KNOTTED LIST STRUCTURES

Computer Organization Unit
General Electric Computer Laboratory
Sunnyvale, California
January, 1962

By

J. Weizenbaum
R. Shepardson
B. Hellerstein
D. Masters

TABLE OF CONTENTS

INTRODUCTION

This paper describes a list processing language called the KNOTTED LIST STRUCTURE system (KLS). In this day of the prolification of computer languages it is in order to say a few words relating to the motivation underlying the design and construction of yet another language. Most importantly, the design of KLS was undertaken not so much in an attempt to create a list language superior to existing languages, but rather as an effort to gain insight into the hardware software relationships list languages in general might bring to light. The fundamental question which ought ultimatly to be answered by efforts of this nature relates therefore to efforts to design computer organizations which may be very effective vehicles for list processing. At the moment, of course, all list processing systems exist, as does the KLS system, merely as compiled macro systems or as interpretive systems, in other words, as programming systems imposed on otherwise "orthodox" computers. The fact that there exist a class of problems which are currently being solved only by list processing systems of one form or another, even though these systems impose serious penalties in computer running time, certainly suggests that one day a computer may be designed which will be particularly effective for that class of problems or for subproblems of "ordinary" problems which fall into that class. The basic motivation mentioned seems therefore to be realistic.

What distinguishes list processing from "ordinary" processing? The main distinction lies in a redefinition of the successor relationship which is fundamental to the random access memory machines known today. In such machines a datum found in a particular memory cell, say cell alpha, has its successor stored in the "next" cell, namely alpha + one. Many of the operational characteristics of ordinary computers are derived from this single fact. Furthermore, a vast inventory of programming techniques is built on it. In particular a premium is placed on the orderly placement of data within the core memory. A consequence of this is that data which in its natural environment has a complex structure (as opposed to, say, matrices which have a very orderly natural structure) must nevertheless be fitted into memory in a very "regular" way. The natural complexity of the data structure must therefore be reflected in the programs which manipulate these data. This gives rise to programs which are very complicated not so much because of what they must "do to" data, but because of the "decoding" they must do on the structural properties of the data themselves. List processing introduces a different successor relationship. In list processing each symbol is accompanied

by an ancillary datum which serves as a pointer to its successor. This pointer is called a LINK. In addition, each symbol carries with it an identifying tag (called the SUFFIX) which, among other things, makes it possible for the symbol itself to be identified as a pointer to some other datum. List processing systems have been proposed in which each symbol carries with it a number of links, thus making possible elaborate cross referencing. The KLS system provides for only one link with each symbol. It is readily apparent that a link is simply an address of a cell in memory.

An objection frequently voiced by people when they are first exposed to a list processing system is that links waste memory space. The issue is really whether or not links are economic carriers of useful information. When data to be stored is naturally highly sequential (e.g. ordinary matrices with few null elements), links can contribute very little. But, where the important point about data is that they are interrelated in a complex way and, often even more important, that processing causes them to be reorganized dynamically, particularly where reorganization itself constitutes part of the solution of the problem, there links contain information which would otherwise have to be contained in program. Experience with list processing programs reveals this point to be of vital importance.

A LIST is a sequence of list cells linked to one another by link addresses. Every list has one member which has the property that the link address of no other cell in the system is pointing to it. This list cell is called the TOP of the list. Every list also has a terminal member, i.e. a list cell whose link address is not pointing to any other member of the list. This cell is called the LIST TRAILER and has a distinct format reflecting its special function. The list cell whose link address is pointing to the trailer is called the BOTTOM of the list. The address of the trailer is the NAME of the list.

Names of lists are single symbols and are handled as such. This means that data structures (of arbitrary lengths and complexities) are capable of being treated as single symbols when appropriate.

When a list is transferred from core to bulk storage (e.g. drum) the trailer stays in core. It is marked such that its list can be retrieved from bulk storage whenever required. Although the body of the list is returned to an entirely new set of disjoint registers, the name of the list does not change.

A SIMPLE LIST consists of a sequence of list cells none of the symbols of which are names of lists and one of which is a trailer (Figure 2). An empty list is a simple list consisting of only a trailer (Figure 1). Any list may be a sublist of

another list, i.e. its name may appear (with suitable suffix) on a list. Sublists may in turn have sublists and so on. A complex consisting of a list together with all its sublists, the sublists of its sublists, etc., is called a LIST STRUCTURE (Figure 3).

In the KLS system a list cell consists of three parts: the ID suffix, the function of which is to identify the function of the cell; the link portion L, which serves to determine the successor relationship of the cell to another; and the symbol portion S which stores symbols. In addition, there are three other types of cells with distinct formats. These are:

1. Two types of trailers
   a) Trailers for lists which have description lists
   b) Trailers for lists which do not have description lists
2. Readers

The following table identifies each type of cell which may occur in the system:

Suffix Table

TABLE I

Suffix	Mnemonic	Type
0	R	Reader
1	DLIST	Description Trailer
2	LIST	Non-description trailer
3	AN	Alphanumeric
4	--	Machine detectable alphanumeric delimeter
5	DI	Decimal number
6	BI	Binary fixed point number
7	BFL	Floating point binary number
8	RO	Occurrence of a list name
9	RM	Mention of a list name
10	A	Address of list cell
11	--	Command
12	--	Address of reader

A trailer consists of four parts: the suffix P, the address of the top of its list (i.e. the list of which it is the trailer), the address of the bottom of its list, and a reference counter. The function of the last will be defined below. The address fields of the trailer of an empty list both contain the address of the trailer itself. The suffix of a trailer may have one of two values. One of these indicates that the list named by the trailer is non-empty and that the S portion of the top cell of the list is the name of a special list called the DESCRIPTION SUBLIST of the subject list. The other value of P signifies that the top cell of the list is an ordinary list cell, possibly the trailer itself.

There exists with the system a special kind of list called a DESCRIPTION LIST. What distinguishes this list from other lists is that it is thought of as having a special format. In fact, any list may be <u>treated</u> as a description list, i.e. may be subjected to operations which assume that list to have the description list format. The TOP cell of a non-empty descrition list is called an ATTRIBUTE, the next cell the VALUE of that attribute, the next cell another attribute, and so on. There are operations which, for example, will search such a list for an attribute and produce the value of that attribute as an output. That value may, of course, be the name of a list structure, a number, or any other kind of datum.

Certain lists are said to be DESCRIBED, i.e. to have a description list associated with them. A described list is one the TOP cell of which contains the name of a list which can be retrieved on the basis of commands which address themselves to certain description list processes (e.g. IDL: Input the name of the description list of the list referenced by the address portion of this command). Any reference to the TOP cell of such a list in non-description list context is a reference to the cell linked to that containing the name of the description list. The programmer need exercise no special precautions in this regard because the trailers of described lists are so marked that the system automatically handles any special problems which may arise in this connection.

Lists and list structures exist as entities in memory. Their content becomes available to other parts of the machine by a mechanism call list reading. There exists a piece of machinery called a READER. A reader is a list cell which contains the following pieces of information:

1. It suffix identifying it as a reader
2. An address called the CURRENT POINTER (CP)
3. An address field containing the address of the trailer for which this cell is currently serving as a reader (N)
4. An address field which, when not empty, contains the name of a list called the control list (CL) of that reader.

A reader is said to be empty or cleared when its CP, N and CL fields are zero. It is in its initial state when its CP field is the same as its N field and neither are empty. There is no restriction on the number of readers in which the name of a given list may appear.

There exist operations whose functions are to advance the CP within any given reader and to cause a certain datum to be deposited in working registers accessable to the programmer. These READ operations make possible the traversal of list structures. The general philosophy underlying reading is that the CP is to be advanced such that it points to one after the other of the list cells of a list structure. When the name of a sublist is encountered as an occurrence, that sublist should either be turned down on, or the main list should be pursued at the programmer's option. In any event, after one of these operations has been partially executed, it remains to be determined what datum is to be deposited in a working register. In general, this datum will consist either of the information pointed to by CP after the advance, or some information pointed to by that information, however indirectly. The content of a CP advance command is therefore:

1. An operation code signifying the advance CP operation per se, i.e. distinguishing it from other operations.
2. A "mode" indicator which can take either the value "linear" or "structure". The first of these indicates that any sublist encountered is not to be turned down on as a consequence of this operation, the the second that any sublist so encountered is to be turned down on.
3. A "target" indicator which determines on what kind of symbol the CP is to stop. The CP continues to advance until either a symbol of the indicated type is found (within the constraints of the indicated mode) or the end of the list (in the linear mode) or the end of the list strucutre (in the structure mode) is encountered. The types of targets which can be specified are:
    a) "Word", the next symbol encountered is the target, whatever its type (other than trailer).
    b) "Element", the next symbol which is not an occurrence of the name of a sublist (nor of a trailer) is the target.
    c) "Name", the next symbol which is an occurrence of the name of a sublist is the target.
    d) "Mention", the next symbol which is an occurrence of the mention of a list is the target.
    e) "Occurrence", the next symbol which is an occurrence of the occurrence of a sublist name is the target.
    f) "Element or Mention", the next symbol which is the occurrence of either a mention of a list name or an element is the target.

The distinctions among "NAME", "OCCURRENCE", and "MENTION" are the following: The NAME of a list, as has already been pointed out, is the address of its trailer. NAME is therefore a general term which serves as a token for the whole list (or list structure) named. If a list L-0 is to be a sublist of another list L-1, then the name of the list L-0 will appear on the list L-1 in the form of a symbol encoded (i.e. with an identifying suffix) as an OCCURRENCE. If, on the other hand, the list L-0 is simply a single datum (no matter how lenthy or complex) on, but not part of the structure of, the list L-1, then its name will appear on the list L-1 in the form of a symbol encoded as a MENTION. The operational significance of this will become clear when the READ instructions are described.

With the machinery so far described, it is possible to traverse a list up to the first encounter of a sublist, traverse that sublist until a sublist of it is encountered, turn down it, and so on until finally a trailer is reached. The mechanism for going back up the list structure is the control list. The control list (CL) whose name appears in the reader is a push down list. Whenever a sublist is turned down on, the address of the cell of the higher level list which led to the sublist is pushed down on CL. When the trailer of a sublist is reached, CL is popped up and the contents of the popped up cell placed in CP. The required advance is then carried on from that point. Hence traversing list structures is accomplished with a minimum of attention from the programmer. The control list is, of course, outside the realm of accessibility of a programmer.

There must be essentially three ways of erasing a list structure; one is to erase or clear a reader of a list structure, another to delete the name of a sublist from a list, and finally to overwrite the name of a list structure. In any case, the intention is to erase the entire list structure to which the reference points, i.e. the list named as well as all sublist structure to which that list leads. (The special case where only the list starting at CP is intended for erasure is not discussed separately here.) The available space list has a TOP the address of which appears in a register called, say MT. A list is effectively erased, i.e. all its cells restored to the available space list, when the present contents of the MT register are made the link portion of the trailer and the address of the top cell of the erased list is placed in MT. Two difficulties are that (1) the name of the list to be erased may appear within the system in other contexts, e.g. in another reader, and (2) the erased list may have sublist structures itself, i.e. some of its symbols may be names of sublists which may need erasure. The second of these

difficulties is taken care of when list cells are fetched from available space. It is at that time that an examination of the previous symbol status is made. If the list cell about to be used last contained the occurrence of a name of a sublist, then that sublist is erased.

The difficulty exemplified by the circumstance that a list may have several readers is remedied by the reference counter in the list trailer. This counter counts the number of references currently being made to that list, i.e. how many readers it has plus how many times its name appears anywhere within the system. When a list is to be erased, the reference counter is decreased by one. Only if the resulting count is zero is the list restored to available space as described.[1] Clearly, the time required to erase a list is independent of the length of the list.

In addition to the list, there is another device for storing information within the system. This is the STORAGE CELL, a term borrowed from IPL-V.[3] The term STACK is used equivalantly. Any cell in memory may be a storage cell. The format of such a cell is exactly that of a list cell. The essential difference between a storage cell and a list is that a list has a trailer the address of which serves as the name of the list, whereas the storage cell is a list without a trailer, a list the name of which is the address of its top cell. This distinction has certain important consequences:

1. No reader may be appointed for a storage cell. Access to information on a storage cell is gained only via its top cell. This means that if the datum next to that contained in the top cell is to be retrieved, the top cell must first be vacated and the desired information placed in it. The operation which causes this sequence of events to take place is called "RSS", or RESTORE STACK.
2. The only way a datum may be placed on a storage cell is for that datum to overwrite whatever information is already contained in the top cell of the stack. Since, in many cases, it is desirable to preserve the datum in the top cell even though some new datum needs to be on the stack, an operation called "PRS", or PRESERVE STACK, is provided which causes the symbol at the top of the storage cell to be copied into a cell which is placed just below the top (i.e. to which the top cell is then linked).

One important application of the storage cell concept is the accumulator of the KLS system. The command format and philosophy of the system is essentially that of a single address computer. This implies that, in general, every command contains an operation code which as usual, specifies what the command is to do to some data, and an address portion which designates the operand on which the specified operation is to be performed. Many operations are, however, binary in the sense that they require two operands (but usually yield only a single result). One of the two operands may be pointed to (however indirectly) by the address portion of the command.

The other operand is generally understood to be in the accumulator. In certain cases the accumulator contains an address or the name of a list which, by a chain of indirection the length of which is not explicitly given, leads to the desired operand. In the KLS system this accumulator is called "WO" and is a storage cell. The operations "PRS" and "RSS" therefore apply to the accumulator. One implication of this fact is that, under suitable circumstances, intermediate results need not be stored in especially selected storage cells but may be merely preserved in WO for later recall by way of the RSS operation.

Commands themselves are symbols in list cells. A program is therefore a list or list structure. The functions usually assigned to the command counter are carried out by a reader which is designated as the CURRENT INSTRUCTION READER. Fetching the next command from storage is thus simply a READ operation. A consequence of this approach is that the control list (CL) which is part of the reader serves as a powerful subroutine linkage device. There exists a command "VST" (VISIT) which is an unconditional transfer of control (i.e. a branch in the program) to the address specified. The effect of this operation is, however, not merely to transfer control but to store the address of the VST command itself on the top of the control list of the current instruction reader. When the command "TERM" (TERMINATE) is encountered as part of a program, the control list on the current instruction reader is "popped up", i.e. its top element deleted, and the link address of the command pointed to by the popped up datum is placed in the current pointer portion of the current instruction reader. The effect of this is, of course, to resume the computation at the point after the VST command upon subroutine termination. Because of the fact that the control list is a list, visits and termination may be nested in any arbitrary way. This helps make the programming of the computation of recursive functions quite easy.

Within the KLS system indirect addressing of operands is the rule rather than the exception. (This is also the case in the IPL-V system.) In most cases the programmer need hardly be aware of the fact that he is arriving at his operands indirectly. Commands which require a particular kind of operand, e.g. the READ operations require a reader, will begin with the address given in the command, examine the contents of the location specified, and, if the information stored there is not of the type required, will follow the chain of indirection (if possible) until the proper datum is located. If the chain of indirection so initiated does not terminate successfully, then an error is reported to the programmer by means of an appropriate remark sent to an output device of the computer.

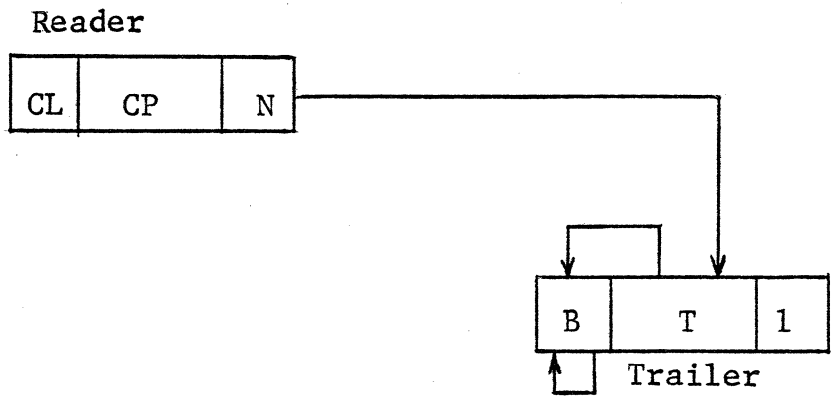
Controlled indirection can also be achieved at the option of the programmer. Associated with certain commands is a designation code called the "Q CODE". For those commands for which Q is meaningful, the following table defines Q:

Q CODE TABLE

Q	MEANING
1	The content of the address given in the command is to be considered as if it had been the address portion of the command and the Q CODE had been absent. (One level address substitution.)
2	This Q CODE applies only to commands the address portion of which leads (by any chain of indirection) to a datum which is either a number or an alphanumeric symbol. The Q = 2 code then signifies that the address of that datum is the desired operand.
3	The same conditions on commands as applicable to Q = 2 apply in this instance. The desired operand is the numeric or alphanumeric datum at the end of the chain of indirection.

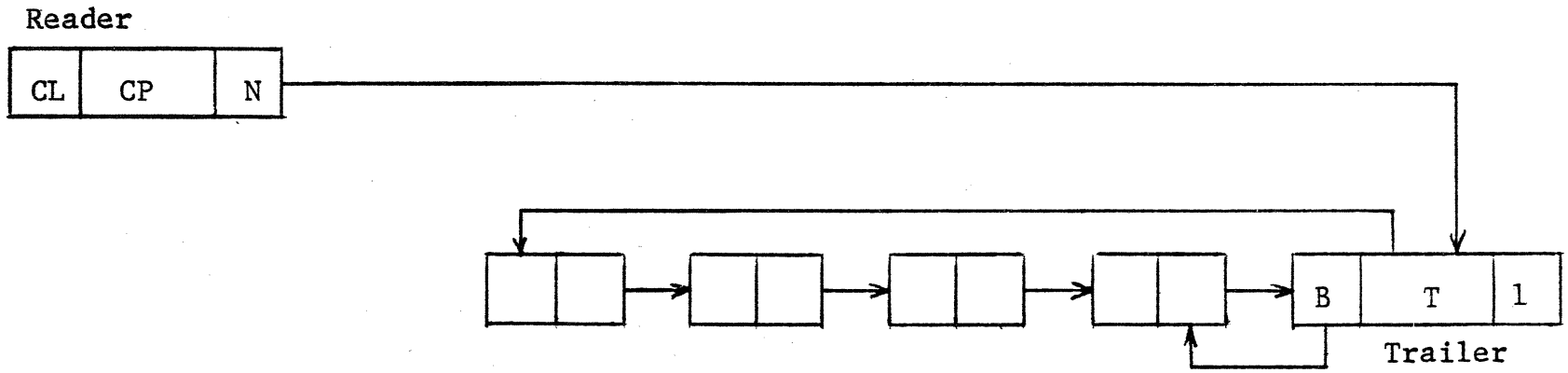
SUMMARY:

The fundamental motivation of this work arose out of questions of machine organizations. The experience gained in designing and using this system has indeed revealed a number of points of great interest in that context. Unfortunately, discussion of hardware issues must be beyond the scope of this presentation. But, apart from gains in that direction, it may be claimed that the KLS system has contributions to make. One of these is that KLS is easy and "natural" to use. Of course, the use of the word "natural" here is undoubtedly related to the fact that the single address concept has become a habit. Nonetheless, the ease with which a small sample of programmers have become enthusiastic converts to this system is remarkable. Another contribution which will lead to further development and is therefore perhaps more important, relates to the so-called "responsibility issue". The question arises as to what program has the responsibility for finally erasing a list which may exist in differing program contexts. When the last appearance of a list name within the KLS system is overwritten or otherwise destroyed, the list associated with that name is erased. Because of the way lists are finally restored to available space, it turns out that nothing is ever erased if the subject program does not run through available space at least once. But the main thing is that a large part of the responsibility (for list erasure) burden has been taken over by the system itself. All other advantages which normally accrue to list processing have been kept. Threaded lists[4] are a subcase of knotted lists, KLS being more general at least in the sense that a given sublist may be a sublist of many lists. The way to see a relationship between KLS and T-Lists is to think of the KLS READER as the T-List ALIAS.



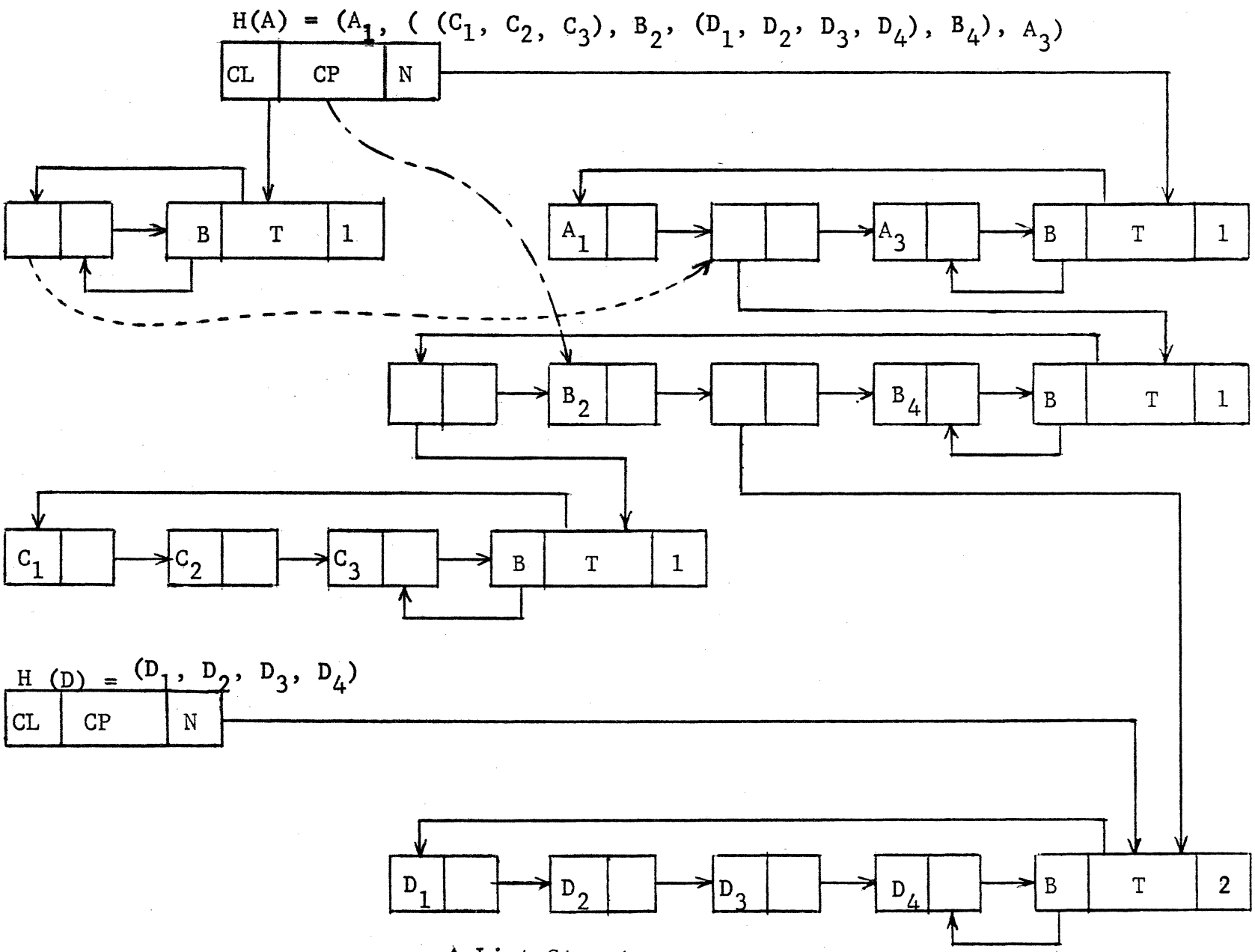
An Empty List

Figure 1



A Simple List

Figure 2



A List Structure
Figure 3

INSTRUCTIONS

INSTRUCTION LIST

INSTRUCTION DESCRIPTIONS

Arithmetic Instructions

ADD	Add
SUB	Subtract
MPY	Multiply
DVD	Divide

e.g. ADD A

The arithmetic instructions above operate on the top two cells of any stack. Q(A) defines the stack and the top two cells of the stack must either contain the operands for the arithmetic instructions or addresses which lead to the operands.

Tally Instructions

TAL	Tally
NTL	Negative Tally

e.g. NTL A

Q(A) must lead to an operand. If the instruction is TAL, the operand is incremented by one (1). The operand is decremented by one by NTL.

Branch Instructions

BWF	Branch on W Flip Flop
BCF	Branch on Control Flip Flop
BDF	Branch on Decision Flip Flop
BSF	Branch on Search Flip Flop
BT(n)	Branch on Toggle "n" (where n may be 0→9)
BTF	Branch on Test Flip Flop

e.g. BDF A

If the particular flip flop or console toggle is set during the execution of the branch instruction testing it, the next instruction is taken from Q(A). Flip flops tested are reset at the conclusion of the instruction.

BU	Branch Unconditionally	
		e.g. BU A
	The next instruction is taken from Q(A).	
BWD	Branch if WO contains a Decimal	
BWA	Branch if WO contains an Address	
BWAN	Branch if WO contains an Alphanumeric	
BWB	Branch if WO contains a Binary Integer	
BWC	Branch if WO contains a Command	
BWD	Branch if WO contains an Element	
BWM	Branch if WO contains a Mention	
BWN	Branch if WO contains a Name	
BWØ	Branch if WO contains an Occurrence	
		e.g. BWC A
	The symbol type contained in the WO register is tested with the appropriate configuration of this command.	
	e.g. If the WO register contains a command at the time when "BWC A" instruction is executed, the next instruction will be taken from Q(A).	
BXL	Branch on X Flip Flop Linear	e.g. BXL A
BXS	Branch on X Flip Flop Structure	e.g. BXS A
	If the X flip flop is in the state being test, the next instruction is taken from Q(A). The X flip flop is left unchanged.	
TERM	Terminate	e.g. TERM
	The instruction reader's (IR's) control list is popped up, the current pointer from the top cell replacing the IR current pointer.	
	If the IR control list is found to be empty, the program is terminated.	
	TERM provides the return for the VST instruction.	

VST	Visit	e.g. VST A
	The current pointer from the instruction reader (IR) is pushed down on the IR control list and Q(A) replaces the IR current pointer. The next instruction executed after VST is taken from Q(A).	
	See TERM for return from VST.	
ASC	Ascend	e.g. ASC
	The IR's control list is popped up and the current pointer from the cell popped off the list is lost. The next instruction is taken in order.	
	NOTE: ASC is not a branch.	
ASG	Assign	e.g. ASG A
	WO assumes the same symbol identity as the cell described by Q(A).	
CLER	Clear	e.g. CLER A
	The symbol described by Q(A) is set to zero and the cell identity set to alphanumeric.	
CXF	Change X Flip Flop	e.g. CXF
	If the X flip flop is in the Linear State, it is set to the Structure State. If it is in the Structure State, it is set to the Linear State.	
PAUSE	Pause	e.g. PAUSE
	The machine stops and waits for operator to restart by depressing the console advance switch.	

RSRL — Reset Reader Linear — e.g. RSRL A

The current pointer in the reader described by "A" is set to point to the trailer of the list into which it is currently pointing.

RSRS — Reset Reader Structure — e.g. RSRS A

The control list of the reader described by "A" is popped up until empty and the reader's current pointer is set to point to the trailer of the main list.

RVØ — Reverse Øne Level — e.g. RVØ A

The control list of the reader described by "A" is popped up and the current pointer from the cell popped off replaces the current pointer in the reader. The control flip flop is reset.

If the control list can not be popped up, the control list being already empty, the control flip flop is set.

RVT — Reverse to Top — e.g. RVT A

The control list of the reader described by "A" is popped up until empty. The last current pointer popped off the control list replaces the current pointer in the reader.

If the control list can not be popped up, the control list being already empty, the reader's current pointer is left unchanged.

SDF — Set Decision Flip Flop — e.g. SDF

The decision flip flop is set regardless of its previous state.

SL — Shift Left — e.g. SL A

The binary fixed point symbol described by "A", where "A" can only be an address

or chain of addresses leading to the symbol, is shifted one bit position to the left. The high order bit is lost and a zero bit is entered into the low order bit position.

If a binary fixed point symbol is not encountered, a statement to this effect will be printed out and the program continued.

SLX — Shift Left Into X — e.g. SLX A

The binary fixed point symbol described by "A", where "A" can only be an address or chain of addresses leading to the symbol, is shifted one bit position to the left, the X flip flop assuming the state of the original high order bit of the symbol and a zero bit is entered into the low order position.

If a binary fixed point symbol is not encountered, a statement to this effect will be printed out and the program continued.

SR — Shift Right — e.g. SR A

The binary fixed point symbol described by "A", where "A" can only be an address or chain of addresses leading to the symbol, is shifted one bit position to the right. A zero bit is entered into the high order bit position and the low order bit is lost.

If a binary fixed point symbol is not encountered, a statement to this effect will be printed out and the program continued.

STØP — Stop Program — e.g. STØP

The program is terminated regardless of the number of entries on the instruction reader's control list.

SXL — Set X Linear — e.g. SXL

The X flip flop is set to the linear state.

SXS — Set X Structure — e.g. SXS

The X flip flop is set to the structure state.

XEQ — Execute — e.g. XEQ (?)

WO must contain an instruction. The WO stack is first restored and then the instruction originally in WO is executed. If the address of the XEQ instruction is blank, the original instruction address from WO is used during the execution. If the address of the XEQ instruction is not blank, this address is substituted in the instruction from WO before execution.

The next instruction to be executed is next to the XEQ instruction in the instruction list unless the instruction from WO resulted in a branch.

CES — Create Empty Storage Cell — e. g. CES A

A cell is obtained from available space. The name of this cell is placed in Q(A), and the identity of Q(A) is made that of an address.

CRN — Create Name — e.g. CRN A

An empty list (empty trailer) is created. Its name is placed in Q(A) with an identity that of a responsible occurrence (RØ).

AVA — Append Value and Attribute — e.g. AVA

"A" must lead to a description list. The "attribute" in WO_0 and the "value" in WO_1 are appended to the description list of the describable list.

WO_0 and WO_1 are unchanged.

EDN — Erase Description List Name — e.g. EDN A

"A" must lead to a describable list. The name of the description list is returned to available space and the list is made non-describable.

If the list is either non-describable or the top element on the list is not a name, the appropriate statement is printed out and the program continued.

EVA — Erase Value and Attribute — e.g. EVA A

"A" must lead to a description list which is searched for an attribute identical to the attribute in WO.

If the attribute in WO is found to be on the description list, both the attribute and its value are removed from the list and the search flip flop set. WO remains unchanged.

If the attribute in WO does not appear on the description list, the search flip flop is reset. WO remains unchanged.

FVA — Find Value of Attribute — e.g. FVA A

"A" must lead to a description list which is searched for an attribute identical to the attribute in WO.

If the attribute in WO is found on the description list, the value is copied into WO and the search flip flop set.

If the attribute in WO does not appear on the description list, WO is unchanged and the search flip flop reset.

IDL — Input Description List — e.g. IDL A

"A" must lead to a describable list. The name of the description list with an identity that of a responsible occurrence is copied into WO.

If the list is not describable or the top cell is not a name, the appropriate statement is printed out and the program continued.

PDL — Push Down Description List — e.g. PDL A

"A" must lead to the name of a non-describable list. WO must contain a name. The name in WO with an identity that of a responsible occurrence is pushed down on the list. The list's trailer is made describable.

If "A" leads to a describable list or WO is not a name, the appropriate statement is printed out and the program continued.

RVA — Replace Value of Attribute — e.g. RVA A

"A" must lead to a description list which is searched for an attribute identical to the attribute in WO.

If the attribute in WO_0 is found on the description list, the value in WO_1 replaces the value on the description list and the search flip flop is set. WO_0 and WO_1 remain unchanged.

If the attribute in WO_0 does not appear on the description list, the search flip flop is reset. WO_0 and WO_1 remain unchanged.

ERL — Erase List — e.g. ERL A

"A" must lead to a list. The list, with the exception of the trailer is returned to available space. The trailer is made empty by setting it to point to itself.

ERN — Erase Name — e.g. ERN A

"A" must lead to a list. The reference counter in the trailer is decremented. If the reference counter goes to zero, the entire list, including the trailer is returned to available space.

ERR — Erase Reader — e.g. ERR A

"A" must lead to a reader. The reader is returned to available space and the reference counter in the list pointed to by the reader is decremented. If the reference counter goes to zero, the entire list, including trailer, is returned to available space.

ERS — Erase Stack — e.g. ERS A

Q(A) must lead to a stack. The entire stack, including the top cell, is returned to available space.

Input Instructions

I (C/F)(1/2) — e.g. IC2

I = Input

C/F = Cards or Flexowriter (Console Typewriter)

1/2 = Mode 1 or Mode 2

Mode 1 — In mode 1, a unity input is expected, i.e., a single symbol or a single list. At the end of the input, WO will contain either the address of the cell containing the single symbol or a responsible occurrence on the list.

Mode 2 — In mode 2, a multiple input is expected, i.e., input a number of symbols or a number of lists. The input is terminated by the Break Pseudo Command (" - "in the command field).

In either mode, if a break is the first input received, WO will be unchanged and the control flip flop set. Otherwise, CFF is reset.

If a PEND (Program End) is encountered, control is transferred to the monitor and the appropriate post mortem performed.

NUL — Nullify — e.g. NUL A

"A" must lead to a reader. The cell pointed to by the current pointer is made null unless the cell contains a trailer.

PRL — Preserve List — e.g. PRL A

"A" must lead to a reader. The symbol plus its ID on top of the list is copied into a cell from available space and pushed down on top of the list.

If the reader points to an empty list, a statement to this effect is printed out and the program continued.

RLD — Restore List to Delimiter — e.g. RLD Λ

"A" must lead to a list. The list is restored linearly up to and including the delimiter. The description list (if it exists) is not included in the restore operation.

If a trailer is encountered before the delimiter, the entire list is erased and the appropriate comment printed.

RSL — Restore List — e.g. RSL A

"A" must lead to a list. The top cell in the list is removed. If a top cell doesn't exist, the control flip flop is set, otherwise, it is reset.

XCL — Exchange on List — e.g. XCL A

"A" must lead to a reader. The top two cells on the list are reversed in order. (There must be at least two cells on the list.)

CØL — Copy Linearly — e.g. CØL A

"A" must lead to a list. A linear copy of the list is made. The name of the copy is placed in WO with a responsible occurrence ID. (Null cells are not copied.)

INN — Input Name — e.g. INN A

"A" must lead to a list. The trailer address is copied into WO with a responsible occurrence (RØ) ID.

LØC	Locate	e.g. LOC A	
	"A" must lead to a list. This list and its sub-list structures are searched for a symbol and ID identical to those in WO. If the symbol and ID are found on the list structure, the search flip flop is set, otherwise, it is reset.		
MNS	Make Name Safe	e.g. MNS A	
	"A" must lead to a list. The reference count in the trailer is increased by one.		
ØCF	Øutput Cell to Flexowriter (Console Typewriter)	e.g. ØCF A	
ØCP	Øutput Cell to Printer	e.g. ØCP A	
	The symbol in the cell described by Q(A) is printed on either the flexowriter or printer.		
	The format is: Symbol location, Symbol ID, and the numeric representation of the symbol.		
ØLF (Blank/1/2)	Øutput List to Flexowriter (Console Typewriter) (Mode)	e.g. ØLF1 A	
ØLP (Blank/1/2)	Øutput List to Printer (Mode)	e.g. ØLP A	
	"A" must lead to a list. The list is printed on either the flexowriter or printer in any one of the three output modes.		
	In the "blank" mode (the mnemonic for the instruction is not followed with either a 1 or 2), the list name is printed on one line followed by a one symbol per line listing on the list, sub-lists are not included.		
	In the "1" mode, the list is printed out horizontally across the page, the name of the list first followed by a colon and apostrophes separating the symbols.		

In the "2" mode, the format is the same as the "1" mode except that there is no separation between symbols.

An alpha numeric symbol requires four type or print character positions. Commands and floating point numbers require 12 positions. All other symbols require 8 positions.

ØSF (Blank/1/2)	Øutput Structure to Flexowriter (Console Typewriter) (Mode)	e.g. ØSF A
ØSP (Blank/1/2)	Øutput Structure to Printer (Mode)	e.g. ØSP A

"A" must lead to a list. The list is printed on either the flexowriter or printer in any of the three output modes. (See Output List commands description of three modes.)

In the output structure commands, the main list, referred to by or through "A", is printed first. If this list has on it references to sub-lists, these sub-lists are printed in the same mode as the main list following it. These lists will be printed in the same order as they occur on the main list with no duplication in the case where two references refer to the same list. Sub-lists of sub-lists are included.

APR	Appoint Reader	e.g. APR A

WO must lead to a list. Q(A) becomes a reset reader with an empty control list for the list.

CØR	Copy Reader	e.g. CØR A

"A" must lead to a reader. A copy of this reader is made in a cell from available space. The new cell address is placed in WO with an address ID.

ICP	Input Current Pointer	e.g. ICP A
	"A" must lead to a reader. The current pointer from this reader is copied into WO with an address ID.	
INR	Input Reader Address	e.g. INR A
	"A" must lead to a reader. The address of this reader is copied into WO with an address ID.	
LCL	Locate Linearly	e.g. LCL A
	"A" must lead to a reader. Each symbol, including its ID, on the reader's main list is compared with the symbol and ID in WO. If the symbol, including its ID, is identical to the symbol and ID in WO, the current pointer in the reader is set to the point to the symbol and the search flip flop is set. If no such symbol is found, the reader's current pointer is not changed and the search flip flop is reset.	
LCS	Locate Structurally	e.g. LCS A
	Locate Structurally is identical to Locate Linearly with the exception that the entire list structure pointed to by the reader, found through "A", is searched for a symbol and ID identical to those in WO.	
SCP	Store Current Pointer	e.g. SCP A
	"A" must lead to a reader. The address in WO is copied into the current pointer of the reader.	
PRS	Preserve Stack	e.g. PRS A
	Q(A) must lead to a stack. The symbol in the top cell of the stack is copied into a cell from available space and pushed down on top of the stack.	

RSD — Restore Stack to Delimiter — e.g. RSD A

Q(A) must lead to the name of a stack. The stack is restored up to and including the delimiter. If no delimiter is found the entire stack is restored and an appropriate comment is printed.

RSS — Restore Stack — e.g. RSS A

Q(A) must lead to a stack. The top cell of the stack is popped up, the symbol in the second cell on the stack taking its place.

If RSS results in WO being restored and WO is a single cell stack, the W flip flop is set, otherwise, it is reset.

If RSS results in any other one cell stack being restored, the control flip flop is set, otherwise it is reset.

XCS — Exchange on Stack — e.g. XCS A

Q(A) must lead to a stack. The top two symbols on the stack are reversed, that is, the second symbol becomes the top symbol, etc. (There must be at least two symbol cells on the stack.)

INP — Input — e.g. INP A

Q(A) is copied into WO. WO will assume the ID of Q(A) with two exceptions:

1) If Q(A) = Trailer, the ID placed in WO will be that of a RO (responsible occurrence).
2) If Q(A) = Reader, the ID placed in WO will be that of an Address.

IWR — Input WO and Restore — e.g. IWR A

Q(A) must lead to a stack. The top cell of the stack is copied into WO and the stack restored.

If Q(A) leads to a single cell stack, the control flip flop will be set, otherwise, it will be reset.

IND — Input Delimiter — e.g. IND

A delimiter symbol is placed in WO.

INS — Input Symbol — e.g. INS A

"A" itself is copied into WO with an alpha numeric ID.

DLE — Delete Linear Element — e.g. DLE A

DLN — Delete Linear Name — e.g. DLN A

DLW — Delete Linear Word — e.g. DLW A

These three commands are identical to their corresponding Read Linear counterparts with the following exception. If the current pointer in the reader for the list being read is pointing to a symbol on the list and not the trailer, the symbol is removed from the list either by deleting or nullifying the cell containing the symbol. The cell is nullified only if it is the bottom cell on the list.

NLR — Null Read — e.g. NLR A

"A" must lead to a reader. The symbol pointed to by the current pointer is copied into WO unless the CP is pointing to a trailer. If the CP is pointing to a trailer, the control flip flop is set, otherwise it is reset.

Read Instructions

R(L/S/X)(W/E/N/M/Ø/EM)

R = Read

(L/S/X) = Linear, Structure or X Mode

W = object of read is Word

E = object of read is Element

N = object of read is Name

M = object of read is Mention

Ø = object of read is Øccurrence

EM = object of read is Element or Mention

Linear Mode e.g. RLN A

"A" must lead to a reader. The reader's current pointer is advanced linearly along the list into which it is currently pointing. After the CP is advanced once, the symbol in the cell to which the CP is advanced is tested for the object of the instruction, in the case of a RLN instruction, a "Name". If the symbol is the object of the instruction, it plus its ID is copied into WO. If the symbol is not the object, the CP is advanced again, etc.

If, after the CP is advanced, the cell tested is found to be a trailer, the previous CP is left in the reader, the control flip flop is set, and WO is left unchanged.

Description list names can not be the object for any of the read instructions.

Structure Mode e.g. RSN A

"A" must lead to a reader. Before the current pointer is advanced, the symbol currently pointed to is tested for the occurrence (Ø) of a Name. If the cell is an Ø, the CP is pushed down on the readers control list (CL) and the Ø replaces the CP in the reader. (The CP is now pointing one level deeper into the list.)

The read mode now is identical to the linear mode with two exceptions:

1) If another ϕ is encountered in the list before the object, the CP is again pushed down on the CL and the reader's CP replaced by the ϕ.

2) If a trailer is encountered before the object, the CL is popped up and the top CP used to replace the reader's CP. (This results in the exit from a sublist into the next lower list.) If the CL is found empty when a trailer is encountered, the CP is left pointing to the cell just ahead of the trailer, the control flip flop is set, and WO is left unchanged.

X Mode — e.g. RXN A

If when a Read X instruction is executed, the X flip flop is in the linear state, the read instruction is executed in the linear mode. If X is in the structure state, the read instruction will be executed in the structure state.

RLF — Read Linear Find — e.g. RLF A

"A" must lead to a reader. The current pointer is advanced linearly along the list, into which it is currently pointing, and each symbol encountered is compared with the symbol in WO_0 using WO_1 (the second cell on the WO stack) as a mask. For every "1" bit in WO_1, the corresponding bit position in the symbols from the list are compared with the corresponding bit positions of the symbol in WO.

If a symbol is found on the list which results in an equal comparison, the symbol in WO_0 and the mask from WO_1 are popped off the WO stack and a copy of the equal symbol pushed down on WO. The CP is left pointing to the symbol on the list and the search flip flop is set.

If no symbol is found on the list which results in an equal comparison, the symbol from WO_0 is popped up, leaving the mask in WO. The CP is left pointing to the bottom symbol on the list and the search flip flop is reset.

RSF — Read Structure Find — e.g. RSF A

The Read Structure Find differs from the Read Linear Find only in the sense that the CP is advanced structurally instead of linearly. The final results are the same for both instructions.

TCE — Test Control List Empty — e.g. TCE A

"A" must lead to a reader. The control list from this reader is tested for empty. If the CL is empty, the test flip flop is set, otherwise, it is reset.

TEQ — Test Equal — e.g. TEQ A

The symbol, not including ID, described by Q(A) is compared with the symbol in WO. If the symbols are equal, the test flip flop is set, otherwise, it is reset.

TGE — Test Greater Than or Equal — e.g. TGE A

The symbol, not including ID described by Q(A) is compared with the symbol in WO. If the symbol in WO is greater than or equal to the symbol described by Q(A) the test flip flop is set, otherwise, it is reset.

TGR — Test Greater Than — e.g. TGR A

The symbol, not including ID, described by Q(A) is compared with the symbol in WO. If the symbol in WO is greater than the symbol described by Q(A), the test flip flop is set, otherwise, it is reset.

TID — Test Identity — e.g. TIS A

The ID of the symbol described by Q(A) is compared with the ID of WO. If the two ID's are equal, the test flip flop is set, otherwise it is reset.

TLE — Test Less Than or Equal — e.g. TLE A

The symbol, not including ID, described by Q(A) is compared with the symbol in WO. If the symbol in WO is less than or equal to the symbol described by Q(A), the test flip flop is set, otherwise, it is reset.

TLEW — Test List Length Equal to WO — e.g. TLEW A

"A" must lead to a list. The number of symbols on the list is computed. (Description lists do not contribute to this count.) If the number of the symbols on the list is equal to the binary integer in WO, the test flip flop is set, otherwise, it is reset.

TLGW — Test List Length Greater Than WO — e.g. TLGW A

"A" must lead to a list. The number of symbols on the list is computed. (Description lists do not contribute to this count.) If the number of symbols on the list is greater than the binary integer in WO, the test flip flop is set, otherwise, it is reset.

TLL1 — Test List Length Equal ønе — e.g. TLL1 A

"A" must lead to a list. If the list length is equal to one, the test flip flop is set, otherwise, it is reset.

TLLO — Test List Length Empty — e.g. TLL1 A

"A" must lead to a list. If the list is empty, the test flip flop is set, otherwise, it is reset.

TLS — Test Less Than — e.g. TLS A

The symbol, not including ID, described by Q(A) is compared with the symbol in WO. If the symbol in WO is less than the symbol described by Q(A), the test flip flop is set, otherwise, it is reset.

TMI — Test for Minus Sign — e.g. TMI A

If the symbol described by Q(A) is less than zero (minus), the test flip flop is set, otherwise, it is reset.

TNE — Test Not Equal — e.g. TNE A

The symbol, not including ID, described by Q(A) is compared with the symbol in WO. If the symbols are not equal, the test flip flop is set, otherwise, it is reset.

TNS — Test Nature of Symbol — e.g. TNS A

The symbol and ID described by Q(A) is compared with the symbol and ID in WO respectively. If the comparison proves equality, the test flip flop is set, otherwise, it is reset.

TUN — Test for Unity — e.g. TUN A

If the symbol described by Q(A) is unity, the test flip flop is set, otherwise, it is reset.

TZR — Test for Zero — e.g. TZR A

If the symbol described by Q(A) is zero, the test flip flop is set, otherwise, it is reset.

PL0-4 — Put on Top (Modes 0 through 4) — e.g. PL2 A

"A" must lead to a list. The symbol in WO is put on top of this list (the top of a described list is the cell next to the name of the description list). The ID of the symbol is determined by the mode of the instruction. For PL2, the 2-mode will result in the symbol being placed on top of the list with an ID that of a responsible mention.

Modes:	Resulting Symbol ID
PL0:	Responsible Øccurrence (RØ)
PL1:	Non-responsible Occurrence (NRØ)
PL2:	Responsible Mention (RM)
PL3:	Non-responsible Mention (NRM)
PL4:	The ID of the symbol as it appeared in WO.

For PL0, PL1, PL2 or PL3, the symbol ID in W0 must be that of a name.

PL5-9 Put on Bottom (Modes 5 through 9) e.g. PL7 A

"A must lead to a list. The symbol in W0 is put on the bottom of the list (the bottom of the list being the cell just ahead of the trailer). The ID of the symbol is determined by the mode of the instruction. For PL7, the 7-mode will result in the symbol being placed on the bottom of the list with an ID that of a responsible mention.

Modes:	Resulting Symbol ID
PL5:	Responsible Øccurrence (RØ)
PL6:	Non-responsible Øccurrence (NRØ)
PL7:	Responsible Mention (RM)
PL8:	Non-responsible Mention (NRM)
PL9:	The ID of the symbol as it appeared in W0.

For PL5, PL6, PL7, or PL8, the symbol ID in W0 must be that of a name.

IN0-4 Insert Before (Modes 0 through 4) e.g. IN3 A

"A" must lead to a reader. The symbol in W0 is copied into a cell from available space. This cell is inserted in the reader's list structure just ahead of the cell to which the readers CP is pointing. The ID of the new cell is dependent upon the mode of the instruction. For IN3, the 3 mode will result in the new cell being inserted into the list structure with a non-responsible mention ID.

Modes:	Resulting Symbol ID
IN0:	Responsible Occurrence (RØ)
IN1:	Non-responsible Occurrence (NRØ)
IN2:	Responsible Mention (RM)
IN3:	Non-responsible Mention (NRM)
IN4:	The ID of the symbol as it appeared in W0.

For IN0, IN1, IN2, or IN3, the symbol in W0 must be that of a name.

IN5-9 — Insert After (Modes 5 through 9) — e.g. IN8 A

"A" must lead to a reader. The symbol in W0 is copied into a cell from available space. This cell is inserted in the reader's list structure just after the cell to which the reader's CP is pointing. The ID of the new cell is dependent upon the mode of the instruction. For IN8, the 8-mode will result in the new cell being inserted into the list structure with a non-responsible mention ID.

Modes:	Resulting Symbol ID
IN5:	Responsible Øccurrence (RØ)
IN6:	Non-responsible Øccurrence (NRØ)
IN7:	Responsible Mention (RM)
IN8:	Non-responsible Mention (NRM)
IN9:	The ID of the symbol as it appeared in W0.

For IN5, IN6, IN7 or IN8, the symbol in W0 must be that of a name.

PS0 — Push Down on Top of Stack — e.g. PS0

Q(A) must lead to a stack. The symbol in W0 is pushed down on the top of the stack.

STL — Store Øn List — e.g. STL A

"A" must lead to a reader. The symbol and its ID are copied into the cell pointed to by the reader's current pointer.

If the CP is pointing to a header or trailer or if W0 contains either a header or trailer, the symbol is not copied into the cell pointed to by the CP and an appropriate statement is printed out.

STS — Store on Stack — e.g. STS A

Q(A) must lead to a stack. The symbol in WO replaces the top symbol on the stack.

ETM — Enter Tracing Mode — e.g. ETM

ETM starts the tracing of instructions and the tracing continues until terminated with a LTM (Leave Tracing Mode). A Visit instruction will cause a temporary termination of the tracing, the tracing continued when the Visit is terminated bring the instruction list back to the level at which the ETM initiated the tracing.

LTM — Leave Tracing Mode — e.g. LTM

Instruction tracing is discontinued upon execution of LTM. Tracing is also discontinued when the instruction list terminates.

Bit Manipulation Instructions

SBIT — Set Bit

RBIT — Reset Bit

CBIT — Complement Bit

TBIT — Test Bit

e.g. CBIT A

WO must contain an operand (BFL, DI, BI, AN). Q(A) must lead to a BI (binary integer), "i", where $0 \leq i \leq 23$. The "i^{th}" bit position of the symbol in WO is operated upon as follows:

SBIT: The "i^{th}" bit is set to a "1".

RBIT: The "i^{th}" bit is set to a "0".

CBIT: The "i^{th}" bit is complemented.

TBIT: If the "i^{th}" bit of WO is a "1", the test flip flop is set, otherwise, it is reset.

Logical Instructions

AND	Logical And
OR	Logical OR

e.g. OR A

Q(A) must lead to a stack. The top two cells on the stack must either be operands or addresses leading to operands. The ID's of the operands found must be identical.

The top two cells on the stack are popped off and the logical "and" or "or" preformed on the two operands. The result of the logical operation is pushed down on the stack.

PROGRAMMING SYSTEM

KLS PROGRAMMING SYSTEM
(Philco S-2000 Version)

The KLS programming system enables one to simulate KLS programs on the Philco S-2000. The system includes its' own operating system, assembler and simulator with extensive debugging facilities.

All KLS system runs are batch runs and each program is assembled by the loader (part of the input system) at load time and then executed.

KLS simulates programs (including assembly time) at the rate of 225 instructions/second without tracing and 20 instructions/second with tracing.

Input to the KLS System is either from cards or the flexowriter. Each line of input represents either a pseudo command or an executable instruction.

PSEUDO COMMAND FORMAT

Col. 1 - 9 ignored
10 - 16 location field
17 - 24 command field
25 - 40 variable field
41 - 80 ignored

EXECUTABLE INSTRUCTION CARD FORMAT

Col. 1 - 9 ignored
10 - 16 location field
17 - 24 command field
25 - 31 address field
36 Q - code
41 - 80 ignored

FLEXOWRITER INPUT FORMAT

Pertinent fields of a line of input from the flexowriter are separated by a ",", and the last field is terminated by a "|".

E.g.:

```
,IC2|
A1,RSE,G-R25,2|
,CLER,ALPHA, 1|
,LIST|
A1,RSE,G-R25 |
,CLER,ALPHA,1|
```

PSEUDO COMMANDS:

PROGRAM: The Program card is the first card of each program and defines the program running requirements. The PROGRAM card requires a special format defined later.

PEND: The PEND card is the last card of each program. When encountered, the present program is terminated and the next program is started.

SUBR: The SUBR command will cause (upon completion of the present input sequence) the searching and loading of a subroutine from one of the libraries. The search is made for a program on the libraries whose name corresponds to the name in the variable field of the SUBR card. Upon finding such subroutine, an IC2 command is used to load the subroutine into KLS. It may in turn call for more subroutines.

Up to 4 subroutine library tapes are allowed in the system. Tape units for the libraries are ITAPE, ITAPF, ITAPG, & ITAPH.

The format of an individual library is a tape produced from cards (code-mode) of which the first card is a KLS-LIB card followed by debugged programs, followed by a LIB-END card.

KLS-LIB The first card of a library tape. (see SUBR card description.)

LIB-END: The last card of a library tape. (see SUBR card description.)

GMODE	The GMODE command declares that all symbolic addresses which follow (until encountering a LMODE card) are to be assumed global.
LMODE	The LMODE command declares that all symbolic addresses which follow (until encountering a GMODE card) are to be assumed local.
+	A + command causes the local symbol table (LST) to be erased when encountered.
-	Serves to terminate the execution of the IC2 instruction.
LIST	Defines the beginning of a list. All cells which follow (until a LEND card) are to link together so as to form a list. The location field may contain a name for the list.
LEND	Defines the list end. (see LIST card)
DLST	Same as a LIST card except the list is defined as a describable list. The first symbol on such a list must be the name of the description sub-list.
DI	This defines a cell as representing a decimal integer. The decimal integer is defined in the variable field, with or without sign, followed by a maximum of 6 decemal digits.
BI	This defines a cell as representing a binary integer. The binary integer is defined in the variable field, with or without sign, followed by a maximum of 7 decimal digits.

BFL

This defines a cell as representing a binary floating poent number. The number is defined in the variable field, with or without sign, followed by 1 to 6 decimal digits. A decimal point may be included. A base 10 exponent (one digit) may follow and must be preceded by a sign.

E.g.:

```
3.1416
-27.2
+16.4 - 3
-28145.6 - 4
```

AN

This defines a cell as representing a 4 character alpha-numeric constant. The first 4 characters of the variable field define the constant.

MAN

This defines a series of cells which contain alpha-numeric information. The variable field for this command starts in column 25 and is terminated at the first occurrance of a "$". In the absence of a "$" column, 80 is the last column of the variable field. The first 4 characters of the variable field form the first alpha-numeric cell, the next 4 characters form the next cell, etc. A partial cell is filled with trailing "n" characters. A MAN command may only occur between a LIST and LEND card, e.g.,

```
A LIST
  BFL  3.1416
  MAN  KLS SYSTEM $
  LEND
```

A

This defines a cell as being an address cell. The variable field contains the symbolic address of the cell which is being addressed.

RO

This defines a cell as representing a responsible occurrance of a sub-list. The sub-list name is in the variable field.

RM	This defines a cell as representing a responsible mention of a sub-list. The sub-list name is in the variable field.
ϕ	This defines a cell as representing an occurrance of a sub-list. The sub-list name is in the variable field.
M	This defines a cell as representing a mention of a sub-list. The sub-list name is in the variable field.
ϕCTAL	This defines a cell contents (both the symbol and the cell identification). The variable field contains 11 octal digits representing the cell contents. This pseudo command should be used only with extreme caution.
R	This defines a cell as representing a reader. The cell is made a reader (reset) of the list whose name is in the variable field. A reader may not be a cell on a list, therefore, must not occur between a LIST and LEND card.

SYMBOLIC ADDRESSES

Symbolic addresses (in the location field, address field or variable field) when encountered are treated as either local or global depending as to whether the input system is in the local mode or global mode. However, Wo is always treated as a special global symbol regardless of mode.

If the input system is in the global mode, a local symbolic address may be referenced by preceding the symbolic address with a "L-".

If the input system is in the local mode, a global symbolic address may be referenced by preceding the symbolic address with a "G-".

Symbolic addresses must be of the form 1 alphabetic characters followed by up to 4 alpha-numeric characters.

KLS PROGRAM CARD

1 - 16 Ignored (may be comments)

17 - 24 PROGRAM

25 - 32 Program Name

33 △

34 N,Y, or T - Trace all commands (if T, test toggle 10)

35 N,Y, or T - Trace VST, TERM and Branch Commands which are met (if T, test toggle 11)

36 N,Y, or T - Execute ETM commands, otherwise ignore ETM commands (if T, test toggle 12)

37 - 39 △△△

40 - 52 Toggle settings for toggles 0 - 12:

0 - reset
1 - set
△ - ignore

53 - 55 △△△

56 - 62 Instruction count limit; if blank or zero, 2000 is assumed.

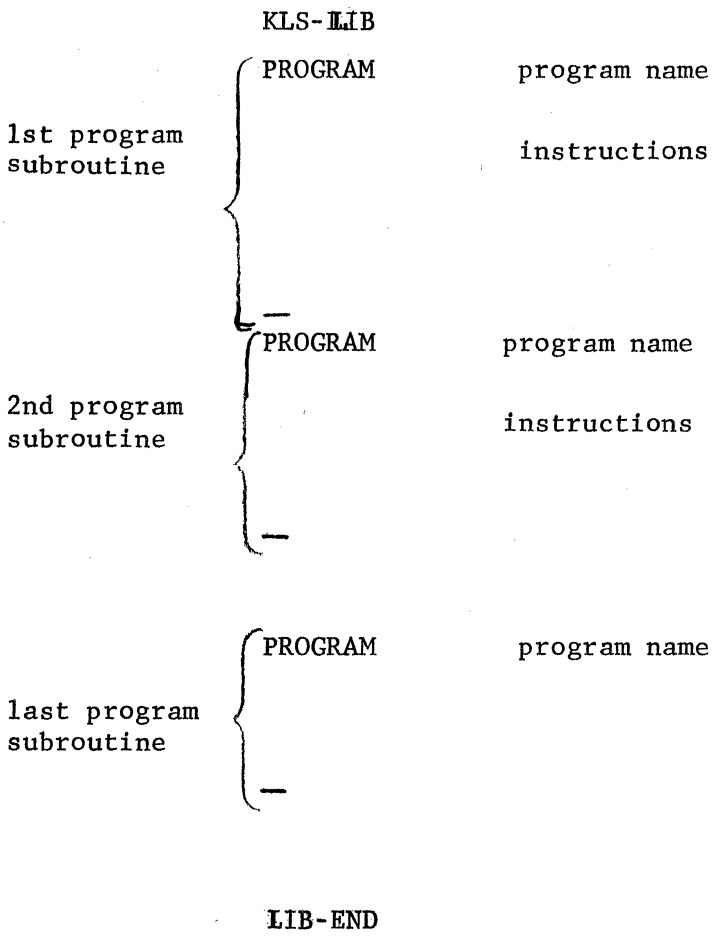
63 Philco S-2000 Dumps:

1 - Dump lower memory in command format
2 - Dump upper memory in octal format
3 - Dump upper & lower memory
△ - No dump

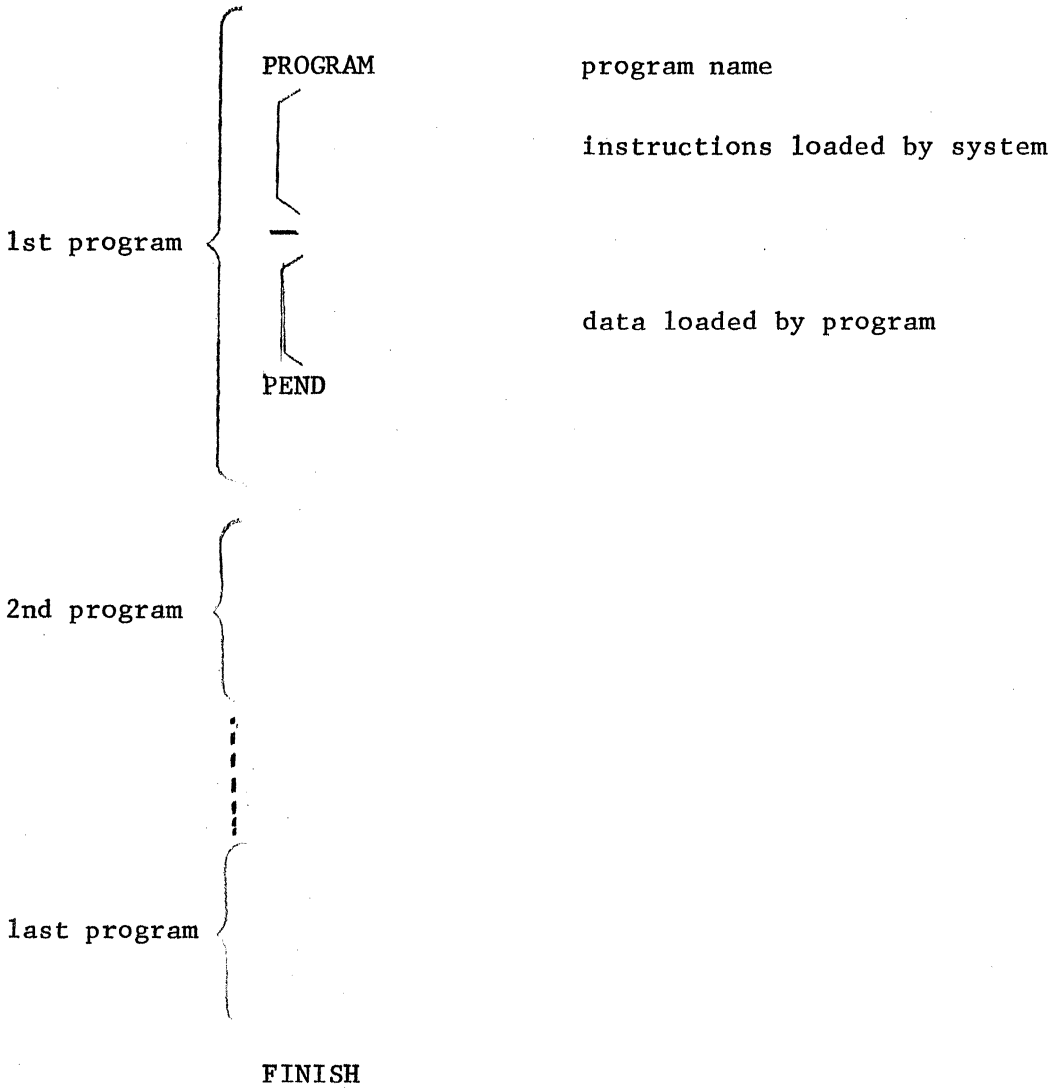
64 KLS Global Symbol Table Dump:

1 - Dump items named in GST; if a list, dump list only
2 - Dump items named in GST; if a list, dump list structure.
△ - No dump.

KLS LIBRARY DECK FORM



KLS SYSTEM INPUT DECK



KLS SYSTEM PROGRAM PROCESSING PROCEDURE:

The following steps define the processing procedure of KLS programs:

1) Search input tape for a PROGRAM card. If a FINISH card is encountered during this search, the run is completed.
2) Initialize available space, toggle settings, etc. according to PROGRAM card.
3) Execute an IC2 command (loading program).
4) Transfer control to first cell on the list whose name is in Wo. Therefore immediately following the PROGRAM card, there must be a list of instructions, the first of which, will be the starting instruction.
5) Upon termination of the program, the required post mortems will be performed. Step 1 is then performed.

KLS PROGRAM TERMINATION:

A KLS Program will be terminated by any of the following circumstances:

1) Execution of a STOP command.
2) Running out of instructions by either bumping into a trailer or executing a TERM command when the instruction reader control list is empty.
3) Error; a diagnostic will be printed.

CHARACTER SET:

0	n - Restore carriage on flexo; filler character on printer, i.e., does not print
1	
2	
3	| - Not legal on flexowriter. (as output)
4	@
5	=
6	;
7	&
8	'
9	+
A	.
B	)
C	%
D	?
E	"
F	-
G	$
H	*
I	#
J	,
K	(
L	:
M	e
N	≡
O	↵
P	<
Q	⌴
R	>
S	Δ blank
T	
U	
V	
W	
X	
Y	
Z	

KLS OPERATING PROCEDURE:

ASSEMBLY:

1) TAC II absolute assembly.
2) Process 2 libraries:
   a) latest APED standard TAC II library
   b) and RCS library on tape 46.
3) Add RPL to separate tape.
4) 4 copies of code edit unless otherwise indicated.

EXECUTION:

1) Set Breakpoint Switch to STOP.
2) If computer appears to be looping perform loop test as follows:
   a) Set toggle 47, if " OK" is not typed on flexo within 6 seconds KLS is in a loop and operator should manually stop machine, then follow step 3. Reset toggle 47.
3) If any stop should occur, record program name and JA register contents only. Then manually jump to 1000L.
4) Always return flex output to run requestor.
5) SMART program card:
   a) A dump card is included although if should only be used if the KLS system cannot be recovered by a jump to 1000L.
   b) Preclear memory starting at location 1000 with "JBTL3". This is indicated by an "F" in col. 19.

SAMPLE PROBLEM
(WANG'S ALGORITHM)

As an illustration of the use of the system, the code for an example problem is given. This code is a realization of a proof procedure for the propositional calculus developed by Hao Wang (6). For a detailed description and justification of the procedure, the reader is referred to the paper cited. In brief, however, the procedure is as follows:

Given any theorem to be proved, that theorem is first written in Polish prefix notation. This notation is convenient in that it brings the main connective to the front of the expression. An arrow is prefixed to the total expression which is then thought of as having a LEFT and a RIGHT side with respect to the arrow.

Initially, of course, the LEFT side is empty. Associated with each connective and the side on which it appears, there is a rule which eliminates that connective and which dictates which of the connected terms are to be either left in place or moved to the other side of the arrow. Certain rules also dictate that a subproblem is to be generated, that is that the connected terms are to be reassigned to LEFT and RIGHT sides respectively in two different ways. One of the expressions so created is then set aside as a problem to be treated subsequently. When finally all connectives have been removed by the application of the rules, the LEFT side is inspected to see if it contains terms in common with the RIGHT side. If it does, then that portion of the proof indicates validity. If all subproblems end validly, the theorem is true, otherwise not.

The way the program handles these issues can be seen from the accompanying illustration of the proof of the simple theorem (Not (P Or Q)) Imp (Not P)).

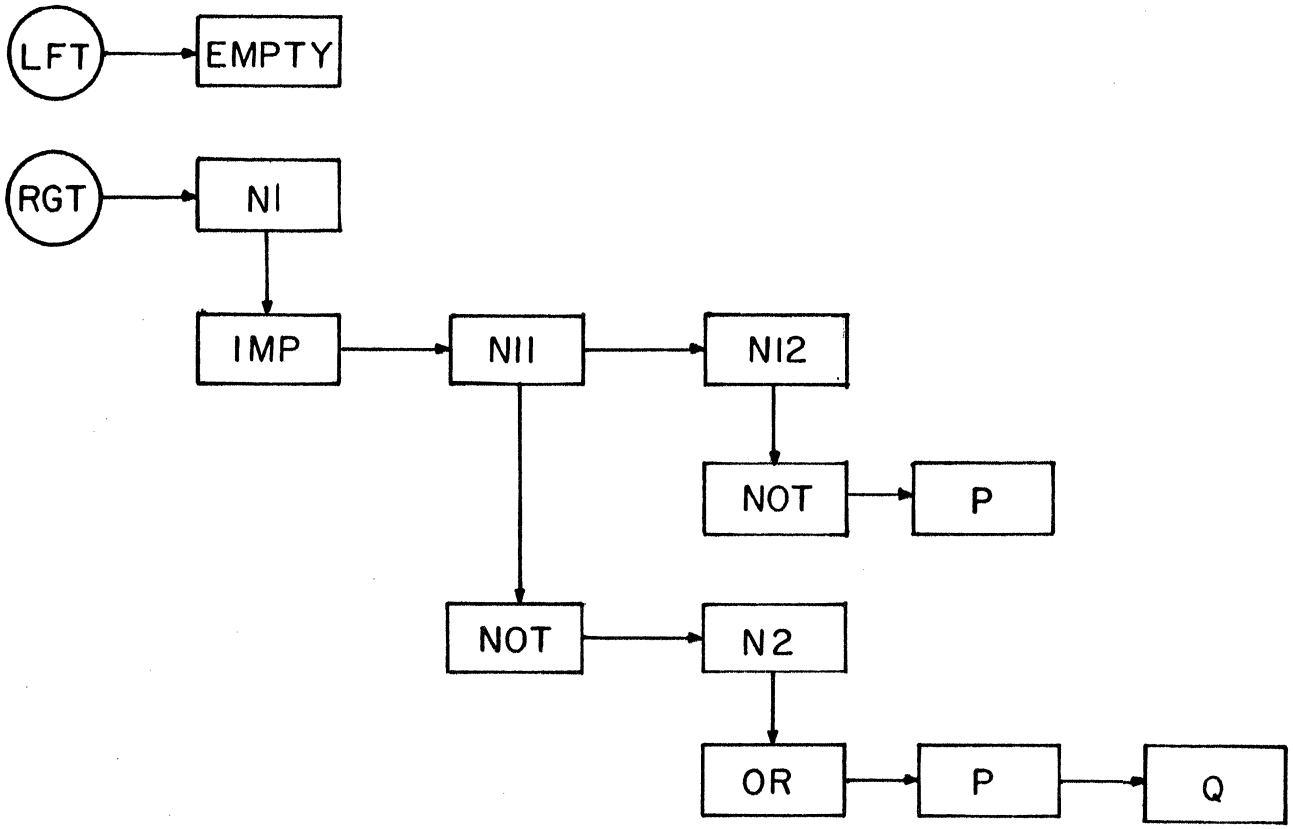
Initially the LEFT side is empty, while the RIGHT side contains the whole theorem in a list structure form. Notice that every connective is the first (i.e. top) element of a seperate list which is, in fact, a sublist of some list structure. The program now looks for the first name of a list on the LEFT side. Finding none there, it proceeds to look for a name on the RIGHT side. This name is that of a list which has as its top element the connective "IMP". The appropriate rule for the "IMP" connective found on the RIGHT side causes the first term of the implication to be moved to the LEFT side and the second term to remain where it is, the connective

itself being eliminated. Thus the second picture is generated. This time, the search for a name on the LEFT side is successful. The "NOT" connective is found and the appropriate rule applied. Going on in this fashion finally results in the fifth picture. There the literal "P" appears on both the LEFT and the RIGHT sides and there are no more names of lists on either side. Since, in this simple example, there are no subproblems, the theorem is proved.

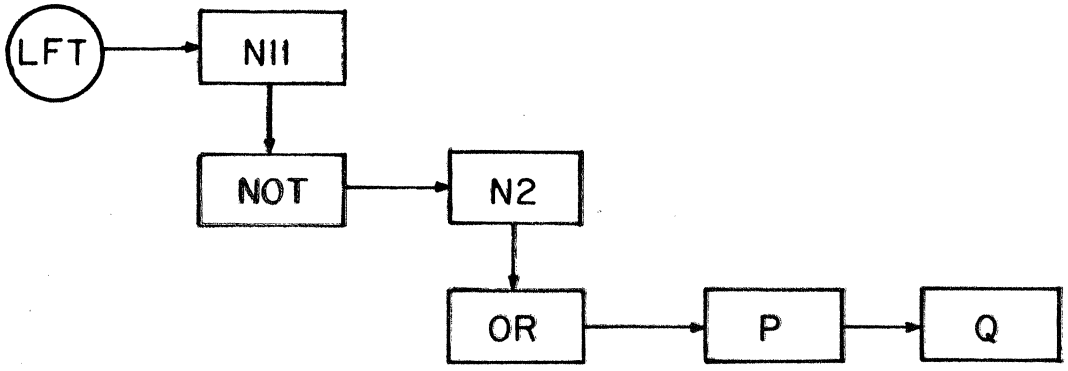
The accompanying program appears more complicated than the discussion would lead one to believe mainly for the reason that it includes editing functions both for input and output. The theorem to be proved, for example, is put into the system in ordinary parenthethis notation which is then translated to list structure format by the initialization procedure. Similarly, the output is presented in a form a little more readable to a nonspecialist than a straight printout of the internal machine format would be. Both these editing programs, by the way, provide good exercises to the interested reader.

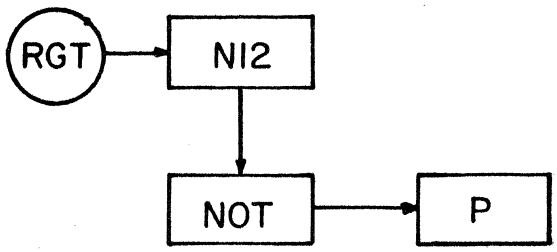
An important consideration relating to the efficiency of the system is that when a list structure is moved, only the name of that structure is actually transferred to a new location in memory. The structure to which that name "points" is carried along as a natural consequence. This illustrates the important property of list manipulation systems that entities of arbitrary complexity may be manipulated as if they were in fact single symbols. Another important observation is that the program is not written with any detailed knowledge of the depth or complexity of the list structures which it must manipulate. The program knows, so to speak, only the general characteristics of these structures. The complexity is contained in the way in which the data is stored, as opposed to the more usual situation in which data is stored very "regularly" but programs are very complex. Furthermore, the complexity in data organization arises partially as a result of processing, i.e. is created dynamically. In particular it is not planned in great detail--only in general outline-- by the programmer. Thus a great planning burden is lifted from the shoulders of the programmer.

((NOT (P OR Q)) IMP (NOT P))

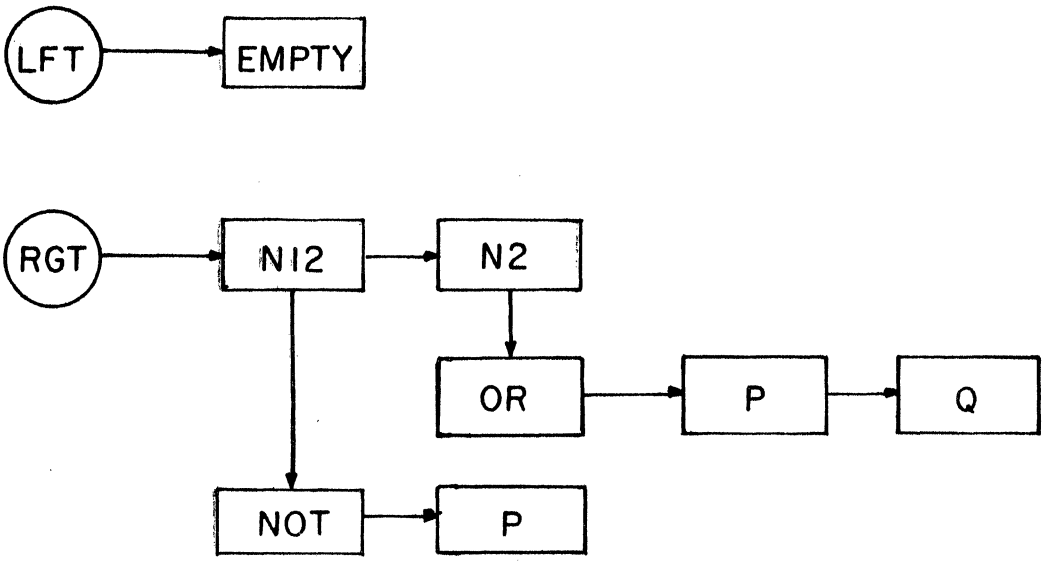


I

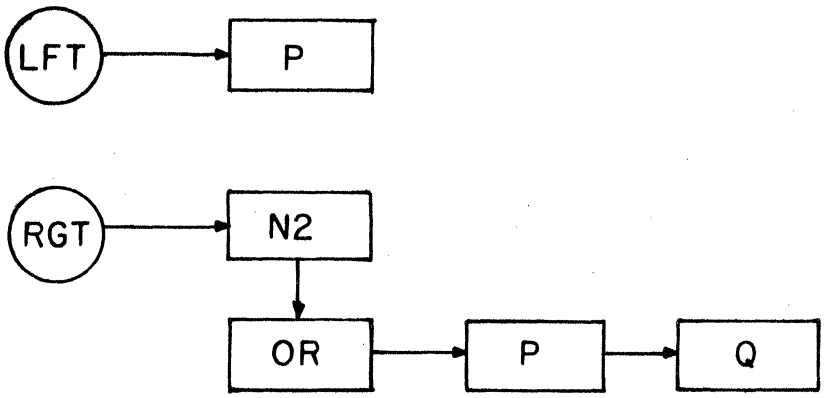




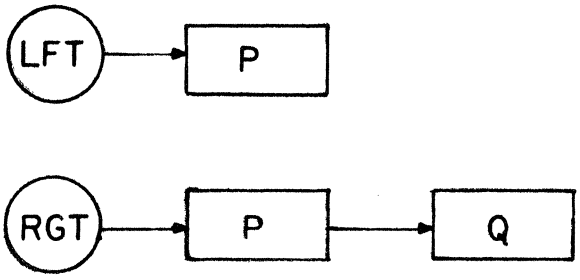
II



III



IV



V

```
        PROGRAM WANG3      YYY
        LIST
EDIT    CRN     X3         2
        CRN     LED
        INN     LED
        STS     OUTP
        BT0     A00
        IC2
        BCF     J5
        OSP2    W0
        APR     ER1
        RLW     ER1
        VST     ED2
        ERL     L
        ERL     R
        INN     OUTP
        PL0     R
        VST     A00        3
        BU      EDIT
ED26    BU      ED24
LPP     TID     LP
        BTF     J1
        TERM
J1      VST     ED23
        TERM
J5      STOP
ED2     RLW     ER1
        TID     NOT
        BTF     XK1
        VST     LPP
        PL4     OUTP
        RLW     ER1
XK1     PL4     OUTP
        RLW     ER1
        VST     LPP
        PL9     OUTP
        RLW     ER1
        TERM
ED23    CRN     W0
        PS0     OUTP
        VST     ED2
        INN     OUTP
        RSS     OUTP
        TERM
DONE    AN      END
LP      AN      (
RP      AN      )
A00     INN     EG
A0      CLER    R1
        CLER    LFT
        CLER    RGT
        APR     R1
A011    RLW     R1
        APR     LFT
        RLW     R1
        APR     RGT
        BU      A01
A1      CLER    CRT
        RSRL    RGT
        RSRL    LFT
        RLN     LFT
        BCF     A2
        APR     CRT
        NUL     LFT
        RLW     CRT
        BCF     A3
A3      FVA     LCL
        BSF     A31
A31     XEQ
A2      RLN     RGT
        BCF     A4
        APR     CRT
        NUL     RGT
        RLW     CRT
        BCF     A5
A5      FVA     RCL
        BSF     A51
A51     XEQ
RNOT    RLW     CRT
        PL9     LFT
        BU      A01
ROR     RLW     CRT
        PL9     RGT
ROR1    RLW     CRT
        PL9     RGT
        BU      A01
RIMP    RLW     CRT
        PL9     LFT
        RLW     CRT
        PL9     RGT
        BU      A01
```

```
RAND    VST     A20
        RLW     CRT
        PL9     W2
        BU      ROR1
RIFF    VST     A20
        RLW     CRT
        PL9     RGT
        PL9     W1
RIF1    RLW     CRT
        PL9     LFT
        PL9     W2
        BU      A01
LAND    RLW     CRT
        PL9     LFT
        BU      RNOT
LOR     VST     A20
        RLW     CRT
        PL9     W1
        BU      RNOT
LIMP    VST     A20
LIM1    RLW     CRT
        PL9     W2
        BU      RNOT
LIFF    VST     A20
        RLW     CRT
        PL9     W2
        PL9     LFT
        BU      RIF1
A01     INN     LFT
        VST     OSPA
        INN     RGT
        VST     OSPA
        BU      A1
A4      RSRL    LFT
A41     RLW     LFT
        BCF     A50
        LOC     RGT
        BSF     VALD
        BU      A41
A50     OCP     D1
        BU      A6
D1      AN      FALS
QED     AN      QED
VALD    OCP     QED
        BU      A6
A6      CLER    R1
        CLER    RGT
        CLER    LFT
        CLER    CRT
        INN     X3
        APR     R1
        RLW     R1
        BCF     A7
        RSRL    R1
        CRN     X4
        RLW     R1
        NUL     R1
        PL9     X4
        RLW     R1
        NUL     R1
        PL9     X4
        INN     X4
        APR     R1
        BU      A011
A7      OCP     DONE
        TERM
A20     COL     LFT
        PL5     X3
        STS     W1
        COL     RGT
        PL5     X3
        STS     W2
        TERM
        LMODE
1       +
G-OSPA  CRN     OUTP
A0      APR     RDR     2
        VST     A2
        OLP2    OUTP
        CLER    OUTP
        CLER    RDR
TERM    TERM
A2      BWN     A3
A5      PL9     OUTP
        BU      A11
A3      TLL0    W0
        BTF     A4
        INP     LP
        PL9     OUTP
        RSW     RDR
```

```
        VST     A2
        INP     RP
        PL9     OUTP
        RVO     RDR
        BU      A11
A4      INP     MT
        BU      A5
LP      AN      (III
RP      AN      )III
MT      AN      ()II
A11     RLW     RDR
        BCF     TERM
        PRS     WO
        INP     COMA
        PL9     OUTP
        RSS     WO
        BU      A2
COMA    AN      •III
        +
        GMODE
        LEND
LCL     LIST
NOT     AN      NOT
        BU      ROR1
OR      AN      OR
        BU      LOR
AND     AN      AND
        BU      LAND
IMP     AN      IMP
        BU      LIMP
IFF     AN      IFF
        BU      LIFF
        LEND
RCL     LIST
        AN      NOT
        BU      RNOT
        AN      OR
        BU      ROR
        AN      AND
        BU      RAND
        AN      IMP
        BU      RIMP
        AN      IFF
        BU      RIFF
        LEND
```

```
EG      LIST
        RO      L
        RO      R
        LEND
```

```
                                    NO,        COMMAND          3          WO=8
        PROGRAM WANG                                                    01/02/62   12118,1
  INPUT LIST REPLACEMENT L
  INPUT LIST REPLACEMENT R
  INPUT LIST REPLACEMENT R10
  INPUT LIST REPLACEMENT N1
  INPUT LIST REPLACEMENT N11
  INPUT LIST REPLACEMENT N12
  INPUT LIST REPLACEMENT N2
!(   (   (   NOT P   )   AND (   NOT Q   )   )   IMP (   P   IFF Q   )   )
!
!IMP !AND !NOT !P   !NOT !Q   !IFF !P   !Q   !
!AND !NOT !P   !NOT !Q   !
!IFF !P   !Q   !
!NOT !P   !NOT !Q   !
!IFF !P   !Q   !
!NOT !Q   !
!IFF !P   !Q   !P   !
!
!IFF !P   !Q   !P   !Q   !
!Q   !
!P   !Q   !P   !
   AN       QED
!P   !
!P   !Q   !Q   !
   AN       QED
   AN       END
!(   (   NOT (   P   OR  Q   )   )   IMP (   NOT P   )   )
!
!IMP !NOT !OR  !P   !Q   !NOT !P   !
!NOT !OR  !P   !Q   !
!NOT !P   !
!
!NOT !P   !OR  !P   !Q   !
!P   !
!OR  !P   !Q   !
!P   !
!P   !Q   !
   AN       QED
   AN       END
!(   (   NOT P   )   IFF (   (   NOT P   )   AND (   (   NOT P   )   OR  Q   )   )
 !
!
!IFF !NOT !P   !AND !NOT !P   !OR  !NOT !P   !Q   !
!AND !NOT !P   !OR  !NOT !P   !Q   !
!NOT !P   !
!NOT !P   !OR  !NOT !P   !Q   !
!NOT !P   !
!OR  !NOT !P   !Q   !
!NOT !P   !P   !
!Q   !
!NOT !P   !P   !
!Q   !P   !
!P   !
```

Proof of Theorems by Wang's Algorithm Produced by KLS Interpreter

January 2, 1962

References:

1. Collins, G.E., A Method for Overlapping and Erasure of Lists, Comm. ACM, 3 (1960), 655 - 657.

2. McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I., Comm. ACM, 3 (1960), 184 - 195.

3. Newell, A., et al, Information Processing Language V Manual, Sect. I, II, Rand Corp., P.-1918, March, 1960.

4. Perlis, A.J. and Thornton, C., Symbol Manipulation by Threaded Lists, Comm. ACM 3 (1960), 195 - 204.

5. Shaw, J.C., et al, A Command Structure for Complex Information Processing, Proceedings of the 1958 WJCC, May 1958.

6. Wang, Hao, "Toward Mechanical Mathematics", IBM Journal of Research and Development, V.4, No. 1, 1/60, p.2.