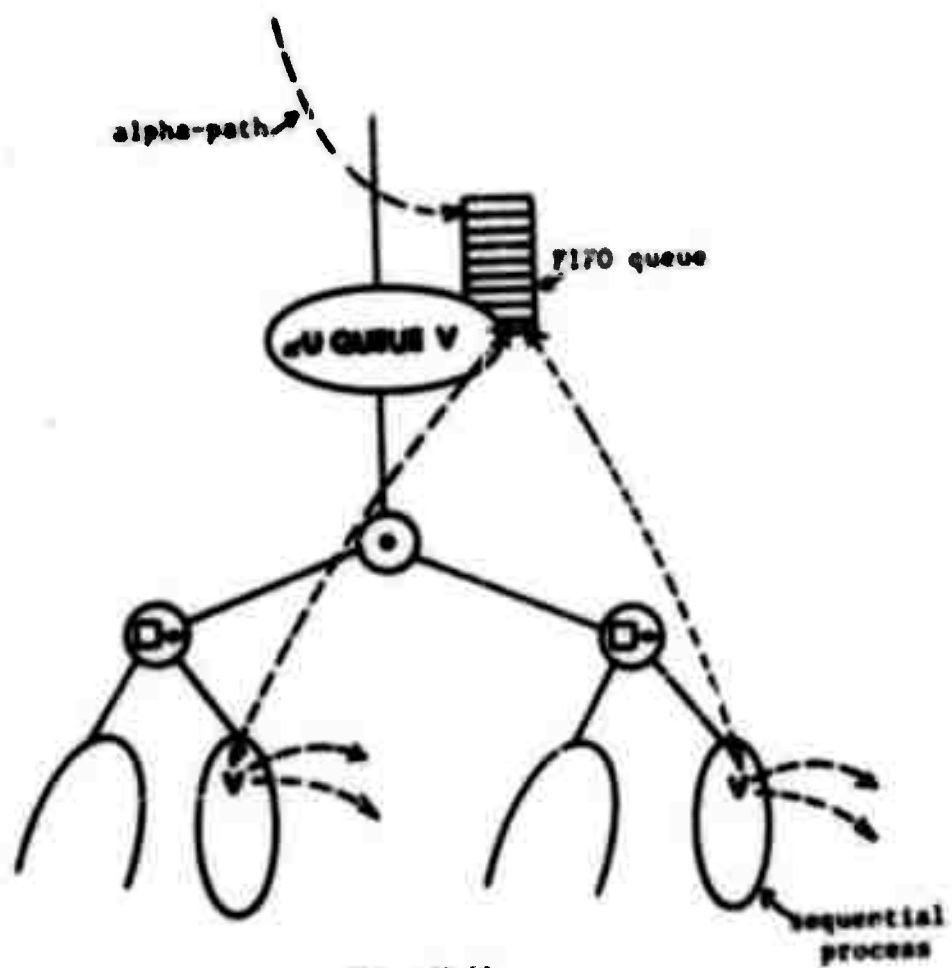
Fig. V-60

Fig. V-61

V.7   Summary.

Let us review the main notions encountered in this chapter.

1. A computation is viewed as a tree whose nodes are automata and whose edges are directed channels of information.

2. Each node has an address in a particular space; any node may send a message to any other node whose address it knows.

3. The universe of values and operations is left open-ended. To each value corresponds a value-node and to each operation an operation-node.

4. A few replacement rules allow to have a linear representation of any DCPL computation tree.

5. A binding process superimposes a graph structure to the computation tree: each variable-node sends a request (containing its address) up the tree. This request is picked up by the corresponding binder.

6. Setdown '↓' allows to send a value "down the tree". Setup '↑' allows to send a value "down the tree" and to the environment; its scope is delimited by the binder NEW.

7. A variable in a procedure may either receive an argument from the environment or send an argument to the environment. In the former case, the variable is bound inside the procedure with a LAMBDA binder-node and the procedure is interfaced with its environment through a node sendown '↓'. In the latter case, the variable is bound inside the procedure with a MU binder-node and the procedure is interfaced with its environment through a node sendup '↑'.

8. The description of any computation tree may be transmitted as a pseudo-value. Whenever a variable receives a pseudo-value,

the corresponding computation tree is implemented in the place of the variable. "X·1" is a pseudo-value describing the tree X·1.

9. A procedure may be modified through interaction with its environment, and then be transmitted as a pseudo-value in order to be implemented in some other place.

10. The different parts of a conditional expression are evaluated concurrently. However, as long as the result of the test is not available, the outgoing messages of the alternate subtrees are picked up and retained.

11. With the nodes *AND* and *OR* it is possible to have an expression produce an event.

12. '$\square$·', trigger, is used to trigger sequential processes, and *IMPL* to master the synchronization of the implementation of the different generations of a recursive procedure.

13. Cells may be declared; they may only be accessed from a sequential process.

14. Alpha-variables allow the construction of alpha-paths on which an indeterminate number of arguments can be sent. They permit cyclic behavior, interlinked coroutines and implicit or explicit FIFO queue servicing.

It should be noted that the concepts described here do not constitute a set of primitive concepts.

Moreover, the behavior of the nodes might be modelled in very different ways. We consider the general notions discussed as more important than the described schemes themselves.

PART THREE

MACHINE ORGANIZATION

CHAPTER VI

IMPLEMENTING DCPL : MACHINE ORGANIZATION

VI.1 Introduction.

Any machine which is intended to actualize and to carry out a DCPL program must respect and be able to express the following properties:

1. A computation in DCPL is represented as a tree whose nodes are automata and whose edges are directed channels of information.

2. Each node has an address in a certain address space; any node may send a message to any other node whose address it knows.

3. During its life, a computation tree evolves, growing and shrinking.

The third requirement may appear to be the most stringent one. We would like to have some host physical structure which can grow and shrink as the guest computation it contains evolves, synthesizing itself from the components available in the surrounding milieu, like a DNA molecule governs its own reproduction. In [24], Von Neumann used a similar analogy, comparing a computing process to a self-reproducing organism. He used as supporting structure a cellular space. Such an approach for designing a machine for DCPL may be valid from a theoretical point of view, however as a result of

concentration phenomena[a], it would turn out to be too impractical.

Different kinds of machine organization are described in the sequel. In particular it is shown how large transfer rate sequential memories may be used. Moreover an organization allowing swapping in advance is discussed.

VI.2 Cellular structure using busses as communication paths.

DCPL computation trees may be implemented in the supporting physical structure shown in fig. VI-1: autonomous "active" cells may communicate to one another through a system of busses; each cell has an address and may send a message via a bus to any other cell. Each node of the computation tree is actualized by one or several cells (the amount of storage a node may need is not bound: there is indeed no limit to the number of pointers a node may contain). The edges of the tree are actualized with pointers. At any time there are cells which are free: they do not participate to any computation tree. Whenever a computation tree grows (or shrinks), free cells are allocated to that tree (or deallocated and freed). If the busses may be used concurrently, such an organization allows concurrency and distribution of control. However, since every cell is "active", such an organization would be prohibitively expensive if large computation trees were to be carried out. Moreover, since there is a lack of locality in such an organi-

---

a) As any reproducing organism, trees may grow exponentially: their embedding in a n-cellular space may therefore be problematic.
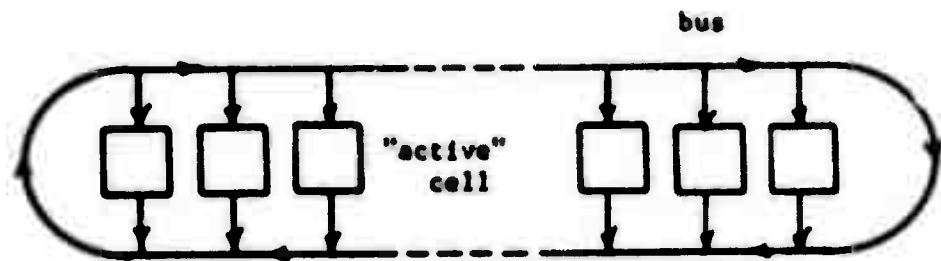
Fig. VI-1
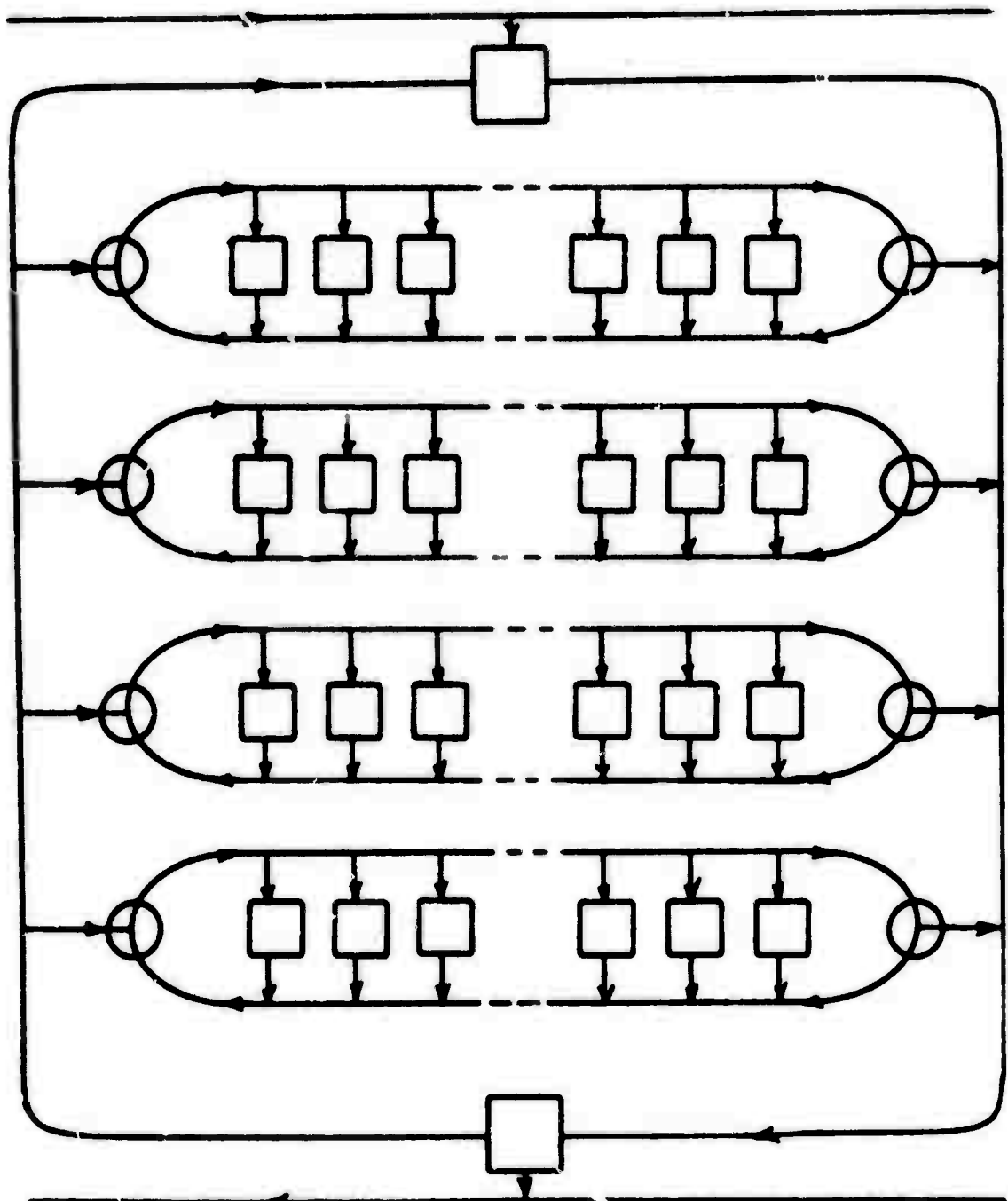
Fig. VI-2

zation, such a communication system would probably clog rapidly whenever the size of a computation would grow. In the organization displayed in fig. VI-2, we can take advantage of the locality a program may present: each cell has a hierarchy of neighborhoods. However, with this organization the cell-allocation problem may become more difficult. It should be noted that such a cellular organization with busses may be very attractive if used as the fastest level of a "memory" hierarchy: the main computation tree is implemented in some other type of organization, small parts of a computation with many local interactions being swapped into the cellular structure to be carried out.

VI.3   A machine organization with the nodes stored on a random access storage medium.

In the previous section, a node was actualized by an "active" cell. This is a straightforward approach when considering a node as an automaton. However, we may view a node behavior in the following way: a node is quiescent until it receives a message; the interaction of a node with a message addressed to it results in a new quiescent state of the node and possibly messages for some other nodes.[a] From this interpretation derives the following machine organization:

The nodes are placed on a storage medium, each node having a physical address on this storage. At any time, the addresses of the nodes for which some messages are waiting form a queue to which

---

a) Whenever a node acts spontaneously in DCPL, this action occurs right after the creation of the node; it may then be considered that a correspondent message is addressed to the node at its creation.

is associated the set of corresponding messages. One or several processors may access the nodes on the storage medium, the queue of addresses and the corresponding set of messages. Any processor may service the queue: it takes the first message, accesses the node to which the message is addressed, "computes" the interaction, stores the new content of the node, and places at the end of the queues the addresses of the nodes to which the resulting messages (if any) are to be sent.

Fig. VI-3 displays such an organization with one processor; in fig. VI-4 several processors may access different parts of the storage medium; to each processor is associated a queue; any processor may access the queue associated to the other processors and place messages in it.

VI.4   A machine organization with a sequential rotative memory.

The computation trees may be implemented on a sequential rotative memory. For instance, in fig. VI-5, a processor controls a read/write head; the addresses of the node for which some messages are waiting, and the corresponding messages, are queued; however, such a queue is not serviced on a FIFO basis: the addresses in the queue are ordered according to the order of appearance of the corresponding nodes under the head. Thus, the top of the queue contains the address of the first node to appear under the head.

One processor may service several sets of tracks, nodes being implemented on each of these sets (fig. VI-6).

Several processors may serve different sets of tracks (fig. VI-7) or may serve the same track at different locations
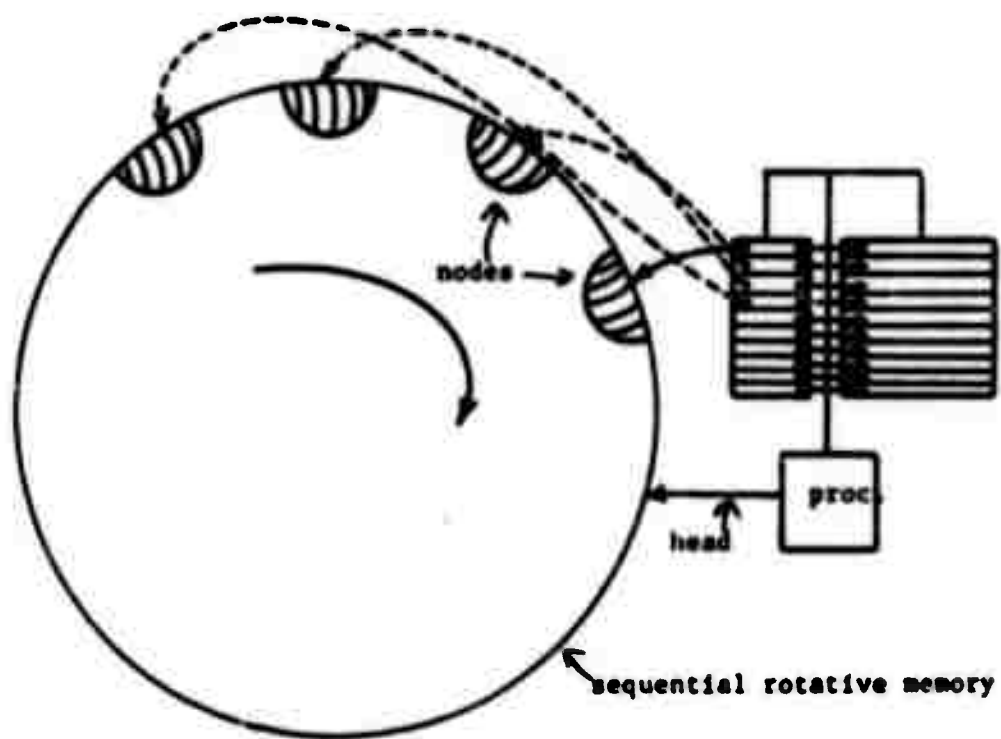
Fig. VI-3
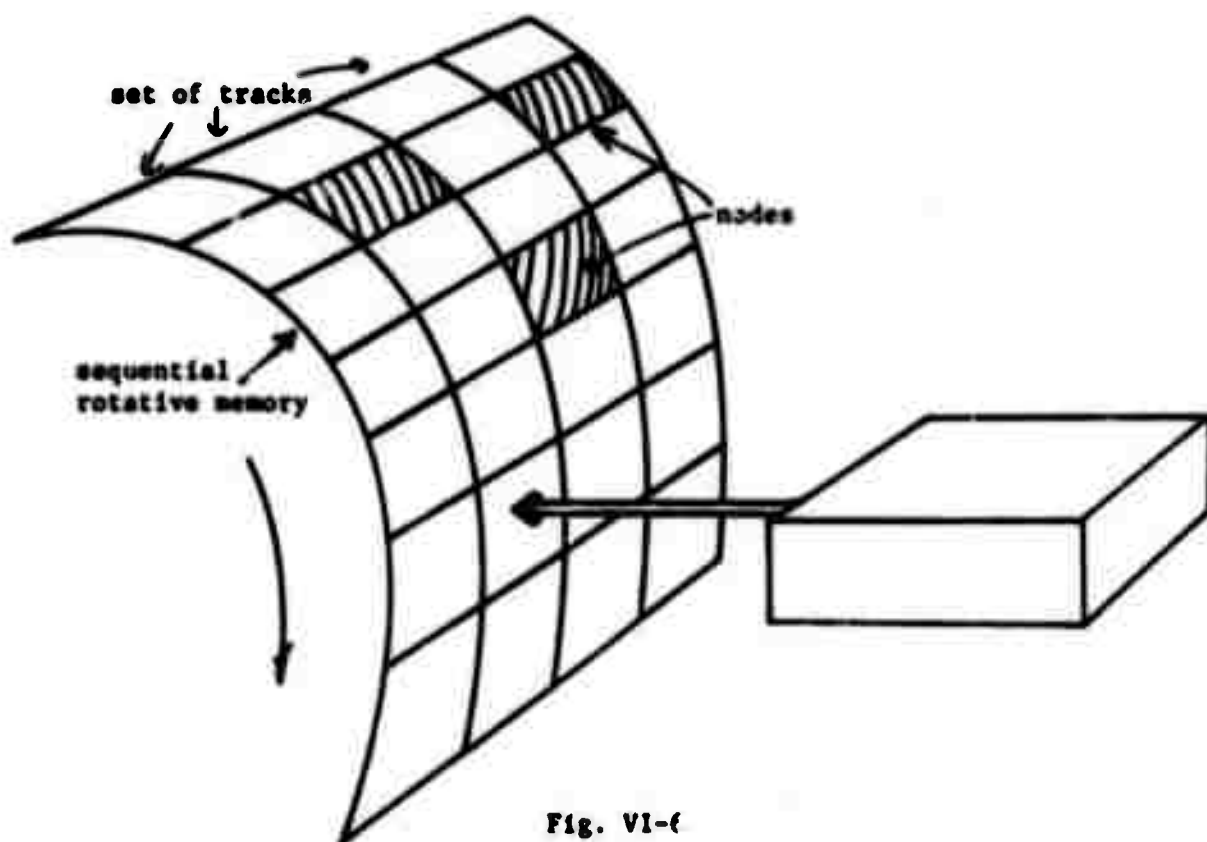
Fig. VI-4
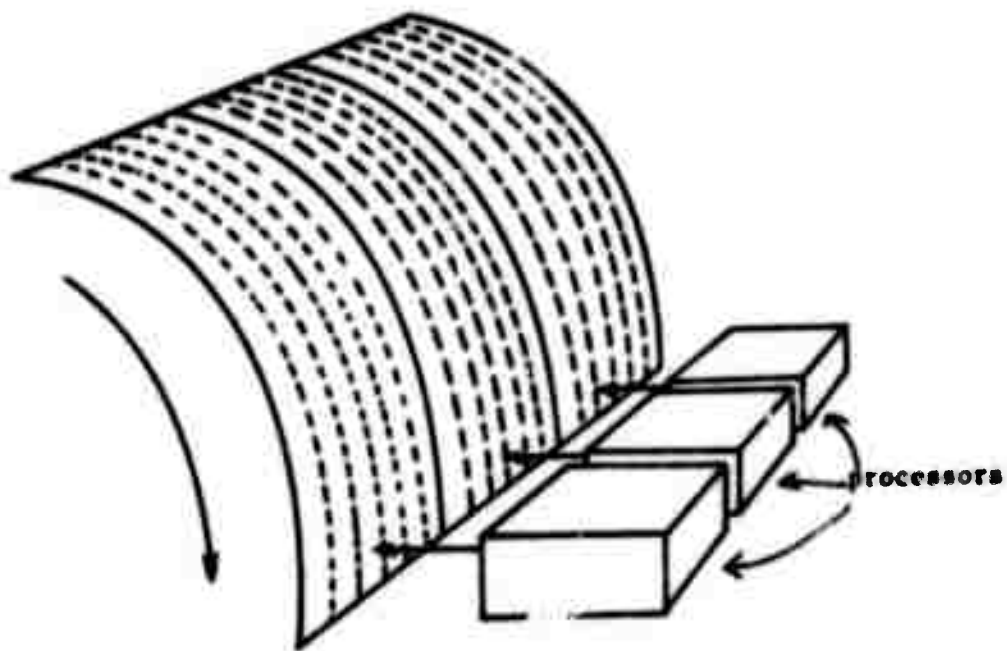
Fig. VI-5

Fig. VI-6

Fig. VI-
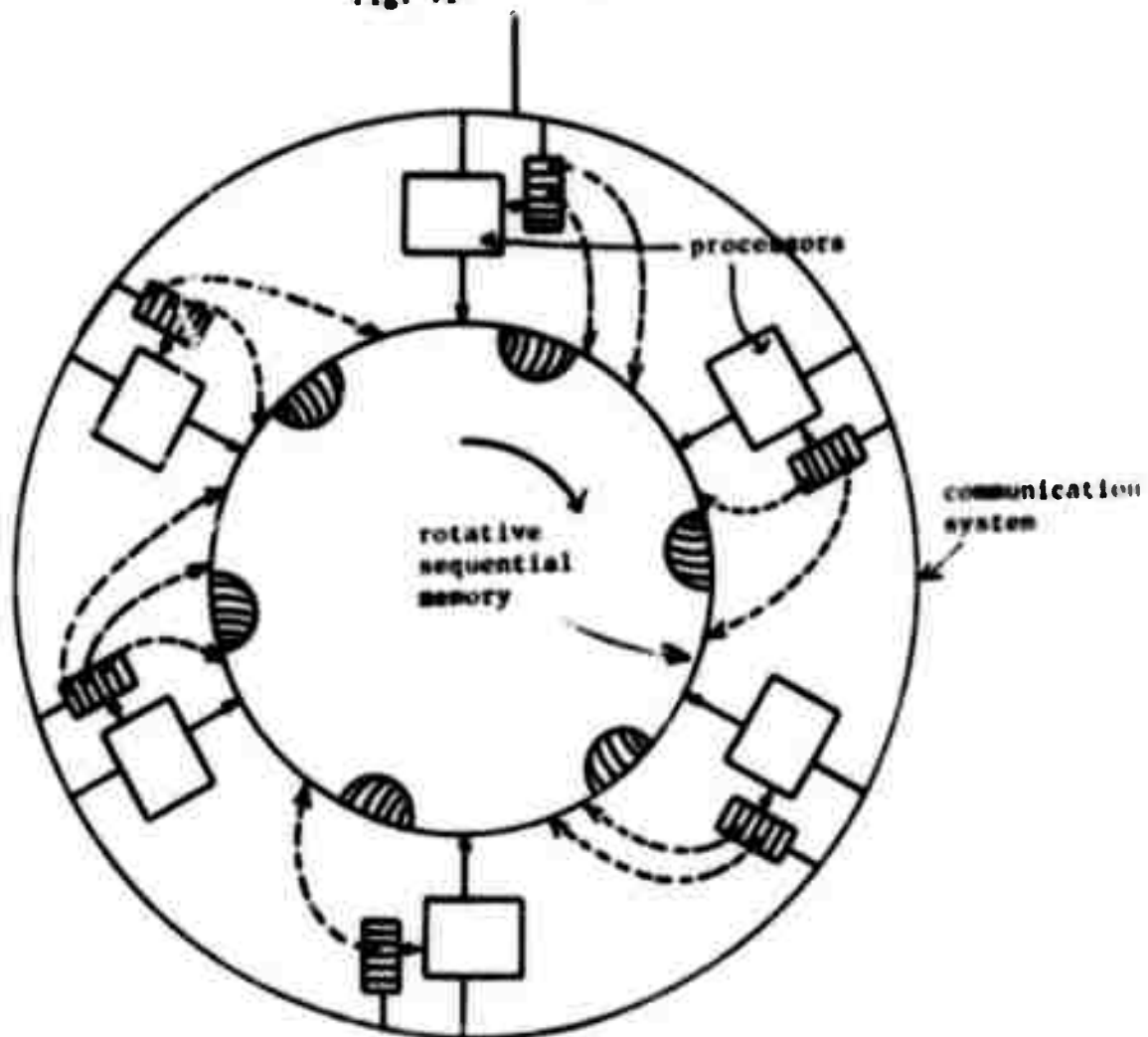
Fig. VI-8

(fig. VI-8). In both cases a communication system must link each processor to the queues associated to the other processors.

A DCPL computation tree may be implemented on a sequential rotative memory so that a son always appears under the head before its father. As a result, any request may be sent from son to father up the tree in a fraction of the rotation time (fig. VI-9). In the same way, structured cells may be implemented in order to be accessed and searched in a fraction of a rotation time (fig. VI-10).

We consider these machine organizations as being of special interest: not only sequential memories are generally rather cheap, but they also allow very large transfer rates. It becomes then possible to take advantage of the possibility of having very fast processors (for instance, a processor designed on a wafer with a cycle time of 20 ns.).

VI.5 Hierarchy of memories.

Whenever it is desired to take advantage of a very fast processor, a hierarchy of random-access memories is used (fig. VI-11): the smaller a level, the faster its memory([25]).When a memory hierarchy is used, the processor works in the fastest level, programs being swapped back and forth between the various levels. It is hoped that the processor would feel that the whole memory is as fast as the fastest one. Unless many iterative computations are expected to occur in the fastest level, it is necessary to have between any two levels a transfer rate large enough to "feed" the processor.

A large transfer rate may be obtained by taking at each level a large block as a unit of transferable information (the slower the level, the larger the block).

moving direction
of the information
stored, relatively to
the head

the nodes are
accessed this way .

head

Fig. VI-9

structured cell

Fig. VI-10

M_1 fastest level

M_2

M_3 slowest level
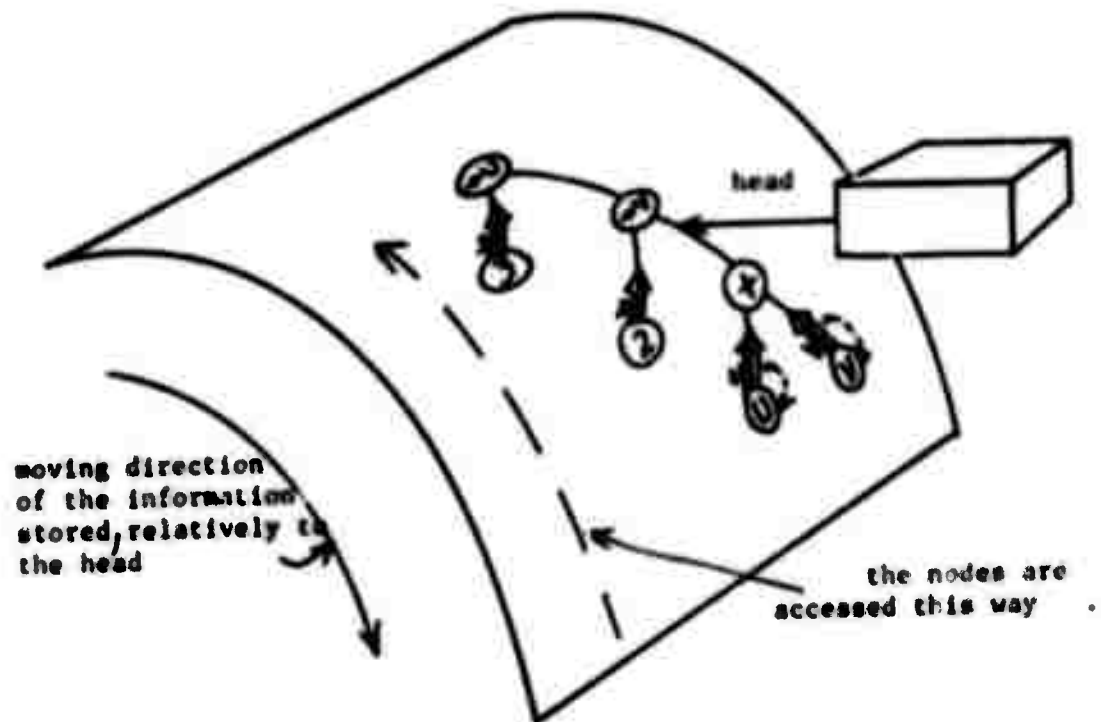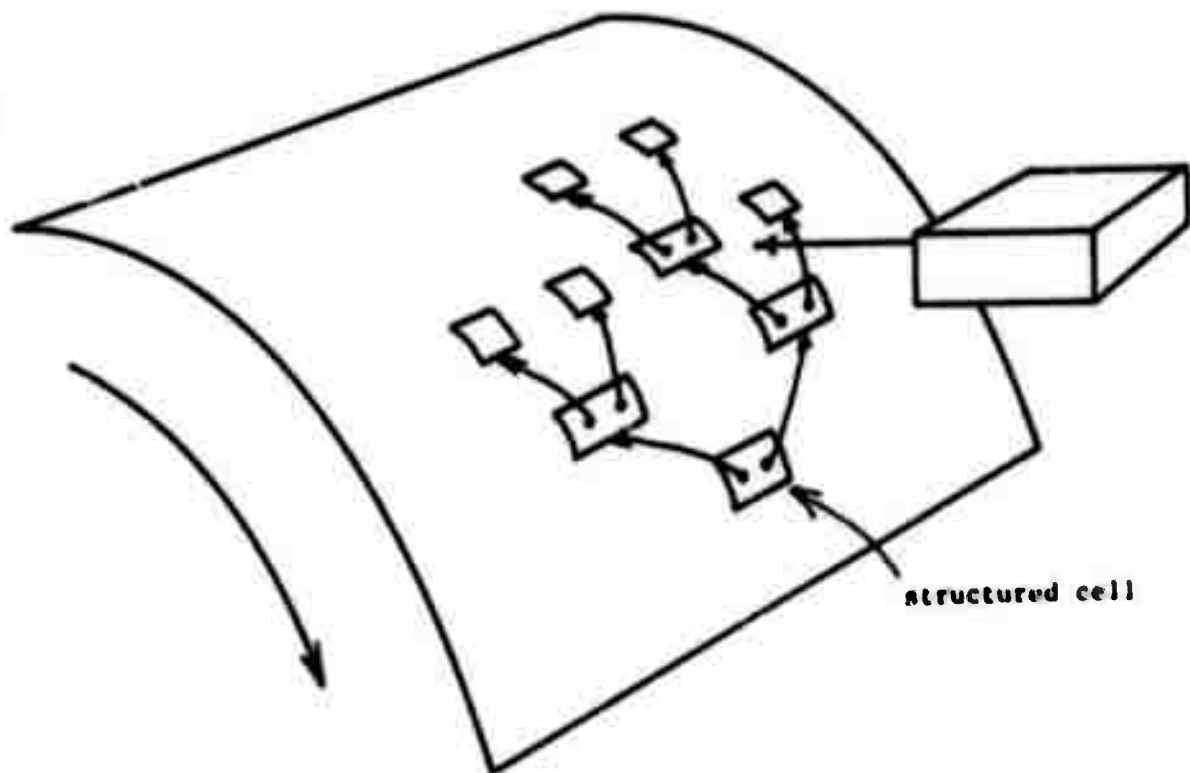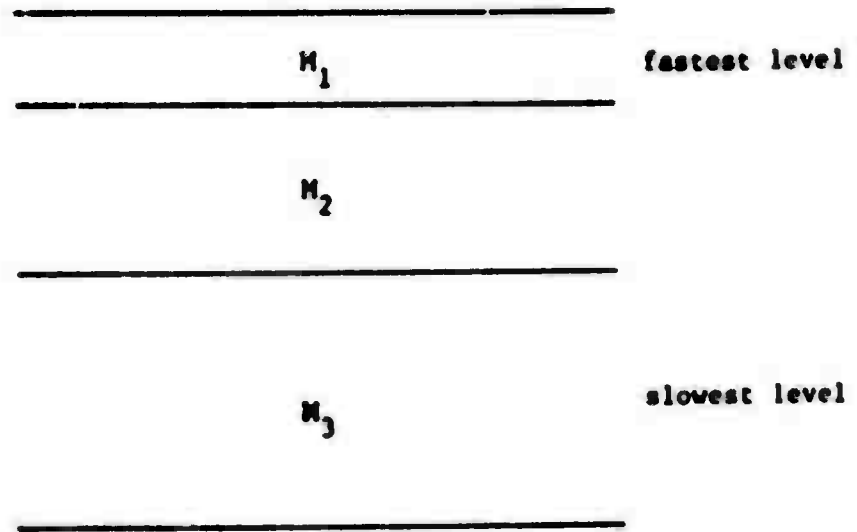
Fig. VI-11: A memory hierarchy

addressee table

sequential connected process

entry filled with an argument

empty entry

event table
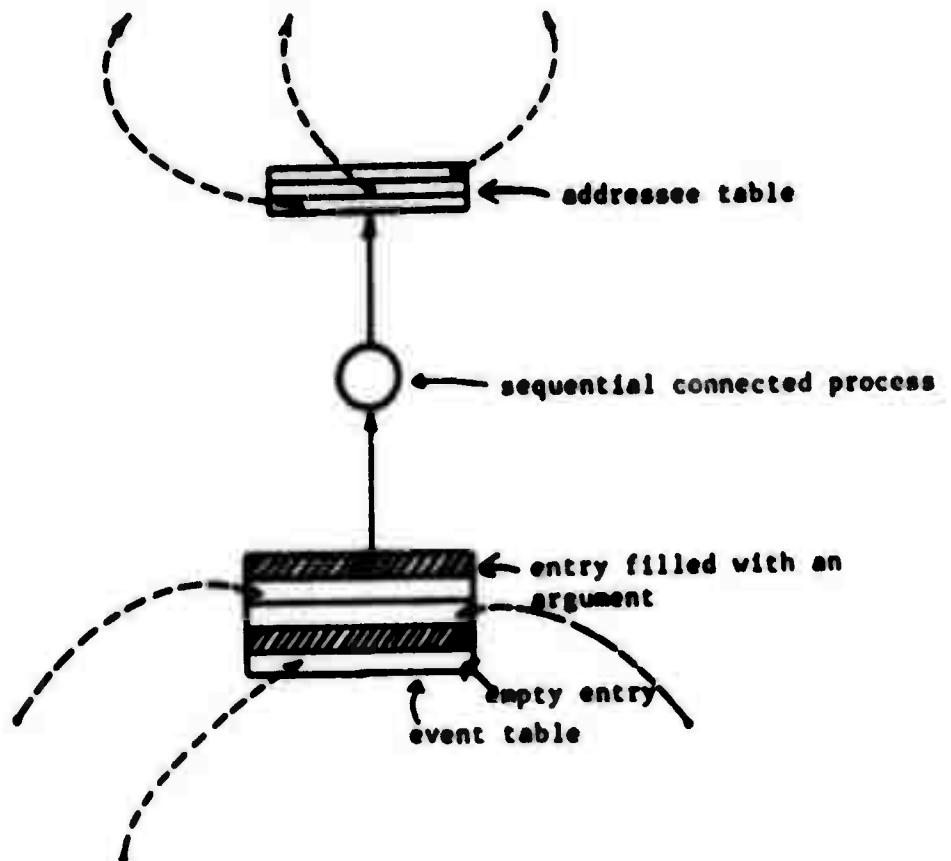
Fig. VI-12: A sequential connected program element

One may believe, however, that only a few words in such blocks would be really used. For this reason Jack Dennis suggests in [4] that information should be moved on demand with the word as information unit, a large transfer rate being assured by performing many computations in parallel.

In order to be able to use efficiently large units of transferable information, we introduce the notion of connected program. A computation is said to be connected if:

1. it can be activated whenever a certain set of arguments is available, and then completed without the need of any external information;

2. no partial result must be supplied to its environment before the completion of the computation.

As a result, a connected computation may be brought in the fastest level of a memory hierarchy and completed without the need for any information from some other level; moreover, there is no advantage in breaking the connected computation into pieces, carrying out separately the various pieces.

The notion of simple connectedness arises because a connected computation tree may be too large to be swapped in the fastest level of the memory hierarchy and have some of its parts not being connected.

A computation is said to be simply connected if it is connected and if any subcomputation it contains is connected. As a result, a simply connected computation may be brought by parts in the fastest level.

Consider the following examples in DCPL:

1. the computation tree (fig. V-18):

    NEW V $(2 \to U; V \to U+1; U+V) + (3 \to U; U+V)$

is simply connected.

2. the computation tree (fig. V-19):

    NEW U,V $(U \to 5; [U+V] \to Z; Z \to U) + (V \to U+1; U)$

is connected, but not simply: the ob $(U \to 5; [U+V] \to Z; Z \to U)$
is not connected since the argument V can only come after the
argument U has been sent to the second expression.

In DCPL it is easy to determine simply connected parts
by taking for instance the largest parts which do not contain
any setup '$\to$'(these simply connected parts are by no means maximal).

Whenever a computation tree is simply connected, it is
irrelevant for its environment whether the computation is carried
out with a distributed or a sequential control.

VI.6 A computation as a network of sequential connected programs.

A _sequential connected program element_ may be described by
giving (fig. VI-12):

   1. an avent table, each entry of which may receive an
      argument from the environment,

   2. a sequential connected process which may be applied
      to the arguments contained in the avent table when this
      table is filled.

   3. addresses of some other avent table entries to which
      a produced value is to be sent (addressee table).

Remark: Any reference in a sequential connected process is local

to its program element. Any binding with the environment involves
only the event and the addressee tables.

Whenever an argument is sent to an entry of the event table,
this entry is filled with it. The sequential process is triggered
when all the entries of the event table are filled. At the comple-
tion of the computation, the produced values are sent to some entries
of some other program elements, according to the addressee table.

A computation may be considered as a network of sequential
connected program elements (fig. VI-13). Fig. VI-14 displays a
machine organization which can carry out such a computation. The
configuration is similar to the one of fig. VI-5: the network of
sequential connected program elements is stored on a sequentially
accessed rotative memory; at any time there are messages waiting
for an entry in an event table; the addresses of these entries
are ordered in the order of appearance of these entries under the
head. Whenever an event table containing an entry for which a
message is waiting passes under the head, the table is examined.
If, in addition to the entry for which there is a message, a non-
filled entry remains, the message is just placed in the event table
at the proper entry. On the contrary if, with the exception of the
entry for which there is a message, the event table is filled, the
corresponding program element is swapped into the scratch-pad
memory (see fig. VI-14) and executed there; at its completion,
the addresses of the event tables to which the messages are to
be sent are placed together with the corresponding messages,
at the proper places in the queue.

Let us suppose now that some of the sequential processes

program element

Fig. VI-13

processor

Fig. VI-14

167
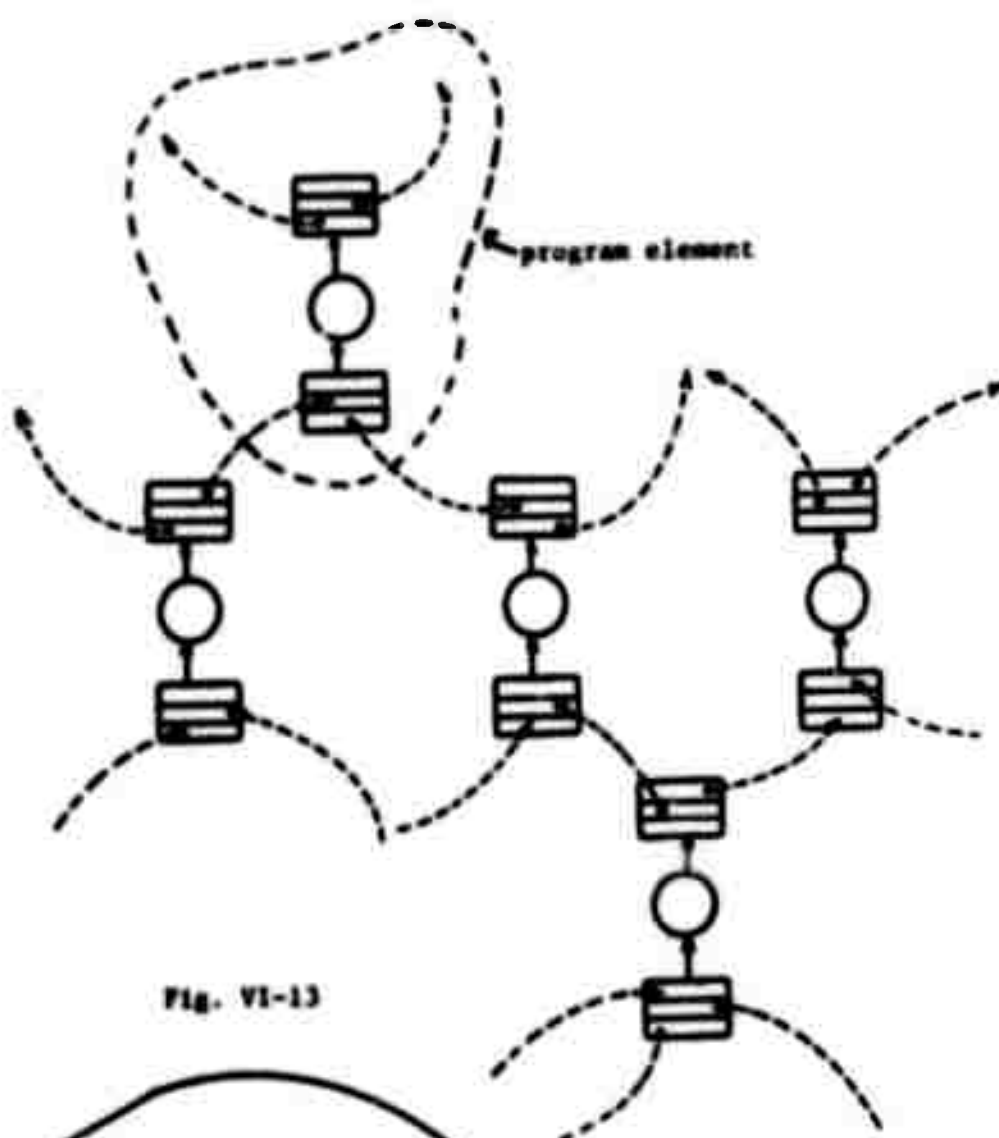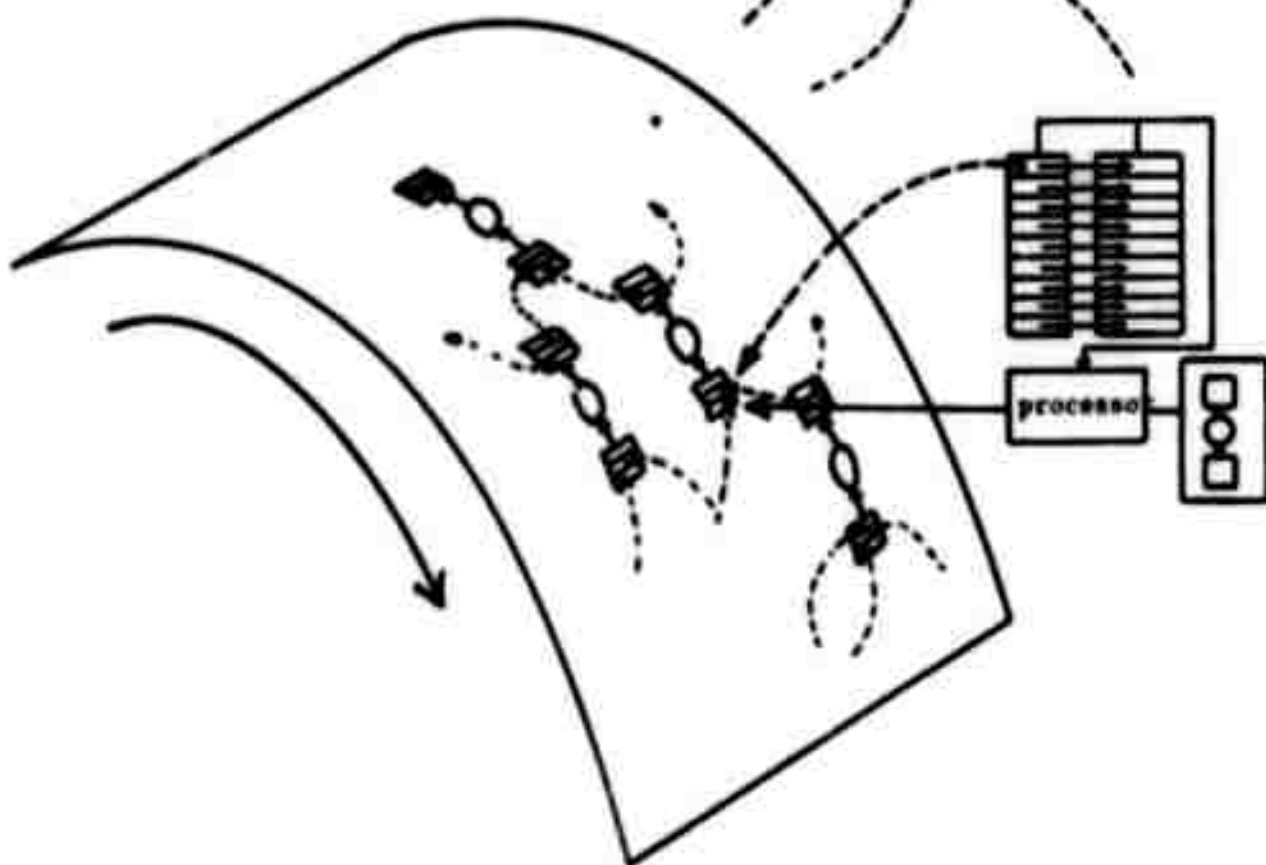
are stored in a library, their addresses appearing only in the
program elements (fig. VI-15). A computation looks at any time
like the network of fig. VI-16. Whenever there is a message for
an entry in an event table which will be filled with this message,
a request for the proper sequential process in the library will
be made; the elementary program can only be performed when this
process will be available. We can already know to which entries
in which event tables we will have messages to send: if these
entries pass under the head while the required program has not
yet been provided, the messages to be sent to these entries are
not yet available. However we may see whether the event table
would be filled if the message were available; if so, we may already
request from the library the sequential process whose address is
contained in this last elementary program. The scheme may be
performed again with this new elementary program. For instance,
in fig. VI-17, one message triggers the request for five sequential
processes from the library.

We have seen that it is possible to single out from the
computation network a subgraph of elementary programs which can
be performed as soon as the corresponding sequential processes
will be available from the library.

Whenever a message is conditional, we cannot know in advance
whether this message will be sent or not to the associated entry
in the corresponding event table. It is then a question of policy
to extend or not the scheme to such event tables.

We can now propose another machine organization with a hierar-
chy of sequential rotative memories (fig. VI-18). For instance,

reference to a sequential
connected process

Fig. VI-15

library

Fig. VI-16

subgraph to be swapped, singled out by the current message

queue of addresses

messages

Fig. VI-17

sequential rotative memories
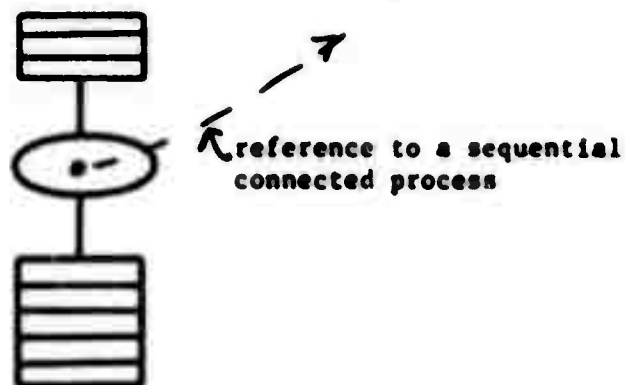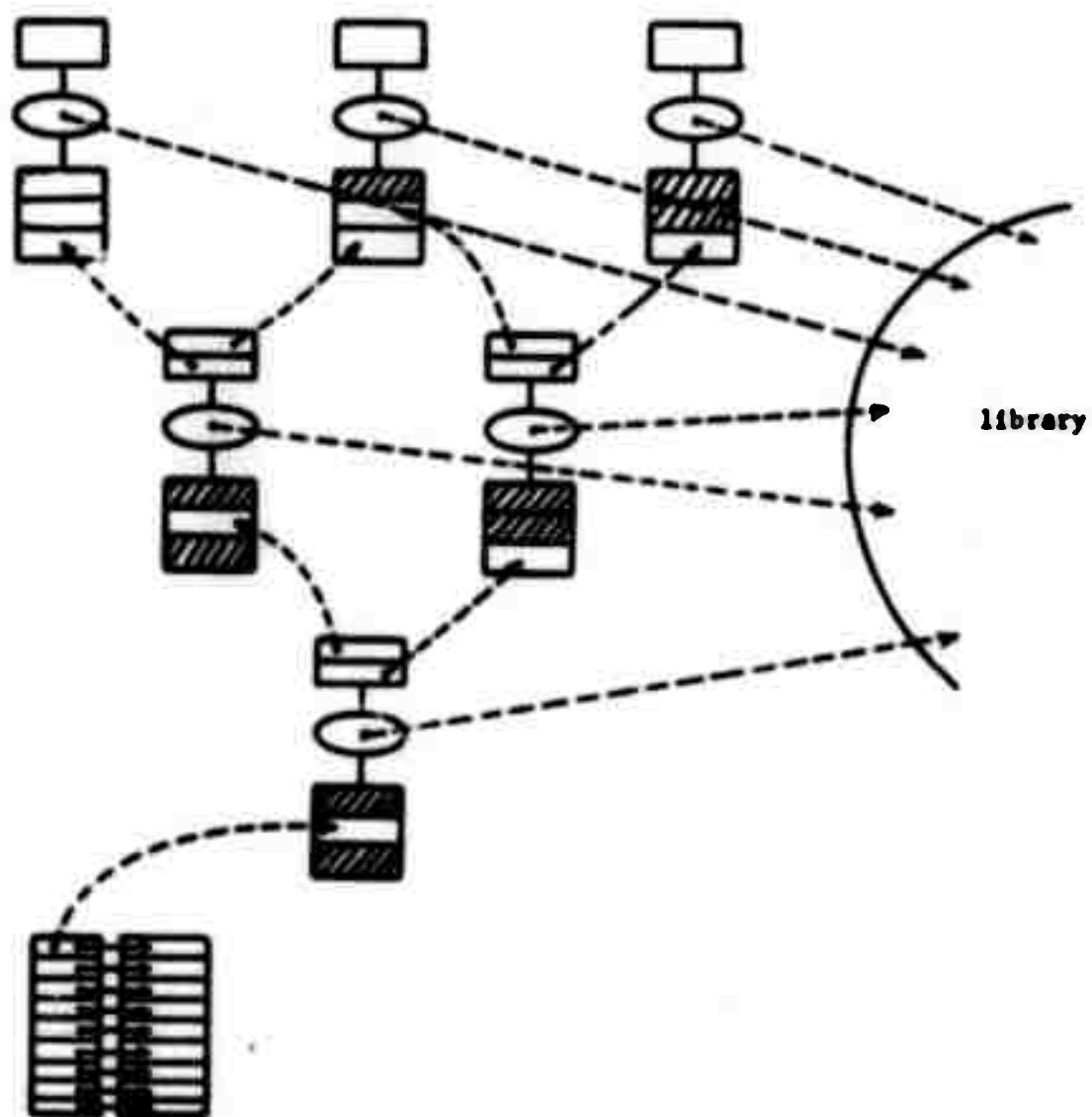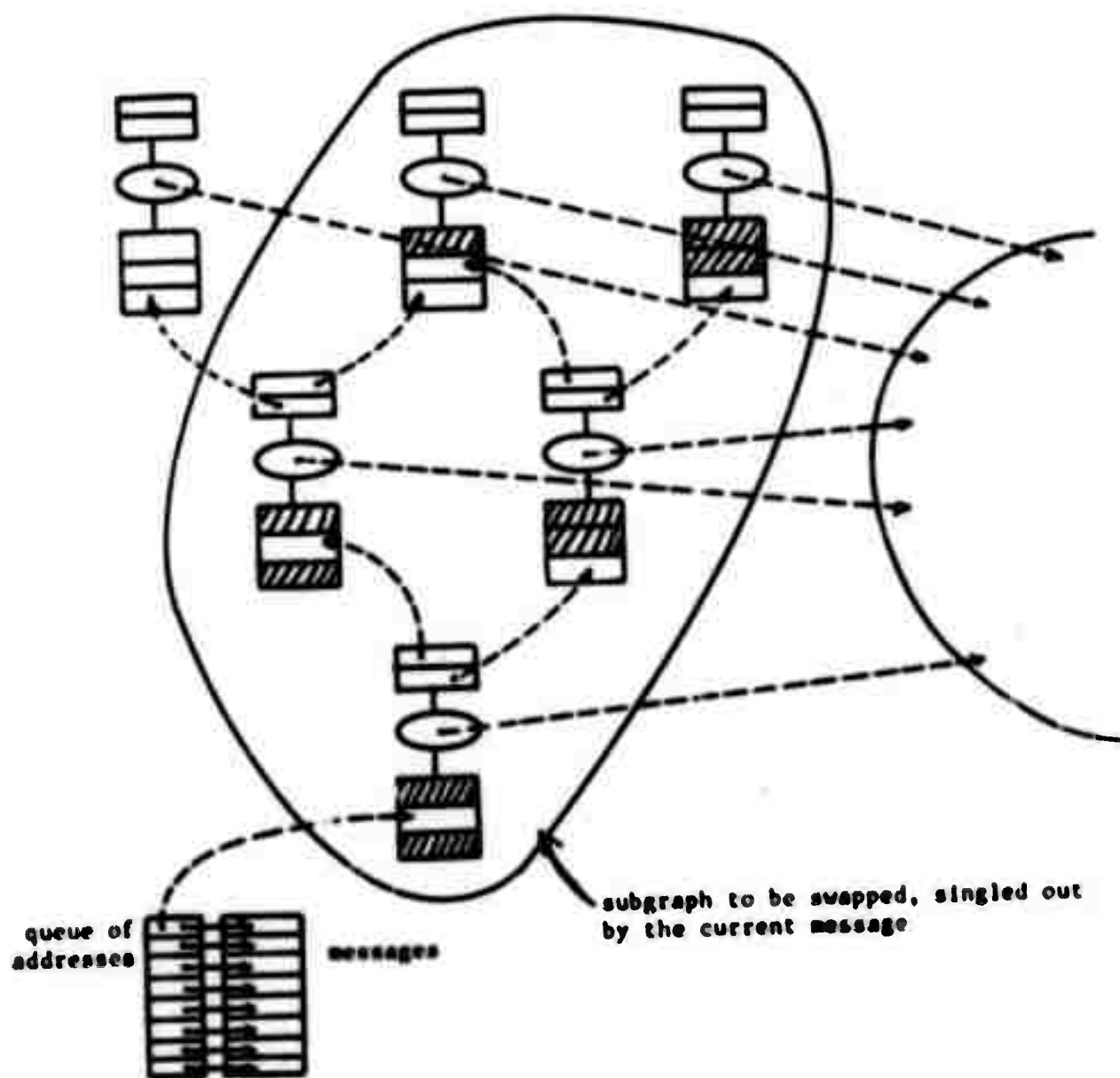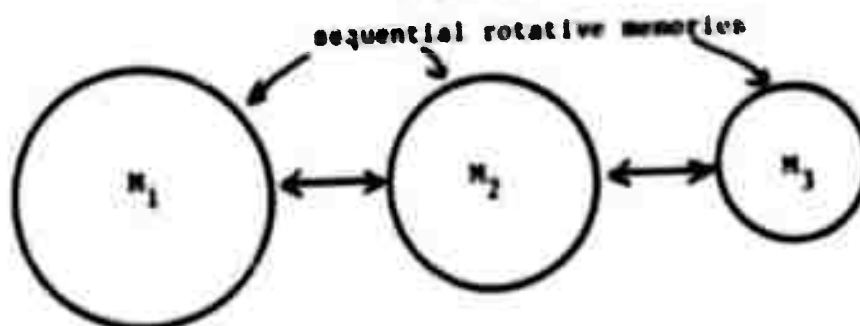
Fig. VI-18

each memory might have the same transfer rate and the capacity might be almost proportional to the rotation time:

	M1	M2	M3
rotation time	16 ms	1 ms	64 μs
capacity	256K	16K	256 words

Subgraphs of computation networks may be swapped between these memory levels using the previous scheme.

A sequential memo· .. a privileged direction: consequently, an implementation of a network will be more efficient if a greater number of messages are to be sent in the pri·ileged direction.

The internal binding of a DCPL procedure may be performed before the procedure is placed in a library. This b'nding super-imposes to the procedure a graph structure which may be expressed as a network of sequential connected program elements. Moreover, this network may be implemented contiguously in a logical space (for instance, a segment). Whenever this procedure is implemented in a computation tree, this segment is stored on the rotative memory in a way which keeps (relatively to the privileged direction) the topology of the network. Then the procedure is bound to its environment, in the computation tree, according to the DCPL binding rules. The convention we have taken previously (a son appears under the head before its father) assures a non-optimal but to a large extent satisfying solution.

A program in DCPL exhibits, to a large extent, the flow of information, the possible concurrency, and the "topology" of the computation structure. This allows new machine organizations which, we believe, would permit to obtain a larger throughput with less resources.

CHAPTER VII

CONCLUSION

Notions which are relevant to both programming language design and machine organization have been discussed in this thesis.

Progresses made in these two domains occurred generally independently; since these developments were to be compatible, artificial restrictions have been imposed in order to define a common frame of reference. We believe such restrictions can only be removed by comprehensive approaches.

Today's programming languages view computers as if they were still simple von Neumann's type machines. A computation is mainly considered as a sequence of instructions, which can modify the contents of some cells. As a result, almost any possibility of having concurrency and distribution of control in computation structures is lost.

Today's machine and system organizations consider programming languages as if they were unable to grasp the information and control topology of the computations which are expressed with them. As a result, possibilities to plan in advance the computation process are lost, producing, in our opinion, less efficient systems.

A DCPL program exhibits concurrency, distribution of control and locality of references. We believe that this will permit to have a more efficient machine organization with less expensive resources:

with a traditional organization, a very fast processor (a "processor on a chip" will be very inexpensive in just a few years) would not be able to be fully used unless a prohibitively expensive very fast memory were to be used, or small iterative computations were expected to occur very often. The machine organization proposed here is believed to be able to "feed" such a processor by using relatively not expensive, very large transfer rate, sequential memories.

LIST OF REFERENCES

1. Adams, Duane A. A computation model with data flow sequencing. (Ph. D. Thesis) Technical report no. CS 117, Computer Science Department, Stanford University, December 1968.

2. Church, A. The calculi of lambda-conversion. _Ann. of Math Studies_ 6. Princeton, N.J.,1941; 2nd edition, 1951.

3. Curry, H. et al. _Combinatory Logic_, volume 1. North-Holland Publ. Company, Amsterdam, 1968.

4. Dennis, J. Programming generality, parallelism and computer architecture. Computation Structures Group, memo no. 32, project MAC, M.I.T., 1968.

5. Dijkstra, E. Cooperating sequential processes. Math. Department, Technological University, Eindhoven, Sept. 1965.

6. Estrin, G. and Turn, R. Automatic assignment of computations in a variable structure computer system. _IEEE Transactions on Elec. Comp._, 12 (Dec. 1963) pp. 755-773.

7. Evans, A. Jr. PAL - A language for teaching programming linguistics. Memo. M-380, M.I.T., Cambridge, Mass.

8. Krutar, R.A. Unpublished notes : "Sequentially cooperating processes", ... , C.S. Dept., Carnegie Mellon Univ., Pittsburgh, Pa.

9. Holt, A.W. Final report for the information system theory project. Rome Air Development Center, ARPA contract no. AF30(602)-4211, Applied Data Research, Princeton, N.J. , Feb. 1968.

10. Karp, R.M. and Miller, R.E. Properties of a model for parallel computations: determinacy, termination, queueing. _SIAM J. Appl. Math._ 14 (Nov. 1966), pp. 1390-1411.

10a. Karp, R.M. and Miller, R.E. Parallel program schemata: a mathematical model for parallel computation. IEEE conference record of 1967 Eigth Annual Symposium on Switching and Automata Theory (Oct. 1967).

11. Knuth, D.E. Semantics of context-free languages. _Mathematical Systems Theory_, Vol.2, no. 2, Springer-Verlag, New York, N.Y.

12. Landin, P.J. The mechanical evaluation of expressions. _Comput. J._ 6,4 (Jan. 1964), pp. 308-320.

13. Landin, P.J. A correspondance between Algol 60 and Church's lambda-notation : Part I , Comm of the ACM 8,2(Feb. 1965) pp. 89-101 , and Part II, Comm. of the ACM 8,3(March 1965) pp. 158-165.

14. Landin, P.J. The next 700 programming languages. Comm. of the ACM 9,3 (March, 1966). pp. 157-164.

15. Luconi, F.L. Asynchronous computational structures. (Ph.D. Thesis) Project MAC, M.I.T., Feb. 1968.

16. Martin, D.F. The automatic assignment and sequencing of computations on parallel processor systems. UCLA Report no. 66-4 (Jan. 1966).

17. Mitchell, B. Theory of Categories.Academic Press, New York and London, 1965.

18. McCarthy, J. et al. LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Mass., 1962.

19. Morris, J.H. Lambda-calculus models of programming languages. (Ph.D. Thesis) Project MAC, M.I.T., Dec. 1968.

20. Patil, S.S. N-server m-user arbiter. Computation Structures Group memo no. 42. Project MAC, M.I.T., May 1969.

21. Rodriguez, J.E. A graph model for parallel computations. (Ph.D. Thesis), M.I.T., Sept. 1967.

22. Shapiro, R.M. The representation of algorithms. Applied Data Research, Inc., Princeton, N.J., Sept. 1969.

23. Van Horn, E.C., Jr. Computer design for asynchronously reproducible multiprocessing. (Ph.D. Thesis) Project MAC, M.I.T., Nov. 1966.

24. von Neumann, J. Theory of automata: construction , reproduction, homogeneity, Part II of "The Theory of Self-Reproducing Automata", ed. A.W. Burks. Univ. of Illinois Press, Urbana, Illinois, 1966.

25. Watson, R.M. Introduction to time-sharing concepts. Shell Development Company, Emeryville, Calif. , Jan. 1969.

26. Wirth, N. and Hoare, C.A.R. A contribution to the development of Algol. Comm. of the ACM 9,6 (June 1966) pp. 413-432.

27. Wirth, N. On certain basic concepts of programming languages. Computer Science Department, Stanford University, Stanford, Cal., May, 1967.

28. Wirth, N. Euler: A generalization of ALGOL, and its formal definition: Part I, Comm. of the ACM 9,1(Jan. 1966) pp. 13-23, and Part II, Comm. of the ACM 9,2(Febr. 1966) pp. 89-99.

VITA

Denis David Séror ███ ████████████████ ██████████. Tunisia

where he attended public primary school. He attended high school in Paris and began his first struggles with mathematics at the Lycée Louis Le Grand. He then was admitted to the Ecole Polytechnique from which he graduated in July 66. He worked two years at the computer center of the Civil Division of the French Atomic Energy Commission, while attending the University of Paris. He finally came to the University of Utah as a graduate student in September 68. Denis D. Seror married Ann Coyle (Mount Holyoke College 69) whom he met on the liner S.S. France, in June 69.