# CAL Lisp: Preliminary Description

(Bill Bridge)
Gene McDaniel
Paul McJones

A project for CS 220 and CS 199.

Full of typos!
Please send a copy of this to
Dan Bobrow.
Hope to see this run soon.

A+

## INTRODUCTION

Lisp has been in the forefront of symbol manipulation programming languages for more than a decade and interactive implementations on timeshared computer systems have kept it up-to-date. In particular BBN Lisp for the SDS 940 and, more recently, the PDP-10 provide a very large address space and powerful primitive operations allowing interactive debugging and editting extensions, among others, to be written in Lisp itself. (1,2). CAL Lisp is an attempt to continue this line of development and hopefully harness a fast computer, the CDC 6400, and a not-so-fast operating system, the CAL Timesharing system.

BBN has diverged from Lisp 1.5 (3) in several ways which increase efficiency without, it is felt, detracting from the language. First, a linear pushdown stack replaces the association list scheme for variable bindings and allows compiled functions to reference their arguments directly. (These compiled functions still place the variable names with the values to facilitate debugging and maintain compatibility with interpreted functions.) Second, the property list of an atom is no longer used by Lisp to hold function definitions and constant values. Each atom has three settable attributes: a definition cell, referenced when the atom is used as a function name; a "zero-level value" cell, used when the atom occurs as a free variable bound nowhere in the stack; and a property list cell, purely for use by the user (including the Lisp-coded extensions) as a hatrack. Third, the treatment of special forms, functions taking unevaluated and/or a variable number of arguments, is nicely handled, with any of the four combinations allowed. A number of generalizations and extensions in many other areas exist and in PDP-10 Lisp provision is made for an openended set of datatypes, after the spirit of SNOBOL4.

We wish to mimic the BBN Lisp faithfully enough to be able to utilize much of its library of software (editor, breakpoint routine, etc.) and yet still clean up a few of the inconsistencies which naturally occur as large systems evolve over the years.

Our machine has a relatively large physical memory (23,000 words available to a user process) with an awkward 60 bit wordlength. Secondary storage capacity is limited with 8 million words of disk and 300 thousand words of extended core storage (ecs). Thus much of our effort has been applied to dense packing of data structures. And since no paging or segmentation hardware exists, we have had to carefully plan their inclusion in the software. The timesharing system does provide a disk file structure buffered by the ECS, and a software map to swap the central memory to/from files (residing on disk or in ECS.)

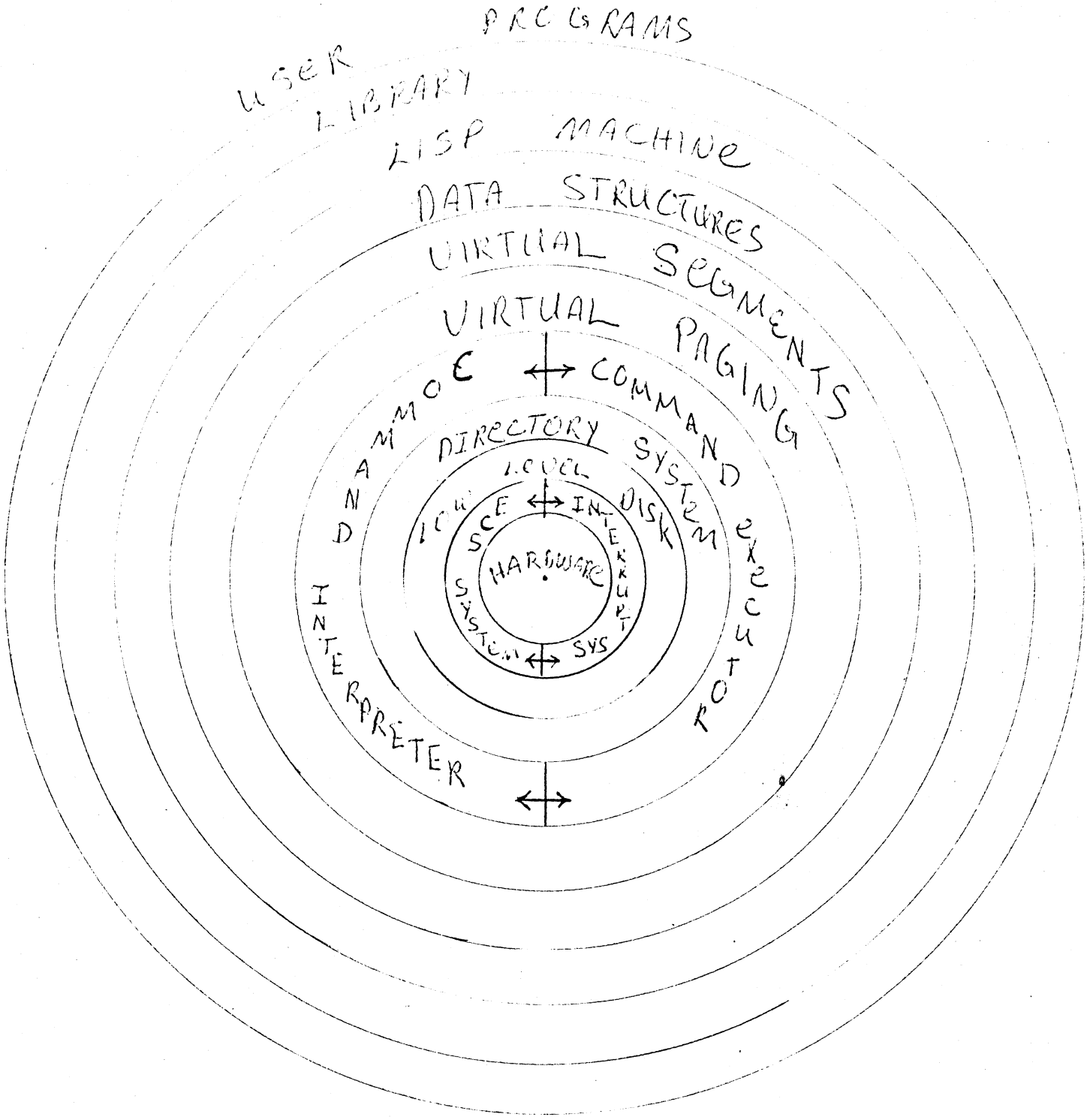
The work is being done in BCPL, a high level language ostensibly suited for systems programming. This should greatly reduce the time spent on coding, but the resultant object program may be too large and slow to be feasible. We envision rewriting sections in assembly language as necessary.

(Figure 1.)

The world of CAL Lisp can be characterized by a complex of eleven concentric rings. Each ring represents a virtual machine, from the hardware (innermost ring) to the user program (outermost ring). The hardware consists of 32 K (K=1024) of CM (60 bit words), 1 CPU, 10 PPUs, 300K ECS and eight million words of disk storage. The ECS and disk act as secondary and tertiary storage devices. The lack of really large scale bulk storage imposes severe restraints on upon the upper layers of virtual machine. Next the ECS-INTERRUPT SYSTEM layer implements a number of orthogonal actions which aid in the implementation of a timesharing system. Of particular interest to Lisp is the ECS-CM interaction, ie., process swapping from and to CM and ECS. Associated with each user process is a software map which describes which blocks of CM are to be written where into ECS (and vice versa). The CAL Lisp system uses that software map to implement the virtual memory layer (described later).

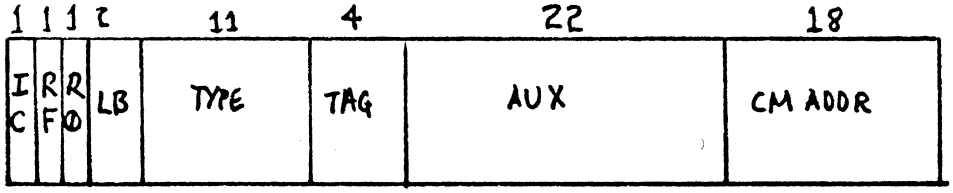
In the CAL Timesharing system files can exist either on the disk or in ECS. The low level disk system was designed to allow unused portions of a file to reside on the disk while the currently used portions of a file may reside in ECS. The directory system implements complicated searching procedures to allow the user to employ symbolic names while describing or asking for his files. The command interpreter-executor follows the two fold task of determining the "semantics" of user or subsystem requests and then attempting to service those requests. The virtual processor viewed from this point is the CAL Timesharing System.

The lowest level processor directly implemented by CAL Lisp is the virtual memory mechanism. There is a 256K virtual memory divided into 1K of 256 word pages. The Lisp system references a virtual address through special macros: fetchvirtual(VA) and storevirtual(VA). They take a virtual address as input and return the current CM address of the word named by that virtual address. A page table is maintained which contains a one word entry for every 256 word page (See figure 2 ). The page table, which is also paged, contains one word for every block in use, but need not contain entries for pages which aren't being used. A virtual address only need be right shifted a sufficient number of bits to produce the correct index in the page table. The virtual memory mechanism returns the CM address of the word being referenced if it is around; otherwise a call is made to the disk system to change the subprocess map.




# LISP PROCESSOR

Figure 1




IC=1 iff this page in core

RF=1 iff this page referenced since last time the page handler looked

RO=1 iff this is a read-only, library page

LB = library file number

TYPE = type flag bits

TAG = auxilliary type information, includes pseudo-page flag

AUX= bits used to handle the various datatypes; used for free chain ptrs, etc.

CM ADDR = block address if IC=1

# Page Table Entry Format

Figure 2

The disk system brings the missing file page into ECS from the disk and places it in the current subprocess map--thereby effecting the paged memory. This action may be preceded by the swapout-to-the-disk for some page chosen by the virtual memory mechanism. This occurs when CM is fully maped with file blocks of the virtual address space (See figure 3 ). There is also a block buffer table which maps 256 word blocks of CM onto the virtual page currently residing there. Storing into a virtual address is more complicated: the paging mechanism must check of the virtual address being stored into belongs to a read only page. If so, a copy of this page is made, the r/o page is returned to the disk, the page table entry is changed to r/w and the appropriate CM address is returned to the caller.

Important attributes of the virtual memory mechanism may be summarized in the following ways:

1) A large virtual address space is mapped into a small physical address space through use of software paging.

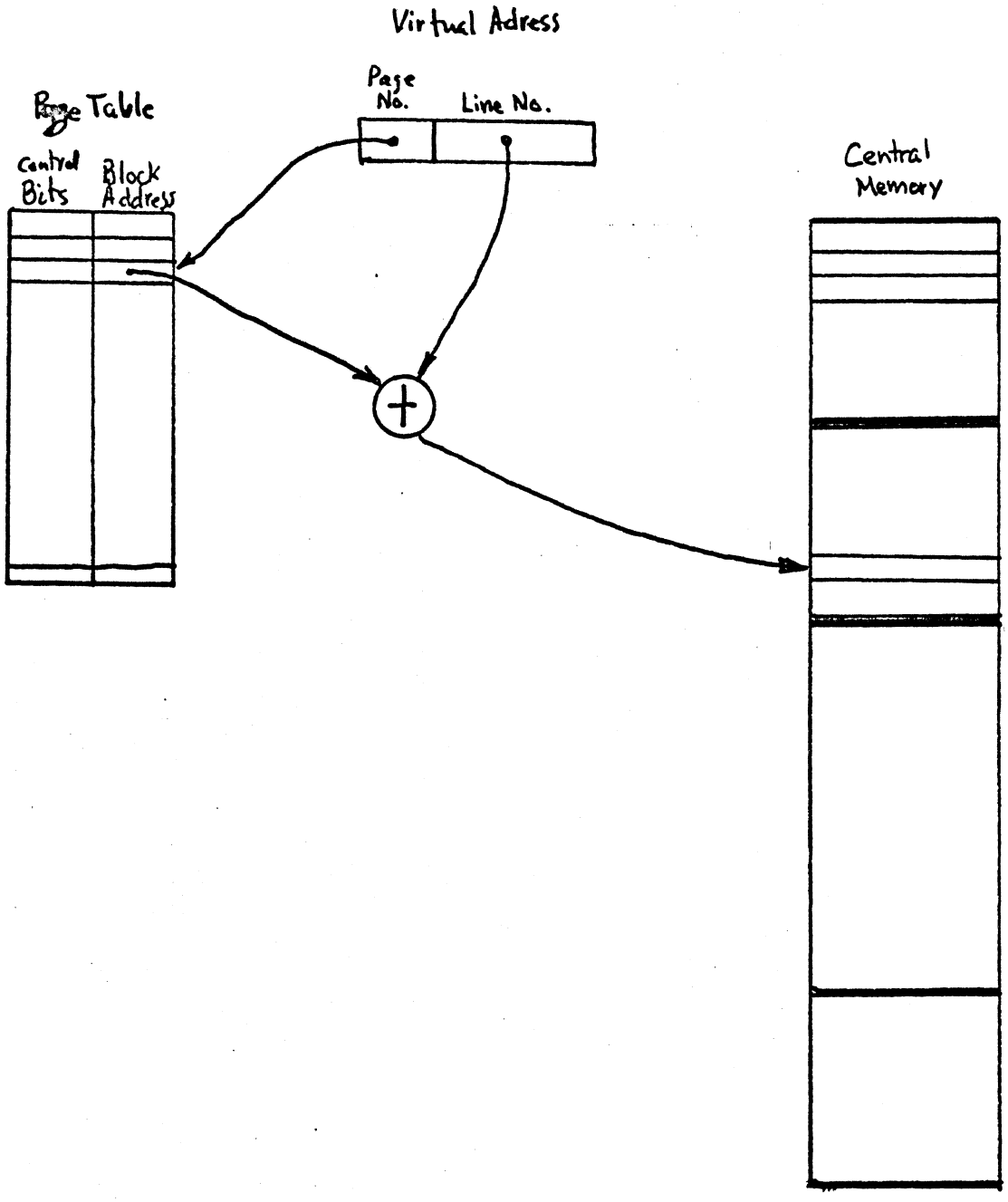
2) Software paging is accomplished by use of the subprocess map maintained by the operating system (which uses it to remove the user process to ECS when his time slice is up, or vice versa).

3) A Least Recently Used swapping strategy is implemented to decide which virtual page must be swapped out of CM to make room for a different virtual page.

4) When the system is smoothly running swapping experiments can be performed to determine good swapping strategies: the swapping mechanism may decide to keep more pages in ECS than there is room for in CM. This would mean that when a page fault occured it would not be necessary to retrieve the page from the disk.

Virtual Segments

The BBN Lisp partitions the virtual address space into various types. An atom could only occur on a page within certain bounds in the virtual address space. While this allows for quick type checking by doing address arithmetic, it also sets a static limit on the number of words which can be devoted to any one type of object. CAL Lisp places no such address restrictions on objects. When a page is created, it is assigned to a specified type (eg., atom, string, etc). There is no limit to how many virtual pages can be devoted to a specified type beyond the intrinsic limit of the 256 K virtual memory.




Virtual Adress Mapping

Figure 3

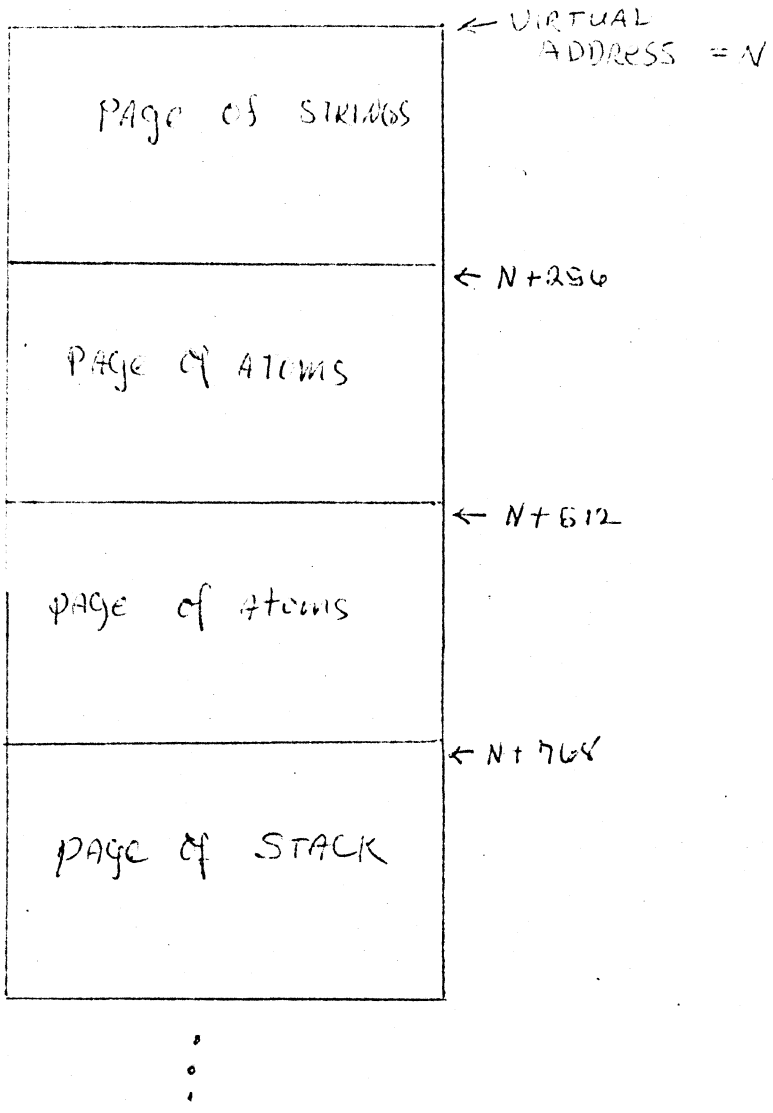
This introduces the following problem: There is no way of predicting in what order the pages of various types will be created. One page might be of type atom while the next two virtual pages might be strings or numbers. (See figure 4 ) . It is apparent that consecutive virtual addresses may lead to objects of completely different type. While all primitive functions should't have to worry about virtual address type, it is clear that chaos will result if virtual addresses aren't neatly "bundled."

The virtual segment layer of the CAL Lisp system stands between the virtual memory and the data structures for the reasons described above. The virtual segment mechanism follows algorithms fitted specifically for the type of the data structure it may be handling at a given time. Thus, when the string creation routine decides it wants k words of string space it should never be concerned with page boundries. The virtual segment processor hands it a linked chain of k words which uses virtual addresses as pointers. Thus the distinction about pages disappears. The stack primitives also illustrate the need for the virtual segmentor. The stacks used in Lisp are implemented through pagina and the virtual memory mechanism. A function using the stack only wants to deal with the stack interms of a current pointer and an index into the stack. Every function is guaranteed 256 consecutive entries in the stack without regard to page boundries. The virtual segment machinery takes all necessary steps to assure that when pushing or poping the stack causes a stack page to be swapped, the resultant physical change in CM is invisible to the function. In general the virtual segment mechanism decides where the next owrd or serveral fords of a given type are to come from. At times the distinction between the virtual segmentor and other portions of the system seems to blur: The primary task of CONS as implemented by BBN and CAL Lisp (aside from putting the car and cdr pointers into the cons node) is to use a relatively complicated algorithm to find the most efficient page to put the new cons node on. This is effectively the kind of task the virtual segmentor performs.

Data Structures

CAL Lisp data structures are the objects which the Lisp machine manipulates and references. These objects include lists,arrays,strings, stacks, atoms, hash tables, and pointer blocks.(among others).

Lists were designed to take advantage of the large--60 bit--word on the CDC 6400. CAL Lisp "lists" are constructed out of nodes each with two fields,




VIRTUAL Address Space and the Problem with page boundries

Figure 4

car and cdr. Each field contains an eighteen bit virtual address; a node occupies 36 bits and there are three list nodes packed into every two machine words (allowing four bits per node for marking,etc). (See figure ). This is accomplished by unpacking the third node into its two constituents (the car and cdr subfields) which are then placed in separate, consecutive words. Each two pages worth of list space requires three pages worth of addresses, so list pages are allocated three at atime, adjacently. Only the first two are mapped to the disk (or even created at all). The third page, known as a pseudo page, gets a page table entry with a special file number and type bits. The list primitives compute the position of these addresses in the remaining portions of the other two pages. Although the two "real" pages are allocated simultaneously, the actual disk space may not be created until later, as needed.

Free storage is kept on a separate list for each "real" page assigned to lists. The head of this list, together with a count of its length are kept in the "aux" field of the corresponding page table entry. The chains to not use regular virtual memory addresses, but rather a scheme using short pointers local to the page (See figure ).

Note that "paired" list pages need not be swapped into adjacent core blocks. Nor do both pages need to be in CM at the same time. The virtual segmentor is concerned with maping references to lists on a pseudo page to a reference on an appropriate "real" page.

The Lisp machine uses two stacks, the control stack with an entry for every function called, and the parameter stack with an entry for each instance of a variable binding (effectively a linearized A-list). The pages constituting each of these stacks are linked together by two doubly linked lists winding through the page table entries (See figure ). To avoid going through virtual memory there is an "active" region of the parameter stack. Up to 256 of the arguments, local variables, and free variables of the current function are kept contiguously in core (in the "stack"). A cell is maintained to contain the CM address of the first stack entry for the current function; compiled code can make "indirec indexed" references through this cell. Control stack entries are illustrated in figure .

An atom is defined to be a word (on a page of type "atom") consisting of three fields:

1) The virtual address of the function definition or the virtual address of NIL.
2) The virtual address of the value or the virtual address of the atom NOBIND.
3) The virtual address of the property list or the virtual address of NIL.

The virtual address of the atom NIL is zero. The atoms whose print names represent the character set used by CAL Lisp can be addressed by adding a constant to their rotated ascii value.

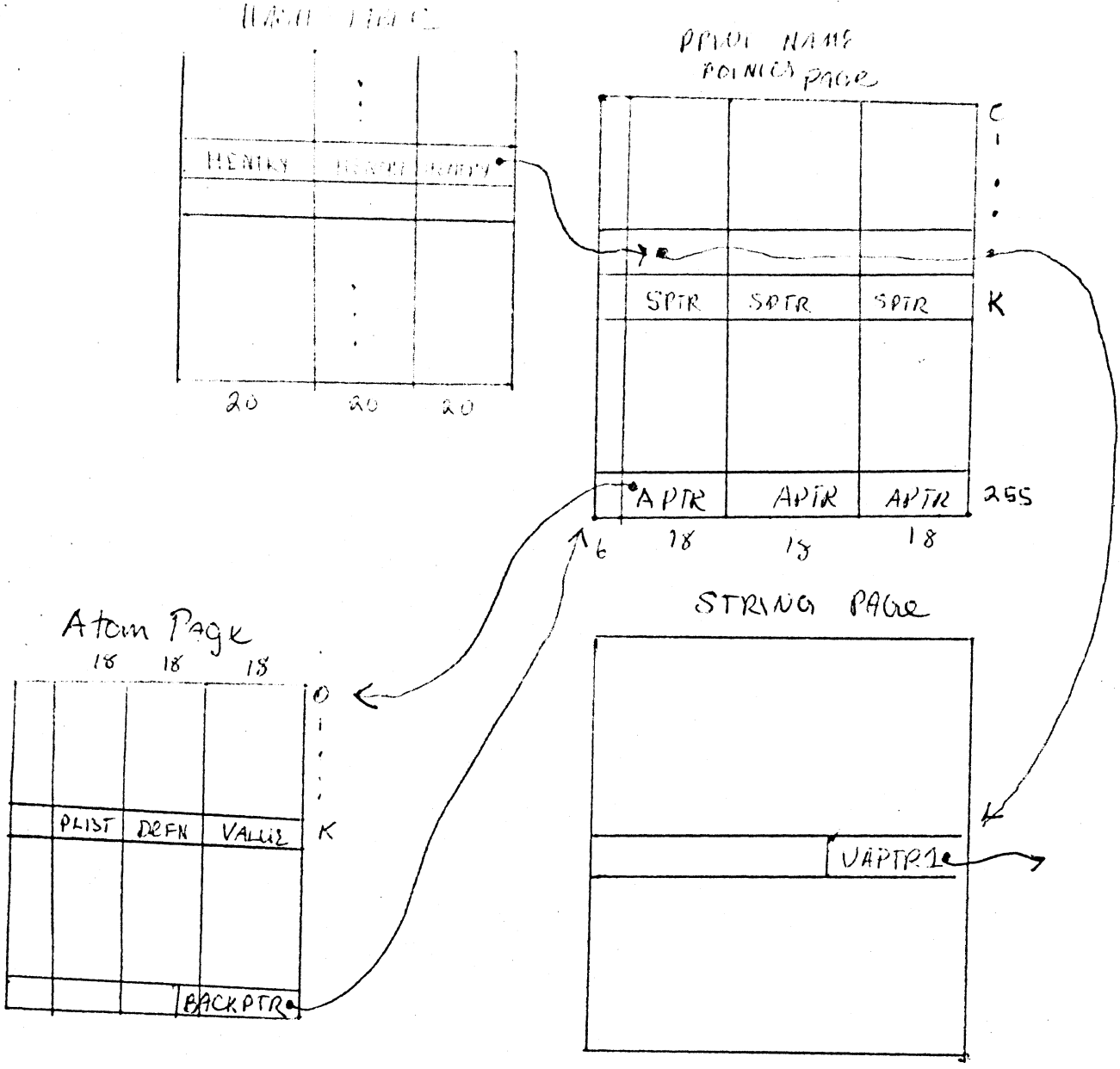
Associated with three pages of atoms is one page of print name pointers. The print name pointer page contains the virtual addresses of the strings which represent the print name for each atom (See figure 5). To take advantage of the large word size, there are three print name pointers per word in a print name pointer page.

The hash table contains entries which point to a field in a print name pointer page. To take advantage of the large word size, a word in the hash table contains three 20 bit entries. The hash table entries contain an 18 bit virtual address into a print name pointer page and a 2 bit modifier denoting which "column" in the word is being referenced. (See figure 5).

When an atom is to be created, the hashing mechanism is presented with the print name of the new atom. If no such atom already exists, the string representing the print name is created, the atom and appropriate print name pointer entry are created and the caller is returned the virtual address of the new atom. An appropriate entry is also made in the hash table. If an atom with that print name already exists, the virtual address of that atom is returned.

A string has two parts. There is a header word describing the string and a series of linked words which contain the string characters. Rotated ASCII is the character code used by CAL Lisp (and by CAL Timesharing System). The words are linked by virtual address pointers so there is no concern with page boundries. There are Lisp machine primitives which directly handle strings (get the i th character, make a list or strings, are two strings equal, etc.). (See Figure 6.)

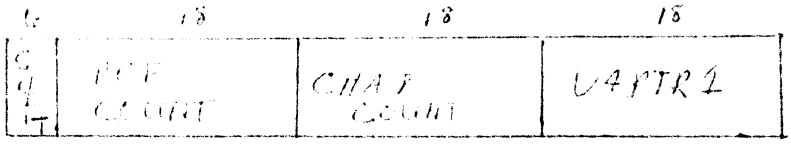
Numbers exist in pages of their own. In a fashion analagous to the BBN Lisp implementation of small integers, small integers in CAL Lisp are never written into "real" space. A bias added to the virtual address of the small integer will give the "real" value of the integer. Large integers and floating point numbers look like normal 60 bit words of that kind. Each has its own page type.



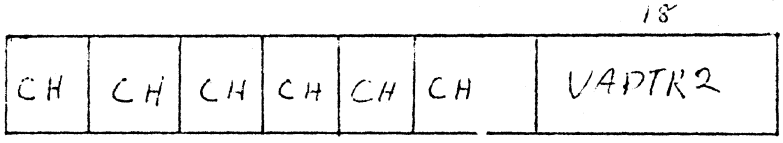

HENTRY = 18 bit Virtual address + 2 bit column id
SPTR = 18 bit Virtual address to STRING ~~BIG~~ HEADER WORD
APTR = Virtual Address of Atom Page
PLIST = Virtual Address of Property List
DEFN = Virtual Address of Definition
VALUE = Virtual Address of VALUE

Figure 5

HEADER WORD




STRING BODY WORD




VAPTR1 = Virtual Address Pointer to First String Body Word

Char Count = NUMBER OF CHARACTERS IN THIS WORD

Ref Count = POSITION IN FIRST STRING Body Word where STRING BEGINS

VAPTR2 = Virtual Address Pointer to NEXT STRING WORD or ZERO

CH = Seven bit rotated ASCII character or ZERO

# STRINGS

Figure 6

## THE LISP MACHINE

Lisp machines are partial recursive functions, and we may employ the jargon of recursive definitions in describing the nature of these programs. When defining a function "in terms of itself", a starting point is needed to prevent circularity. The basis in Lisp is a collection of primitive functions coded into the system and called subroutines or subrs for short. Subrs manipulate the objects defined by Lisp (lists, atoms, numbers, strings, etc.) , perform input/output, and alter the flow of control in function evaluation; about 100 of them are needed altogether.

Continuing the analogy we see that the induction step consists of the ability to define new functions in terms of old ones, via the lambda expression. In BBN and CAL Lisp, the interpretation of functions is subdivided among a collection of subrs. EVAL is essentially a function caller - it evaluates the arguments (if necessary) and transfers control to a subr or a compiled function definition. COND, PROG, etc. are all subrs due to the flexibility of the special form mechanism; they can be independently changed or even written in Lisp and compiled. (For example, see Figure 7 for a Lisp definition of LABEL.)

Underlying the Lisp machine is the pushdown stack, on which are bound the values of LAMBDA and PROG variables and the states of functions currently being evaluated. Actually two stacks are used: a parameter stack for bindings and a control stack for saving return links. This makes the variable searcher a little simpler at the cost of increased page breakage to support the two stacks. References to the control stack simply go through the "fetch virtual" and "store virtual" actions of the paging mechanism. The parameter stack is treated specially,

```
(PUTDQ LABEL
       (NLAMBDA (FN EXP)
                (LIST (QUOTE NLAMBDA)
                      (QUOTE X)
                      (LIST (LIST (QUOTE NLAMBDA)
                                  FN
                                  (LIST (QUOTE EVAL)
                                        (LIST (QUOTE CONS)
                                              FN
                                              (QUOTE X))))
                            EXP)))))
```

Look Ma, no hands!

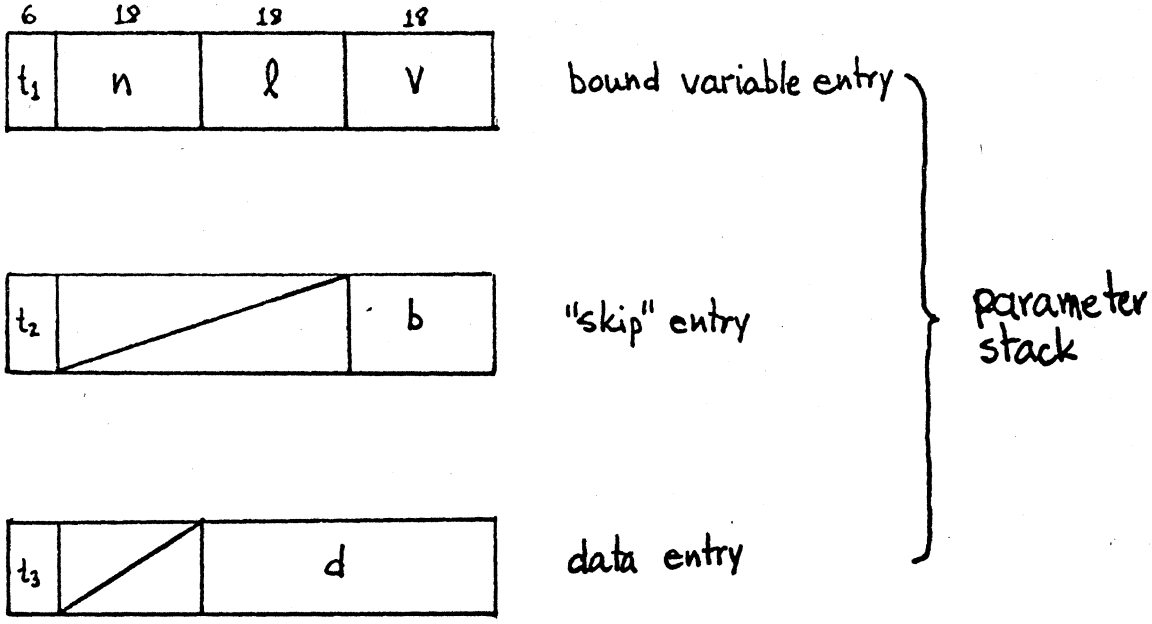
Figure 7

though. Two contiguous core blocks are reserved to hold the two top (most recent) pages, and a pointer into this CM region to the first parmater of the current function is maintained. In fact compiled code will run with this pointer in an index register so parameter references will be very fast. This scheme limits the amount of local storage (arguments and temporaries) a function may have to one page's worth, split across the two contiguous blocks.

One drawback of the pushdown stack over an association list for variable bindings is that the FUNARG device of Lisp 1.5 cannot be simply implemented. We propose to use a mechanism, used by PDP-10 LLsp (2), which handles functional arguments but not functions which return functions as value (at least when the returned function references the environment from which it was returned.) A special parameter stack entry distinguishable from a bound variable entry is used when a FUNARG is evaluated to cause the variable searcher to skip down the stack, ignoring the range of bindings not logically accessible to the functional argument. (See Figure 8 for stack entry formats.)

References to "global" variables, unbound by any function but with a value cell whose contents are not equal to NOBIND, ordinarily cause a search of the (unbypassed portions of the) parameter stack, to ascertain that the variable is indeed unbound. It would be desirable to keep a small hash table of recently used global variables and their values, to avoid the stack search if possible. Unfortunately this idea seems to conflict with the "skips" in the stack. A variable would be put in the hash table only if a stack search did not find a binding for it, but this search must look inside "skipped" regions of the stack. Otherwise, if a variable bound in such a region were then placed in the hash table, then when the skip was removed the correct binding of the variable would be hidden.

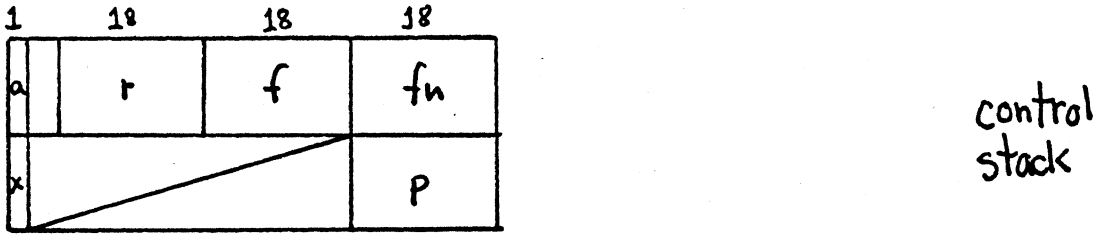
This problem can be circumvented but it may not be worth it.




n = name = an atom not T or NIL
v = value = any Lisp object
l = location = stack pointer or NIL - used by compiled code for free variables

b = bypass = pointer to stack entry beneath this one

d = data = an unboxed integer




fn = function name = atom or S-expression
f = function definition = subr or S-expression or compiled code
r = return address, in system code iff a = 0
p = caller's parameter stack pointer, which is in same page as current pointer iff x = 0

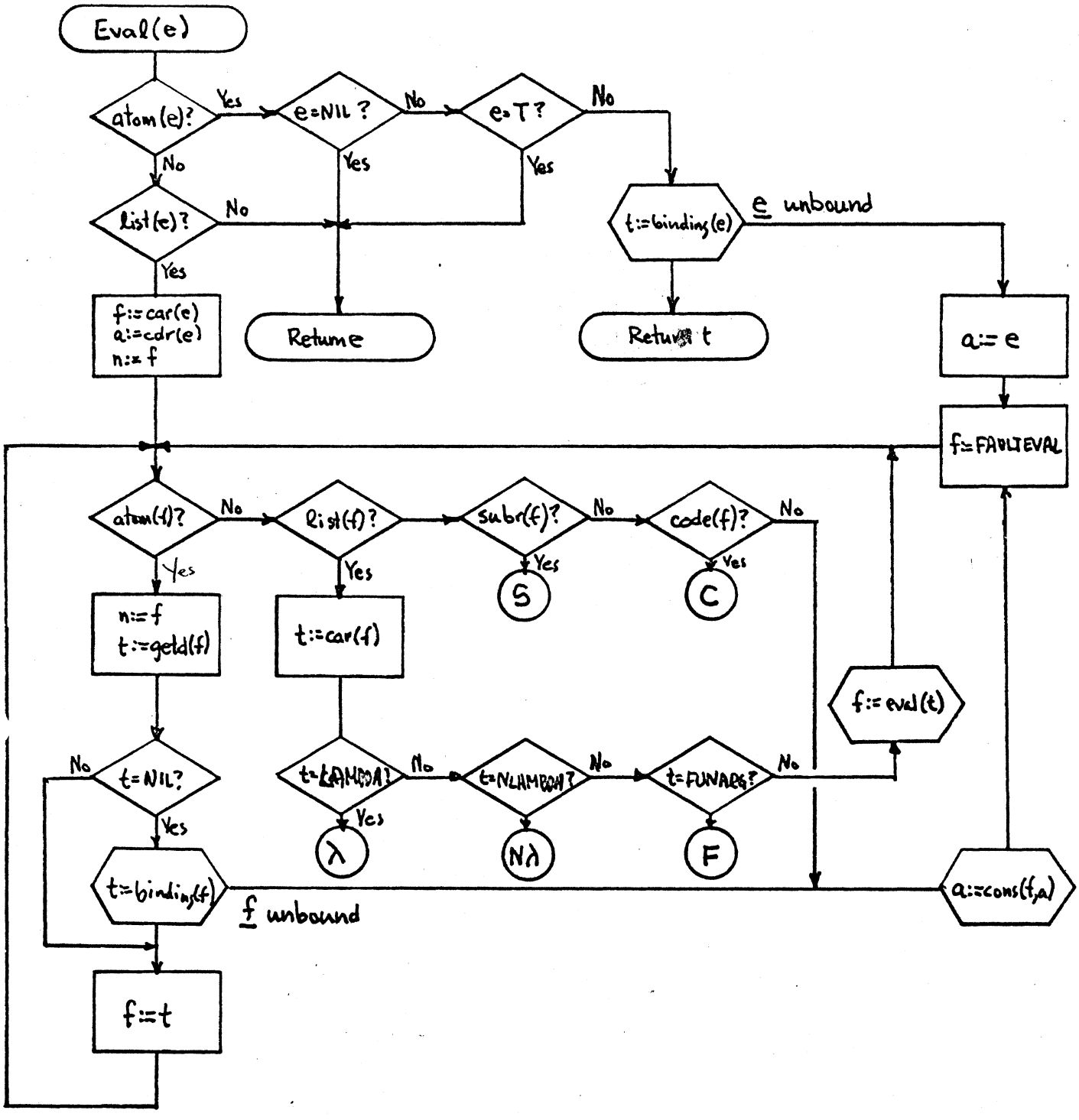
## Stack Entry Formats

Figure 8

The EVAL of CAL Lisp is very similar to BBN's; the flowchart of Figure 9 should be self-explanatory. One difference is that as the arguments for a function call are evaluated, they are bound to the successive integers 1,2,3,... . This is in contrast to the "lower case a,b,c,..." approach of BBN and allows the full ASCII character set to be used in atoms which are variables.

The subr has been defined to be one of the datatypes of CAL Lisp; a subr object is an entry in the subr table containing the expected number of arguments, whether or not the arguments are to be evaluated, and the address in system code of the actual subroutine. (See Figure 10.) The subr table is not part of writable virtual memory, although it is in the page table as must be all objects. This level of indirection allows the SYSIN/SYSOUT state-saving mechanism to be used across reassemblies of the system code, which may alter the subroutine addresses.

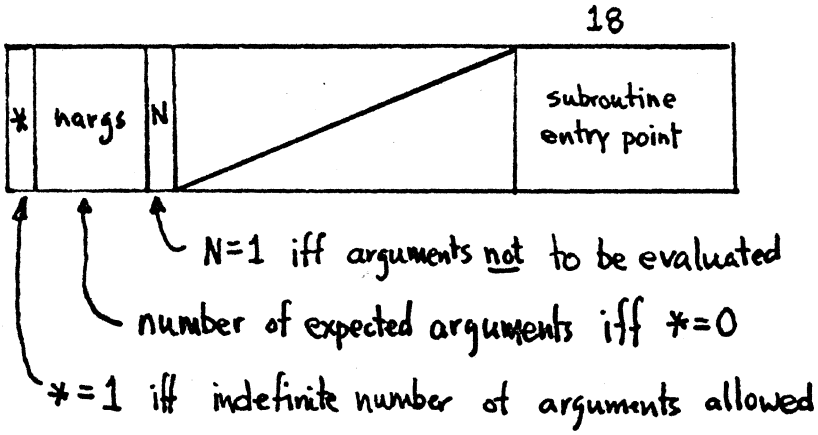
(LOGOUT)




# EVAL Mainloop

where is the rest of this?
(sections S, C, λ, Nλ, F)

Figure 9




Subr Table Entry Format

Figure 10

References

(1) Bobrow, D.G., Murphy, D.L., and Teitelman, W., The BBN Lisp System (Reference manual), Bolt Beranek and Newman Inc., Cambridge, 1969

(2) Teitelman et al., Memoranda, BBN, 1970-1971

(3) McCarthy, John, et al, Lisp 1.5 Programmer's Manual, M.I.T. Press, 1962

(4) Peter Deutsch