MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Computation Structures Group Memo 156

A Structure Controller for Data Flow Computers

by

William B. Ackerman

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

Jan. 1978

This paper presents an implementation of a structure controller for a data flow computer. The computer is presumed to be similar to that in [Den1], and the controller performs the memory operations described in [Ack1], and makes use of the memory system described therein. The reader is referred to those documents for more details.

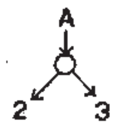
Structures are realized as acyclic binary trees, in which a node is an elementary value or is a pair of pointers to other nodes. The position of any node of a structure is therefore defined by its _selector,_ which is the string of zeros and ones telling which of the pointers to follow at each node when tracing the tree from the root to the given node. The special elementary object _nil_ is used to represent "nonexistent" substructures. It is the only elementary value specifically recognized by the structure controller - all others are treated as uninterpreted data.

There are two machine level operations performed by the structure controller:

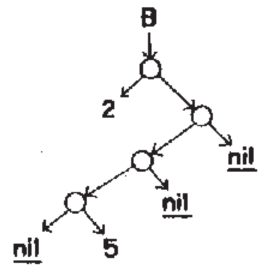
SELECT[structure, selector] --> object
returns the object at the point in the structure indicated by the selector (the selector is a bit string, which is presumably an elementary data type handled by the computer). It is an error if the structure controller attempts to "run past" an elementary value other than _nil,_ that is, if there is such an elementary value in the given structure, whose selector is a proper prefix of the given selector. If the structure controller attempts to run past the value _nil,_ it simply returns _nil_ as the result.

APPEND[structure, selector, object] --> structure
returns a structure which is identical to the given one, except that it contains the given object at the position indicated by the given selector. Whatever object was previously at that position is absent in the result. Any elementary object in the original structure at a position whose selector is a prefix of the given selector must also be absent in the result.

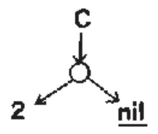
Nodes and branches are added during an APPEND as necessary to create a place for the item to be appended. They are removed as necessary to avoid any substructures whose only terminal nodes are nil.



APPEND[A, '1001', 5] yields



APPEND[B, '1001', nil] then yields



REFERENCE COUNTS, SHARING, AND COPYING

It is the nature of a data flow computer that its semantics must be completely local. The meaning of the result of an operation such as an APPEND must be completely defined by the meaning of its arguments, independently of any other operations that may be performed on the same arguments elsewhere in the computer. There are two (closely related) reasons for this requirement: It would not be possible to define the semantics of data flow languages is a useful and understandable way if this were not the case, and the behavior of a data flow computer would not be determinate if its operations did not meet this requirement. Therefore, the data flow computer must handle structures in an "applicative" manner, as opposed to the "impure" in which they are handled in many common languages with data structure facilities. The meaning of a data structure must be independent of any conceptual "global state" of the memory, even though the structures are stored in a global memory. This requires very careful design of the structure controller.

One solution to the problem would be to forbid any sharing or overlapping of structures. Since every structure would have its own private area of memory, there would be no global effects. However, this would probably be prohibitively inefficient. It would require each structure to be completely copied whenever its value is duplicated. The solution to be used is instead to copy nodes when they are to be written on and there are other pointers to the same node. The determination of whether there are other pointers to a node is made by examining the node's reference count.

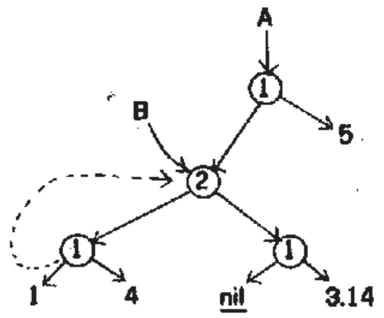
Each node of a structure has a reference count, which is the total number of pointers to that node from all sources - other nodes and "tokens" in the computer. All operations that create or destroy pointers must alter the reference count. For example, when a <u>true</u> or <u>false</u> operator destroys a token, the count must be decreased. When a pointer is written onto a node, the pointer that was previously in the memory location being written is destroyed and hence must have its count decreased. When a SELECT is performed, the count of the result must be increased, and the count of the original structure must be decreased, to account for the tokens that are created and destroyed. (In fact, the increase on the result must happen <u>before</u> the decrease on the argument, to avoid a possible erroneous cell destruction.)

The cell allocation and management algorithm (see [Knu1] section 2.3.5) is as follows: Free cells are kept on a free storage list. When a new cell is needed, one is removed from this list, and its reference count is set to one. Whenever a reference count is decreased and the result is zero, that cell is <u>reclaimed</u>, that is, returned to the free storage list. When this happens, the pointers contained in the cell are destroyed, so the reference counts of the nodes pointed to must be decreased. If either or both of those counts go to zero, those cells are reclaimed, and the process repeats itself.

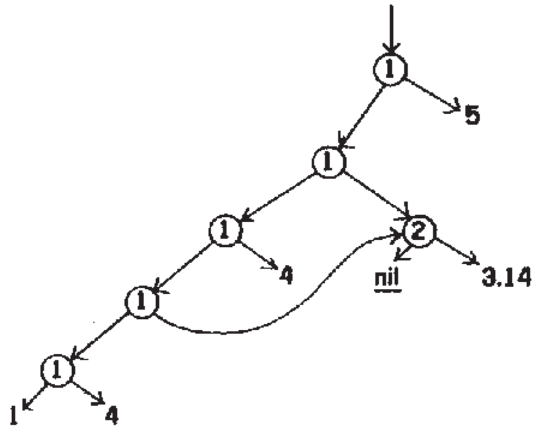
The reference count method of cell management is known to work if no directed cycles ever exist in any structure. (If directed cycles can exist, such a cycle could be abandoned by destroying all pointers to it from the rest of the computation, but it would never be reclaimed because each node would have a reference count of one.) Therefore, to prove correctness of the structure controller, it is necessary to show that no cycles will ever be formed and that the APPEND operation will be performed without side effects. Pure LISP, that is, a system with only CAR, CDR, and CONS operations, satisfies both of these criteria. Cycles can never be formed because a node can only point to nodes older than itself. CONS is free of side effects because it only writes on a freshly created cell.

It will therefore suffice to show that the behavior of the structure controller performing an APPEND is equivalent to the way it would behave if were defined in terms of CAR, CDR, and CONS. A detailed proof of this has not been worked out. However, an argument can be given for the correctness of the decision whether to copy a node. When the structure controller wants to write on a node (say, on the CAR side) while doing an APPEND, it checks that node's reference count. If it is one, meaning that the only pointer to that node is the structure controller's own pointer, it simply writes on the CAR, leaving the CDR unchanged. Otherwise, it gets a fresh node from the free storage list and writes on the CAR of that, filling in the CDR with the CDR of the original node, and then decreases the original node's reference count. The structure controller's rationale for this is as follows: If the only available memory operations were were CAR, CDR, and CONS, it would always use CONS to get a fresh node, specifying that the CAR of the new node is to be whatever it wants to write, and the CDR is to be the CDR taken from the original node. This is exactly what it in fact does when the reference count is not equal to one. Now what happens if the structure controller finds that the reference count of the original node is one? It knows that, when it decreases the count after reading the CDR, the old node will be reclaimed. When it performs a CONS to build the new node, it might get the same node back, and could bypass the reclamation and generation by simply re-using the same node. The CDR part will already have the correct data, so only the CAR needs to be written.

The effect of the structure controller's copying behavior, and its avoidance of circular lists, is illustrated by the following example. The numbers appearing inside nodes are reference counts. If A and B denote pointers into the following structure (ignoring the dotted line):



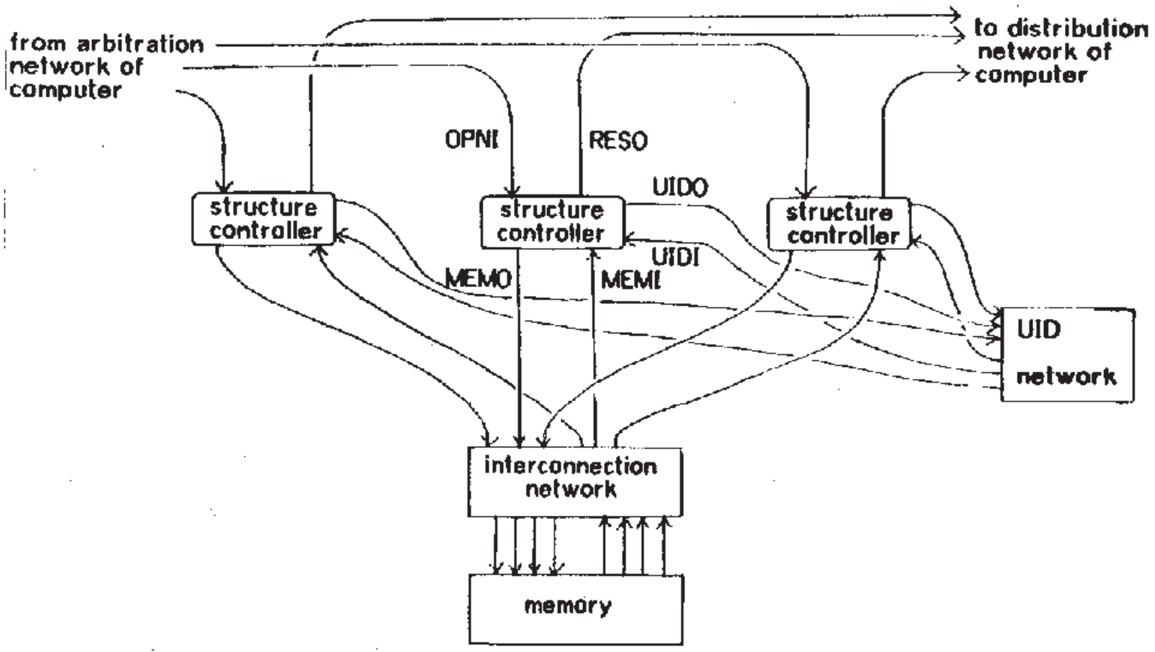
a very naive execution of APPEND[A, '000', B] might replace the pointer to "1" with a pointer back to B, as shown by the dotted line. This is incorrect. The correct result of APPEND[A, '000', B] is:



Because reference counts must be updated whenever structure valued tokens are duplicated or destroyed, and only the structure controller can manipulate reference counts, the structure controller must perform all instructions that duplicate or destroy structure tokens. In particular, the true and false conditional gates (see [Den1]) cause tokens to be conditionally destroyed. These instructions can be handled by very simple functional units if the type of the data token is elementary, but if it is a structure, the instruction must be processed by the structure controller. The processing of these instructions is completely straightforward and is not shown in this paper.

THE CONTROLLER'S ENVIRONMENT

The structure controller is to be one of several identical units connected to the computer, the memory through the interconnection network, and each other through the UID network, as shown below.



The functions of the ports are as follows:

OPNI - receives operation packets from the computer. These packets are either SELECT(struct, sel, dest_1, ... dest_N) or APPEND(struct, new-val, sel, dest_1, ... dest_N). The computer sends each operation packet to any structure controller that is able to receive it, that is, any controller that has acknowledged the previous one. (This is the same as what the computer does with arithmetic operation packets, sending them to whichever arithmetic processor is able to receive them.)

RESO - transmits results of structure operations back to the computer, one packet for each destination that was given in the operation packet. These packets are (value, dest).

MEMO - transmits commands through the interconnection network to the memory. These packets are $\text{FET}^{(\pm)}$(addr, tag) or UPD(addr, data, ref). The memory is implemented as several modules, each handling certain addresses. The interconnection network sorts these packets according to address, sending each to the right memory unit. $\text{FET}^{(\pm)}$ means FET, FET^{+}, or FET^{-}.

MEMI - receives replies from $\text{FET}^{(\pm)}$ commands. These packets are $\text{RTR}^{(\pm)}$(ADDR, DATA, REF, TAG). The interconnection network routes these packets from the memory back to whichever structure controller sent the command.

UIDI - receives UID packets, giving fresh cells to be used for the creation of new nodes. These packets are UID(addr, object). "Object" is some value that, if it is a structure, must have its reference count reduced. This will be explained later.

UIDO - transmits UID packets. The UID network simply takes UID packets according to the various structure controllers' abilities and redistributes them according to the controllers' needs.

THE UID NETWORK AND FREE STORAGE

There should always be a packet available at UIDI. When it needs a fresh cell, the controller simply takes that packet and acknowledges it. Upon receiving the acknowledge, the UID network provides another packet. The structure controller must constantly provide fresh cells to the network through port UIDO. Whenever the UID network acknowledges the last packet and the controller's free storage list is not empty, the controller takes a cell from the list and transmits it at UIDO.

Whenever the structure controller reduces the reference count of a cell to zero, it reclaims that cell by placing it on the free storage list. It must also decrease the reference counts of the cells pointed to by the pointers in the reclaimed cell. This may cause one or both of those cells to be reclaimed, so the procedure is recursive. Since each reclamation can cause two others, the recursion is difficult to handle. (The structure controller has no stack memory.) The procedure used is only to reduce the reference count of the node pointed to by the right half of the word at the time it is reclaimed. While the word is on the free storage list, its left half is preserved, and its right half is a pointer to the next cell on the list. This makes reclamation of the cell give rise to only one other reclamation instead of two, so the recursion can be handled iteratively without a stack.

When the structure controller receives a cell at UIDI, the left half still contains a pointer to a node that must have its reference count reduced. The structure controller reduces that count before using the cell. It could get this pointer by reading it from the new cell, but, since the cell was read when it was removed from the free storage list, an extra memory reference can be saved by passing that pointer along with the cell's address. For this reason, the UID packet contains both a cell and an object (pointer) whose reference count must be reduced.

If no packet is available at UIDI when one is needed, the computer has presumably run out of memory. This is an unpleasant situation to deal with, since its occurrence is nondeterminate (one run of a program might succeed while another run fails).

The simplest thing to do is to terminate the computation. There is a chance that, by simply waiting, a free cell might be created by another part of the program, allowing the computation to proceed. However, this strategy gives no positive indication that a computation has failed other than the fact that it stops executing instructions, which may be undesirable. In the implementation to be given, it is assumed that packets are always present at UIDI.

INITIALIZATION OF THE FREE STORAGE LISTS

Before starting program execution in a data flow computer, all of memory must be put into the free storage lists. The algorithm to be given assumes that the free storage list in each structure controller is a chain of words whose head is pointed to by a register called FREE. The right half of the data in each word contains the address of the next word in the list. The right half of the last word contains an elementary object, such as nil. The left half of each word contains a structure whose reference count is to be reduced when that word is taken from the list. It is assumed that these lists are initialized by dividing up the total memory space into as many parts as there are controllers, and linking all words in each part into a chain. The left half of each word is initially filled with some harmless elementary object such as nil, and the head of each chain is put into the FREE register of its structure controller. As the final initialization step, each controller must then behave as though it had received an acknowledge on port UIDO, that is, it must take a word from the list and send it out through UIDO.

GENERAL DESIGN OF THE CONTROLLER

Each controller can handle some fixed number of concurrent operations. Each of these operations requires a number of private variables that must be stored in the controller's local memory. All of the variables in the flowcharts given later except LOCK and FREE are private. The private variables are actually arrays whose size is the number of concurrent structure operations that the controller can handle. Each operation is given a "tag", or index, which is used to index the arrays and to identify memory transactions.

The total number of controllers, and the number of concurrent operations that each controller is designed to handle, is a complex decision based on the speed of the memory system, the speed of the controllers' logic and internal memory, the speed of the switching networks and communication ports, and the required performance of the system.

When a structure operation is received at OPNI, an unused tag is selected for it. (If there are no unused tags, the controller does not accept packets at OPNI.) The operation then proceeds concurrently with all the others, with all of its memory transactions labelled with its tag, and all of its state information stored in the various arrays at an index equal to its tag. Each RTR$^{(\pm)}$ packet returned by the memory has a tag field (not shown) that is equal to the tag sent in the corresponding FET$^{(\pm)}$ packet, so that the controller knows which operation the RTR$^{(\pm)}$ refers to. When the structure operation is complete, the result packets are sent out and its tag becomes free. The operations are processed according to the flowcharts. For simplicity, tag fields in packets are not shown, and references to the state arrays are shown as simple variables. All memory transactions are shown in boxes and, except for UPD, consist of transmission of a command followed by a wait for a reply. UPD transactions consist of a transmission only. Some overlapping of memory transactions within one structure operation would be possible, but it is not shown since the state transitions are simpler when viewed as a uniprocess flowchart. (In particular, the "discard" function can proceed concurrently with the rest of the operation.) FREE and LOCK are global variables. All others are arrays indexed by an operation's tag. FREE points to the head of the free storage list. LOCK is initially zero, and is set to one whenever a memory transaction is in progress which manipulates the free storage list.

The structure controller never issues an UPD command unless the reference count is known to be one. Since this is so, there can be no transactions pending on that cell, so the requirements for correct memory use (see [Ack1] section 3) are met. This is contingent, of course, on the rest of the computer correctly realizing its specification. Any incorrect handling of a reference count by the computer (for example, if it duplicated a structure valued token without increasing its reference count) could lead to an UPD packet being sent while there are transactions pending.

THE "SELECT" AND "APPEND" ALGORITHMS

The SELECT operation is straightforward. The string S is set to the selector. A is set to the incoming structure, so that it can be discarded (have its reference count reduced) at the end of the operation. R is a working pointer that traces the structure. The controller repeatedly reads the word at R, picking out the left or right half of the data, depending on the next bit of S, and putting the result back in R. When S runs out, R is the result. Its reference count is increased the appropriate number of times, and then A is discarded.

The APPEND operation is much more complex. It is designed to require only a single pass down the structure, creating new cells only when the reference count forbids writing on existing cells. A is a working pointer that traces down the structure. R is the result, which is set on the first iteration either to A or to the cell that A was copied into. H is true during the first iteration and false later. G and GG are flags that become false as soon as the operation passes a node whose reference count is greater than one. When this happens, all subsequent nodes must be copied.

A special case arises if the value to be appended (the variable C) is nil. The structure controller must take care to remove any structure that would contain nothing but nil as its terminal nodes. It uses the variables W and P for this. W can take the values true, false, 0, or 1. When it is 0 or 1, the entire subtree from P down to the current place in the structure is a chain of nil's. If this situation persists when the controller has reached the end, that chain is discarded by writing on P. The value of W (0 or 1) tells which half to write on. If W is true, the chain of nil's extends all the way to the top, so the entire result is to be discarded and replaced with the elementary object nil. If W is false, there are no chains of nil's to be discarded.

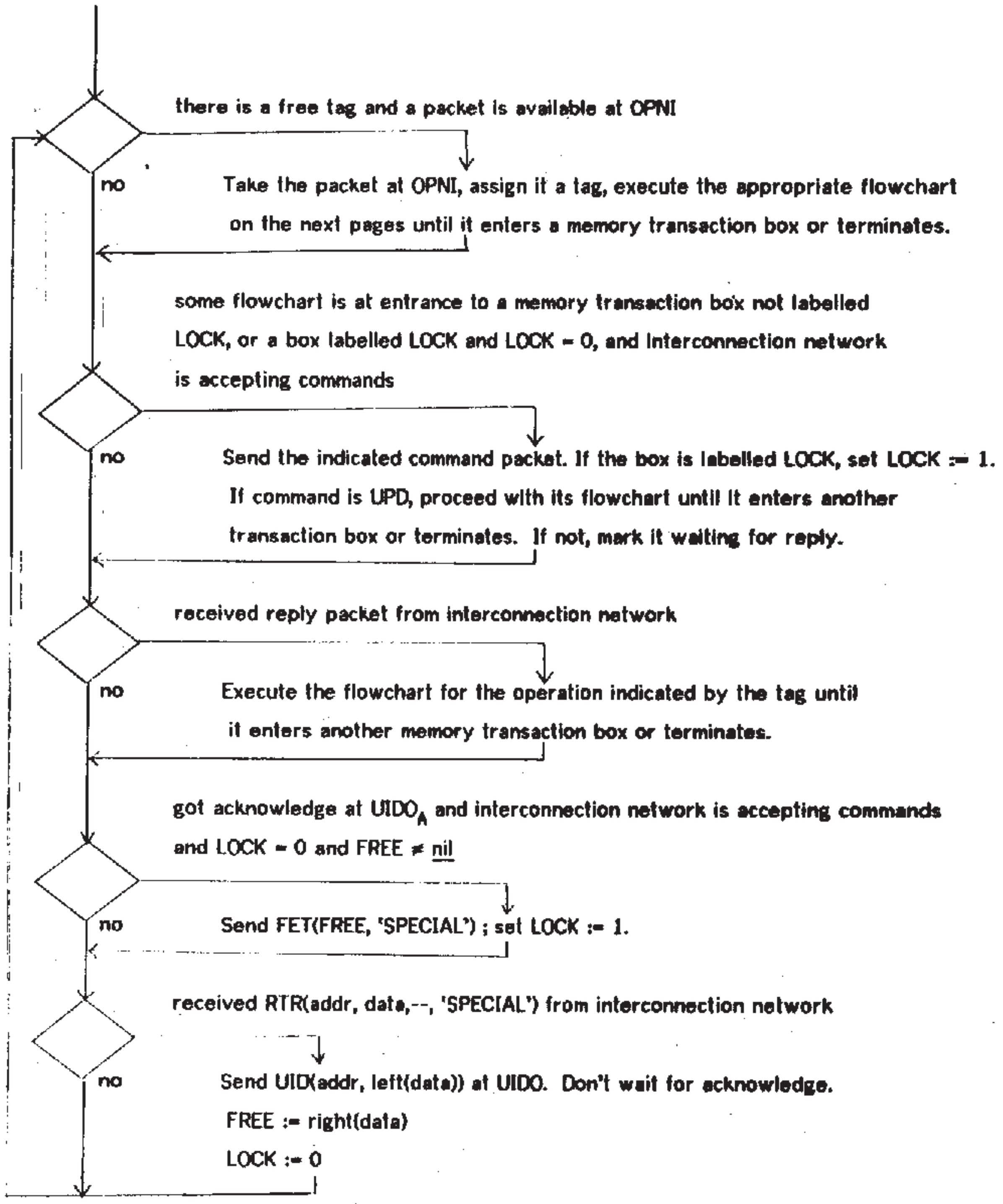
The structure controller algorithm is exhibited in the following flowcharts. First the overall sequence is given, and then the manipulations that are private to each structure operation.

REFERENCES

[Ack1] Ackerman, W. B. "A Structure Memory for Data Flow Computers", Laboratory for Computer Science (TR-186), MIT, Cambridge, Massachusetts, August 1977.

[Den1] Dennis, J. B., D. P. Misunas, and C. K. C. Leung "A Highly Parallel Processor Using a Data Flow Machine Language", Computation Structures Group (Memo 134), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1977.

[Knu1] Knuth, D. E. "The Art of Computer Programming, Vol. 1: Fundamental Algorithms", Addison-Wesley, Reading, Mass., 1968.



there is a free tag and a packet is available at OPNI

no

Take the packet at OPNI, assign it a tag, execute the appropriate flowchart on the next pages until it enters a memory transaction box or terminates.

some flowchart is at entrance to a memory transaction box not labelled LOCK, or a box labelled LOCK and LOCK = 0, and interconnection network is accepting commands

no

Send the indicated command packet. If the box is labelled LOCK, set LOCK := 1. If command is UPD, proceed with its flowchart until it enters another transaction box or terminates. If not, mark it waiting for reply.

received reply packet from interconnection network

no

Execute the flowchart for the operation indicated by the tag until it enters another memory transaction box or terminates.

got acknowledge at $UIDO_A$ and interconnection network is accepting commands and LOCK = 0 and FREE ≠ nil

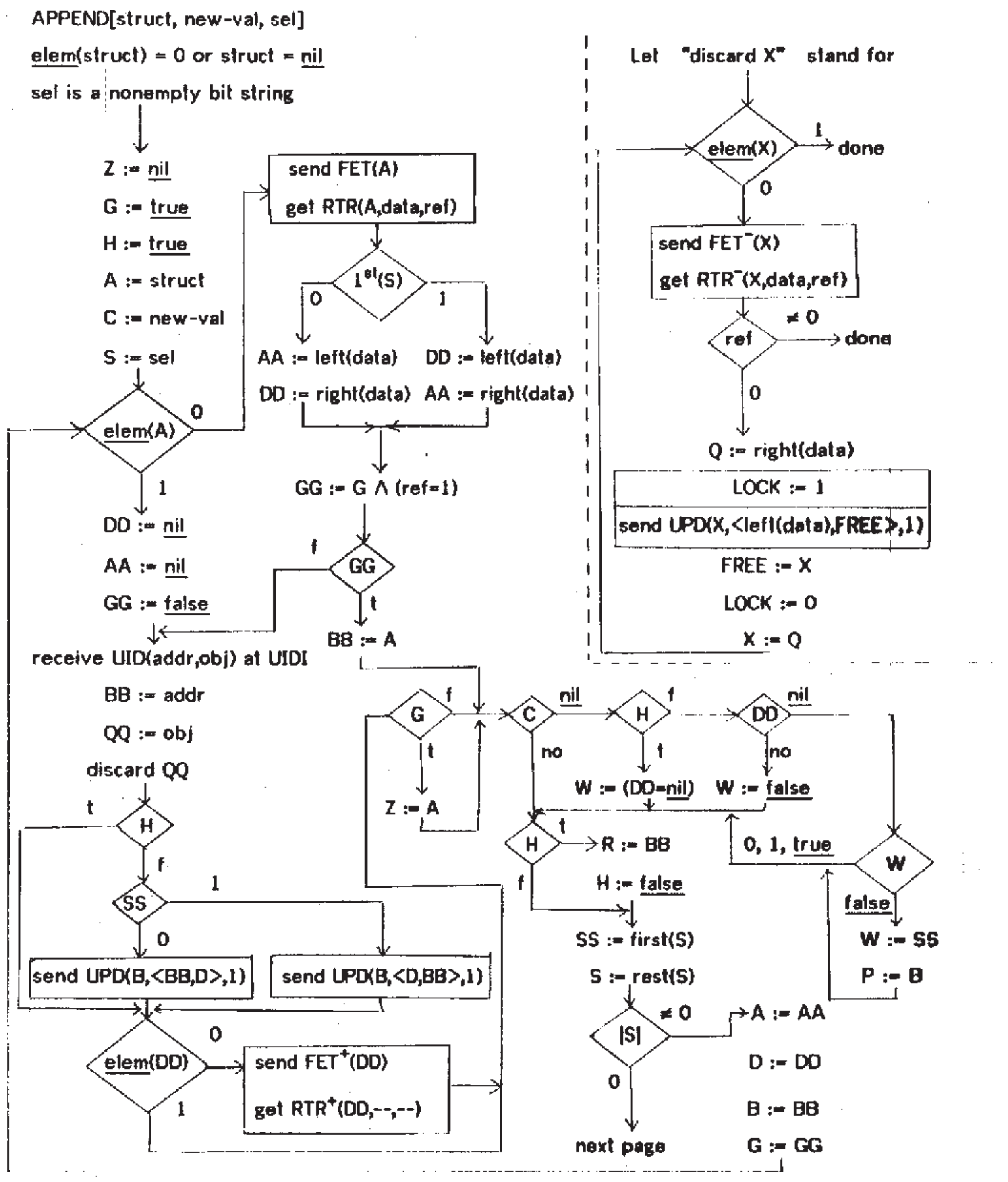
no

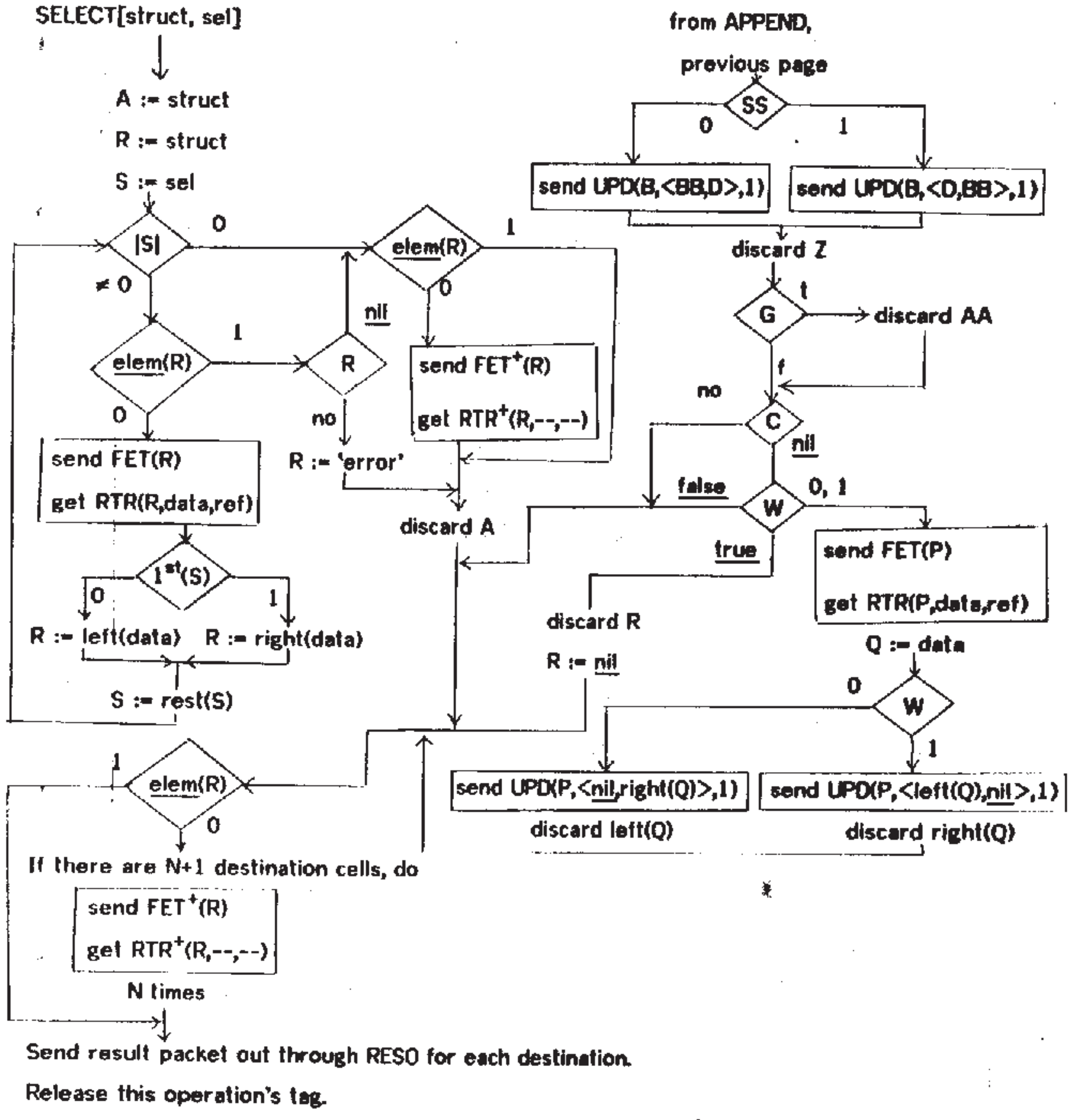
Send FET(FREE, 'SPECIAL') ; set LOCK := 1.

received RTR(addr, data,--, 'SPECIAL') from interconnection network

no

Send UID(addr, left(data)) at UIDO. Don't wait for acknowledge.
FREE := right(data)
LOCK := 0


APPEND[struct, new-val, sel]
elem(struct) = 0 or struct = nil
sel is a nonempty bit string
Z := nil
G := true
H := true
A := struct
C := new-val
S := sel
elem(A)
0
1
DD := nil
AA := nil
GG := false
receive UID(addr,obj) at UIDI
BB := addr
QQ := obj
discard QQ
H
t
f
SS
0
1
send UPD(B,<BB,D>,1)
send UPD(B,<D,BB>,1)
elem(DD)
0
1
send FET⁺(DD)
get RTR⁺(DD,--,--)
send FET(A)
get RTR(A,data,ref)
1st(S)
0
1
AA := left(data)
DD := right(data)
DD := left(data)
AA := right(data)
GG := G ∧ (ref=1)
GG
f
t
BB := A
G
f
t
Z := A
C
nil
no
H
f
t
W := (DD=nil)
DD
nil
no
W := false
H
t
f
R := BB
H := false
SS := first(S)
S := rest(S)
|S|
≠ 0
0
next page
A := AA
D := DD
B := BB
G := GG
W
0, 1, true
false
W := SS
P := B
Let "discard X" stand for
elem(X)
1
done
0
send FET⁻(X)
get RTR⁻(X,data,ref)
ref
≠ 0
done
0
Q := right(data)
LOCK := 1
send UPD(X,<left(data),FREE>,1)
FREE := X
LOCK := 0
X := Q


SELECT[struct, sel]
A := struct
R := struct
S := sel
|S|
0
≠ 0
elem(R)
1
0
R
nil
no
R := 'error'
send FET(R)
get RTR(R,data,ref)
1st(S)
0
1
R := left(data)
R := right(data)
S := rest(S)
elem(R)
1
0
send FET+(R)
get RTR+(R,--,--)
discard A
elem(R)
1
0
If there are N+1 destination cells, do
send FET+(R)
get RTR+(R,--,--)
N times
Send result packet out through RESO for each destination.
Release this operation's tag.
from APPEND,
previous page
SS
0
1
send UPD(B,<BB,D>,1)
send UPD(B,<D,BB>,1)
discard Z
G
t
discard AA
f
no
C
nil
false
W
0, 1
true
send FET(P)
get RTR(P,data,ref)
discard R
R := nil
Q := data
W
0
1
send UPD(P,<nil,right(Q)>,1)
discard left(Q)
send UPD(P,<left(Q),nil>,1)
discard right(Q)