(ERIC NORMAN)
MADISON ACADEMIC COMP. CNTR.
1210 WEST DAYTON ST.
MADISON, WIS. 53706

This report describes the implementation of LISP on the Univac 1108. The primary purpose of this report is to show how certain of the more sticky problems were solved in this implementation. These, or similar techniques could be used in the implementation of other languages. Secondarily, this report will also serve as the maintenance document for 1108 LISP and should be read by anyone before he peeks into the system.

Familiarity with the 1108 is not a requirement to understand what is contained here. I have attempted to describe what is going on in machine independent concepts.

Familiarity with LISP is required. Before any one reads this, he should read the 1108 LISP Reference Manual, or at least have some idea of what LISP is all about.

The LISP system on the 1108 is essentially an interpreter that the user uses to evaluate expressions. When he needs more speed, he may use a compiler to generate machine code. The code generated by the compiler is placed directly in core and replaces the source language that was interpreted. There are also facilities for saving objects that have been constructed (like compiled functions) for use at a later date.

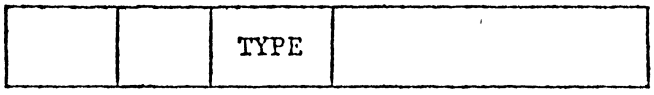
When originally pondering this implementation of LISP, I paid particular attention to two problems. First, I wanted a dynamic memory allocation scheme. There are many different types of objects in LISP (like numbers, lists, atomic symbols, compiled code, etc.). It is impossible to predict in advance how much space would be needed for these different types of information. I wanted a system that could allocate whatever space was needed and yet still be able to keep track of what is going on so that - say - garbage collections can be performed.

Secondly, I wanted a good and speedy lambda-calculus type evaluation scheme. The one normally used is terribly slow (spends too much time looking up things on property lists), and somewhat messy (in some contexts a symbol names a function while in others it might name something else).

In order to solve the memory allocation problem, I considered core chopped into pages. Each page is 128 words long (this can be easily changed). This means that words 0 through 0177_8 are on one page, as are words 017000_8 through 017177_8, and so forth. These pages extend all the way toward 0377777_8. So what, you ask.

Well, the big rul is that we can put any type of information on any page as long as we put only that type of information on that page. For instance, words 017000_8 through 017177_8 might be allocated for storing atomic symbols, words 017200_8 through 017377_8 might be used for integers and so forth. After the system runs a while, memory will be chopped up into pages of integers, pages of list structure; pages of compiled code, etc. In this way, the system adjusts to how much of each type of information is needed.

Now, the next problem is that given a pointer, we need to be able to tell what kind of information it points to. Therefore, a page table is maintained that contains an entry for each page. Each entry looks like:



Type is a code indicating the type of information contained on the associated page. E.g. word at PAGTAB + 0 describes the page from 0 to 0177_8, PAGTAB + 074_8 describes 017000_8 through 017177_8, etc. The page size is a power of two so that this association can be affected easily with a shift. That is, given a pointer, we find out what kind of fellow is being pointed at by shifting off the low order 7 bits and using this as an index into the page table.

The other fields of the page table are currently not used, but are reserved for possible future goodies. For instance, we might want to do a virtual memory scheme someday. Then we would need a field to indicate the presence or absence of the page in core as well as well as a field giving the actual address.

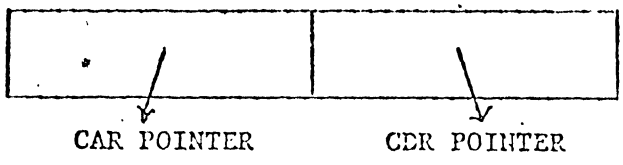
Notice that the available pages are all words from address 0 through 0377777_8. In reality, many of these are not really available; like only words 010000_8 through maybe 045177_8 might really be available for holding information. These are the only pages linked together in an available page list. Furthermore, notice that the largest possible address is 0377777_8 instead of 0777777_8. This is because a pointer in 1108 LISP can be either positive or negative. At one time, the big plan was to run this garbage collector in parallel with everything else. If this were done, the garbage collector could mark activity by complementing pointers and not hurt anybody because pointers are grabbed with a load-magnitude. Tricky, huh? too bad I never did it.

There are eight types of information in 1108 LISP and they are indicated by the following codes:

0 = list structure, i.e. consed nodes;
1 = integers;
2 = octals or print-name characters;
3 = floating point numbers;
4 = addresses out of bounds, i.e. address within system coding, pages not yet allocated, or pages not even part of the system;
5 = compiled code;
6 = linkage nodes (see below):
7 = atomic symbols.

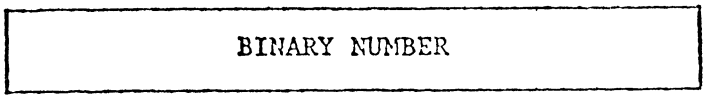
CONSED, type 0

1 word each, 128 per page.



Numbers, types 1, 2, 3

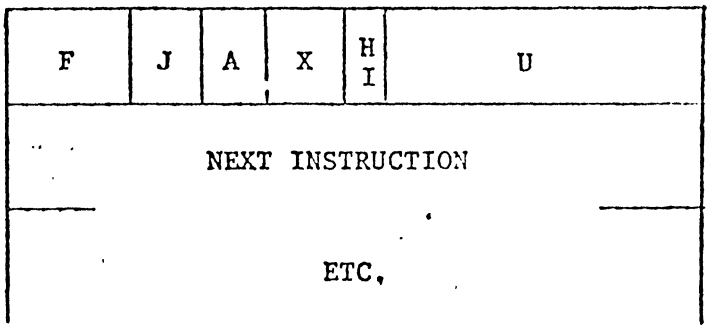
1 word each, 122 per page.



The reason that there are only 122 per page is that every 32nd word is set aside as a bit table to that the garbage collector can mark the 31 words following it.

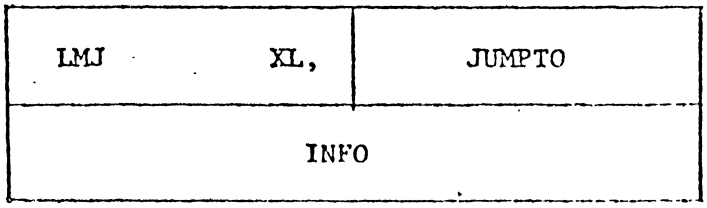
COMPILED CODE, type 5

1 word, 128 per page.



LINKAGE, type 6

2 words each, 64 per page.



The general reason for linkage nodes is that we need to allocate little tiny pieces of executable code. I will go into more detail later.

ATOMIC SYMBOLS, type 7

2 words each, 64 per page.

VALUE	PROPERTY LIST
HASH LINK	PRINT NAME

where:

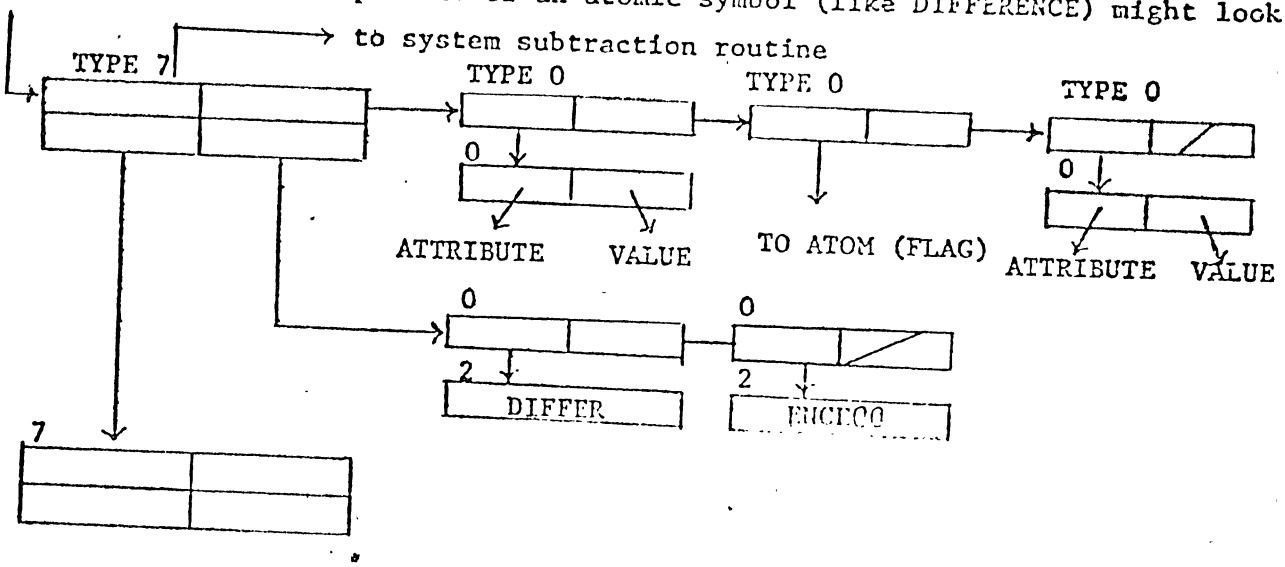
VALUE = pointer to value of this atomic symbol if it has a constant binding, 0 if it does not;

PROPERTY LIST = pointer to property list;

HASH LINK = pointer to next atomic symbol that has the same hash total (see chapter on input);

PRINT NAME = pointer to print name (a list of octals, each one containing six characters of the print-name in fieldata).

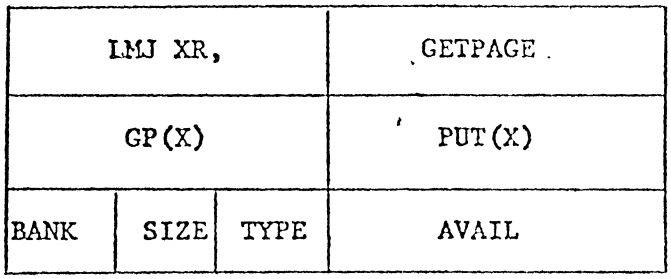
This means that the picture of an atomic symbol (like DIFFERENCE) might look like:



If we consider a given point in time, the 1108 LISP system actually contains many available space lists. There is an available space list for each type of information along with an available page list. Now really, there are two available page lists, one for the I-bank and one for the D-bank, but this is due to the 1108 hardware; let's pretend that there is only one.

The idea for allocating storage is that when we need a certain type of node, we extract one from the available space list for that type, or if the available space list for that type is empty, then we take an available page and build an appropriate available space list on it.

All this magic is effected like so: there is a table of 3 word items, one item for each of the 8 types. Each item looks like:



where:

GETPAGE = the page extraction routine (see below)
GP(x) = page initialization routine for this type.
PUT(x) = node storage routine for this type.
AVAIL = available space list for this type.
BANK = preferred bank, 1 for compiled code.
SIZE = size of node of this type.
TYPE = type, what did you expect?

Note to non-1108 folk. LMJ is the subroutine linkage instruction, i.e. put program counter into index register and jump.

In a virgin system, there is only an available page list; the available space list for each type is empty. When we want to create a node of a given type we load up appropriate registers with what is going to be put into the node and then do an LMJ XL, TYPTAB+3*type. This gets us to the first word of the item for this type which immediately links off to the page getting routine (GETPAGE above). This is a common routine that each type entry points to whenever there is no available space list for this type. This routine will: extract a page from the available page

list (a garbage collection happens if none are available), builds an available space list for this type on it using code pointed to by GP(x), points to it with AVAIL, puts the type in the page table, and finally changes the first word of the item so that it points to the normal storage routine. (i.e. it now says LMJ XR,PUT(x)). Now we can start over. Instead of linking to GETPAGE we enter the normal storage routine which will extract a node from the available space list, fill it in, and set a new available space list. But just before leaving it checks to see if the available space list for this type is now empty, and if it is, the first word is reset to point to GETPAG so that if we need another node of this type, another page is allocated.

One might ask if it is possible for an available space list for a type to reside on more than one page. Of course, but only after at least one garbage collection has happened. The garbage collector will create a new available space list for each type and these lists will be exhausted before any new pages are allocated for that type.

Garbage collection is a two phase process and is started whenever we need another page and none is available. First, all active structure is marked by a recursive subroutine. The base cells from which marking starts are: all cells on the value stack (see chapter on stacks); the hash table; and whatever pointers were to be stored in the node being created. After all active structure is marked, memory is swept and available space lists are built for each type of information.

Numbers (types 1, 2, and 3) are marked by setting a bit in a bit table which can be found by dividing the address of the number by 32. The quotient times 32 addresses the bit word and the remainder tells us which bit.

Nodes with pointers (types 0, 6, and 7) are marked by complementing an address contained therein, i.e. car pointer for consed nodes, jump-to address for linkage nodes, and hash link for atomic symbols. Then each pointer in such a node is marked recursively.

Compiled code (type 5) is marked by setting the lower half of the associated page table entry non-zero and then marking from the address field of each instruction. Note: those instructions that do not contain

a valid pointer will appear to point out of bounds so that we do not get hurt by assuming they are all addresses. Obviously, addresses out of bounds (type 4) are not marked.

Sweeping is accomplished by scanning through the page table. For each entry therein, a sweeping routine is entered for the type of information contained on that page. Each such routine will unmark all marked nodes and tie all unmarked nodes together in an available space list. When done, we check to see if there was anything active on the page. If there was, then the available space list for this page is added to the available space list for this type of information. But if the entire page is garbage, then we reset it to type 4 and add it to the available page list.

When done sweeping, we fix up the node storage table by plugging in the available space lists for each type and setting the jump addresses in the first words to the right thing. Finally, we check to see if the node being created can now be stored; i.e. it can if its type has a non-empty available space list or if a page is available. Obviously, if there is no room for the node, then we bring the world to an end.

A few general remarks are worth mentioning. First, notice that compiled code does get collected somewhat. That is, if enough compiled code becomes inaccessable that spans a page then the page is released and the code disappears.

Second, we notice that this method has a small tendency to pack memory without having to change pointers. That is, after we collect garbage, we are going to end up with an available space list for each type of information that is scattered across all pages containing this type, this available space list will be completely exhaused before a new page has to be obtained. This method is not fool proof though. For example, we could run a job that begins by allocating 200 nodes for integers but only 2 of them remain active, one in each page of integers created. If the run never needs to create another integer node, then we are wasting 128 words i.e. the page that would be released if one of the integers were moved to the other page.

Due to the way LISP operates, garbage tends to be released in large blocks and will therefore tend to span pages so that quite a few pages are usually released during a garbage collection.

So maybe you ask, "Why don't you compact memory so that everything is nice and tidy?" This seems like an unnecessary waste of time here. The normal reason for compacting memory during a garbage collection is that two things, like free storage and compiled code space, are growing toward each other. In 1108 LISP, I do not have this problem due to my superduper dynamic allocation scheme. It might be wise if I compacted memory as a last resort, but I just have not done it yet (there are minor problems about making sure what is and what isn't a pointer in compiled code).

Now to move on to bigger and better things, we still have the problem of evaluating LISP expressions. This means that we have to implement the lambda-calculus, well, we all know that LISP does not use the real lambda-calculus, but it comes reasonably close. Anyway, in essence the problem is binding arguments to which a function is being applied to the variables used in the definition of the function, correspondence of actual and formal parameters, if you wish.

Suppose we have a function defined by a lambda-expression say (LAMBDA(X Y) EXPR), and we apply it to some arguments, maybe by ((LAMBDA (X Y) EXPR)ARG1 ARG2). What this means is that we have to evaluate EXPR and during this evaluation something has to tell us that X is bound to ("means") ARG1 and Y is bound to ARG2. The technique I chose is the famous old association list. An association list is a list of consed pairs $(V_1 \cdot A_1)(V_2 \cdot A_2)\ldots)$ where the V's are variables and the A's are arguments. The current association list is always pointed to be a fixed register. In the above example during the evaluation of EXPR, the current association list starts out like ((Y.AEG2)(X.ARG1)...). Any time that we encounter X during the evaluation of EXPR, we will go peek into the current association list and discover that its value is ARG1.

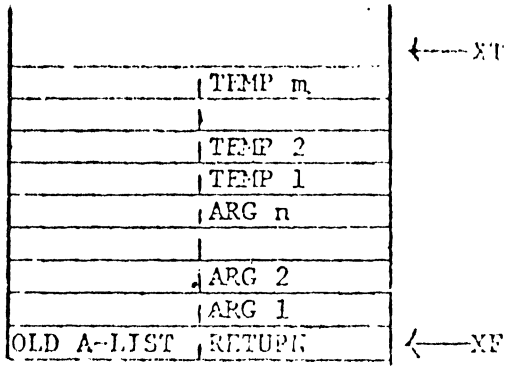
The only trouble with the association list is that it is not too efficient; one can spend quite a bit of time looking things up on it.

In some other implementations of LISP, the "special cell" trick is used. It is usually more efficient, but has another drawback, it just doesn't work! For instance, using special cells, it is just impossible to define a function that composes two other functions, i.e. COMPOSE=(LAMBDA (F1 F2) (FUNCTION(LAMBDA(X)(F1(F2 X))))), By using the association list, along with some other trickery, the above function can indeed be defined.

Furthermore, the inefficiency doesn't bother me at all. One of the design goals of 1108 LISP was to provide a compiler that could be used to augment the interpreter by translating interpreted code into machine code. One of the compiler's main job is to notice which variables do not have to be looked up on the association list and generate code to load their values directly.

The only other thing that is needed to realize an evaluation scheme for LISP is some sort of stacking mechanism to keep everything under control. In 1108 LISP there are two stacks. The primary one I call the value stack and always contains pointers, namely, arguments for functions, local variables, return addresses, and the like. The other stack is the control stack and contains non-pointers, usually indexes into the value stack.

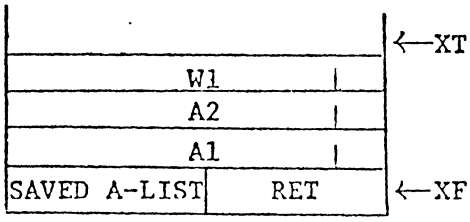
In 1108 LISP, a function is represented by a pointer to the code to execute in order to compute whatever result is to be computed. That is, the code for a function diddles the stack so that the result of the function is sitting on top of it. During execution of the code for a function, the value stack looks like this:



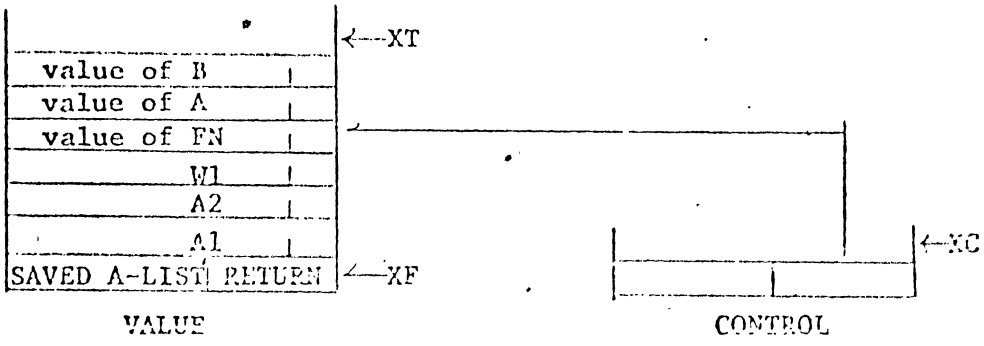
XF and XT are the index registers controlling the value stack; XT always points to the next available word and XF points to the current function base, i.e. the word where the result of the function will eventually be stored. During execution of the function, this word holds the return address and the old association list (the current one is pointed to by another register, XL).

The other words are the arguments of the function and any local variables that it might require. For instance, the code compiled for a function that was defined by (LAMBDA(ARG1 ARG2...ARGN)(PROG(TEMP1 TEMP2... TEMPM)...)) would assume that the stack looked as above.

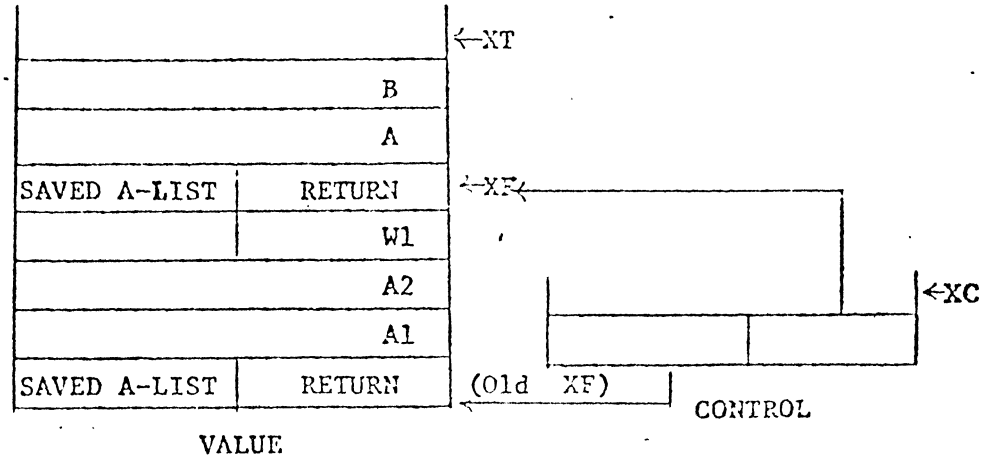
Now what we need to do is to be able to apply another function. The general rule in 1108 LISP for evaluating an expression like (FN A1 A2...AN) is to stack the value of everything, including the function. This means that when we finish stacking everything and want to apply the function, we have to know where it is in the stack. So the very first thing that we do before starting the evaluation is to mark the stack. This is done by putting a pointer to the top of the value stack (current value of XT) into the top of the control stack. Lets draw some pictures; lets suppose that the value stack looks like:



and we want to evaluate (FN A B) and leave its value on top of the value stack. We mark the stack and move the values of FN, A, and B onto it. Now it looks like:



Now, we link to the function application routine. This routine moves up the stack pointers. That is, XF is saved in the control stack, the address of the code representing the function is picked up (top of control stack points to word containing it), the return address and association list are saved in this word, XF is changed to the new base, and the function is jumped to. Now the stacks look like:



Now the stacks have been pushed up and we can execute the code for the function that was just entered. When we have the result computed, we leave it in a specified register and leap to the exit routine. The exit routine simply undoes everything and leaves the stacks just like they were before with the result pushed into the top of the value stack (it is also possible to simply leave it in its register without pushing it into the value stack).

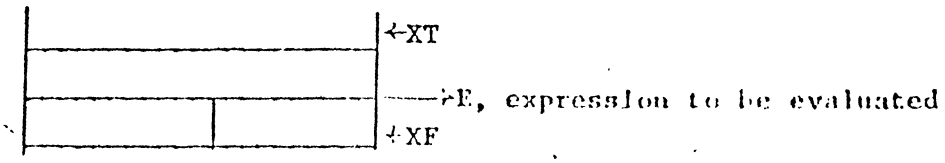
The most important thing to notice here is that the function to be applied was moved onto the stack in exactly the same manner as its arguments. That is, the function can be looked up on the association list, computed by some other expression, or retrieved as the constant value of an atomic symbol (the usual case). This gives an elegance to 1108 LISP that is missing in other systems. In other systems, one looks up the indicator EXPR for functions and APVAL for arguments. A symbol can mean two different things depending on where it is used - for shame!

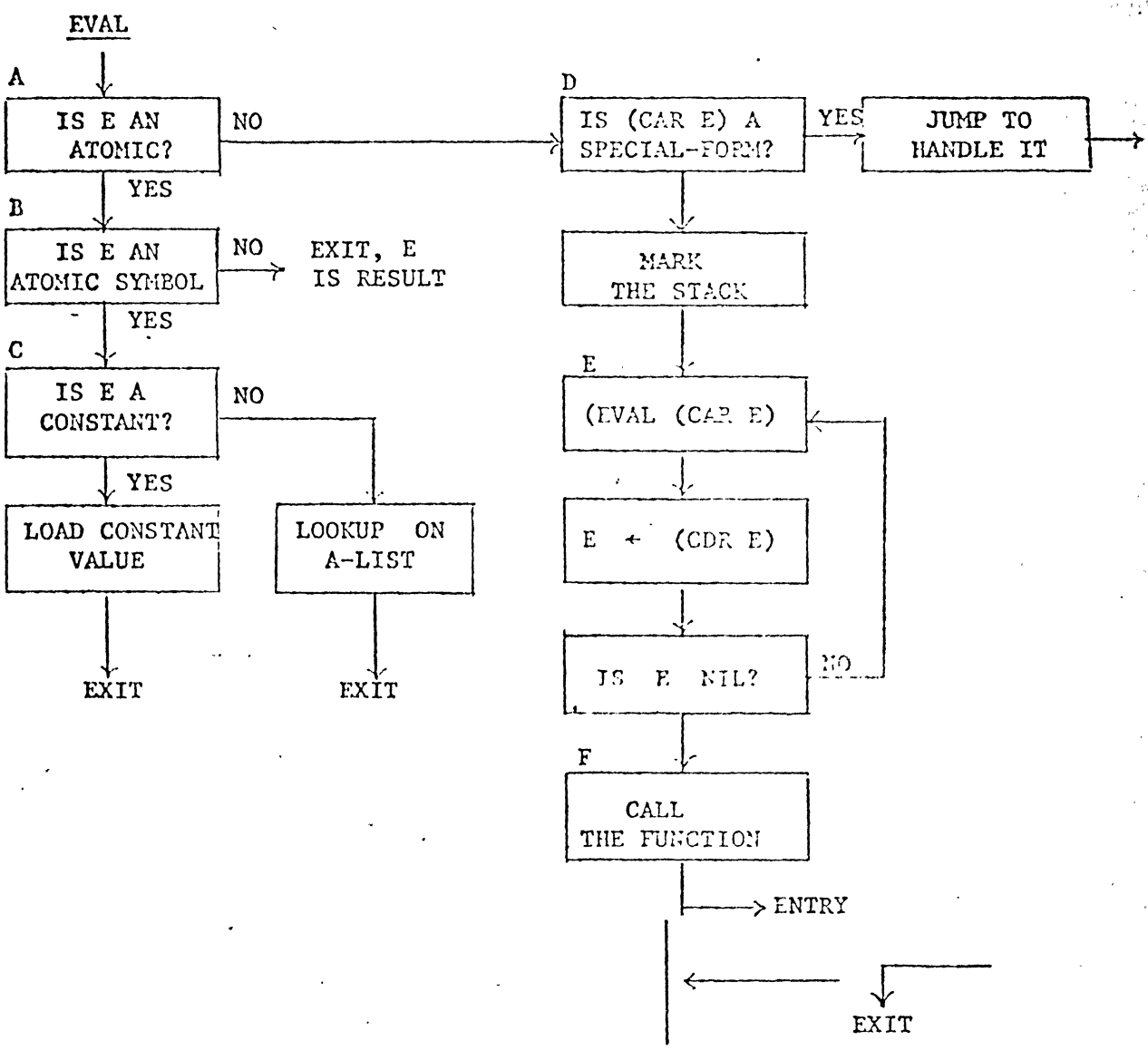
Then to make things worse sometimes the function is desired as argument and we have to fake the system out by saying (FUNCTION PLUS) or worse yet, (QUOTE PLUS), yechhh! In 1108 LISP, whenever you write PLUS, you mean that function that computes the sum of its arguments, that's all there is to it.

As a sidelight, we can say (FUNCTION FN) in 1108 LISP, but the purpose is simply to solve the free variable problem; it in no way acts as a 'quoting' mechanism. As a matter of fact, as long as I am editorializing, this shouldn't even be necessary; the purpose served by FUNCTION should be done automatically when we handle LAMBDA.

In 1108 LISP, I could have done this very easily. I didn't because I wanted to maintain some compatability with other systems. Now I wish that I had; I also wish I had changed CAR and CDR to something reasonable like FIRST and REST, O well, the world probably isn't ready for something quite so radical.

Lets proceed. The evaluation algorithm in 1108 LISP essentially looks at an expression and manipulates the stack in the indicated manner. We can describe this process very easily with a simple flow chart. Let E represent the expression to be evaluated. This means that we enter the evaluator with the stack looking like:





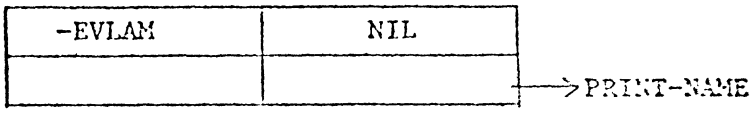
NOTES:

A - E is an atom if its type $\neq 0$.

B - E is an atomic symbol if type 7.

C - E is a constant if (CAR E) $\neq 0$;
(CAR E) is the value, see description of atomic symbols.

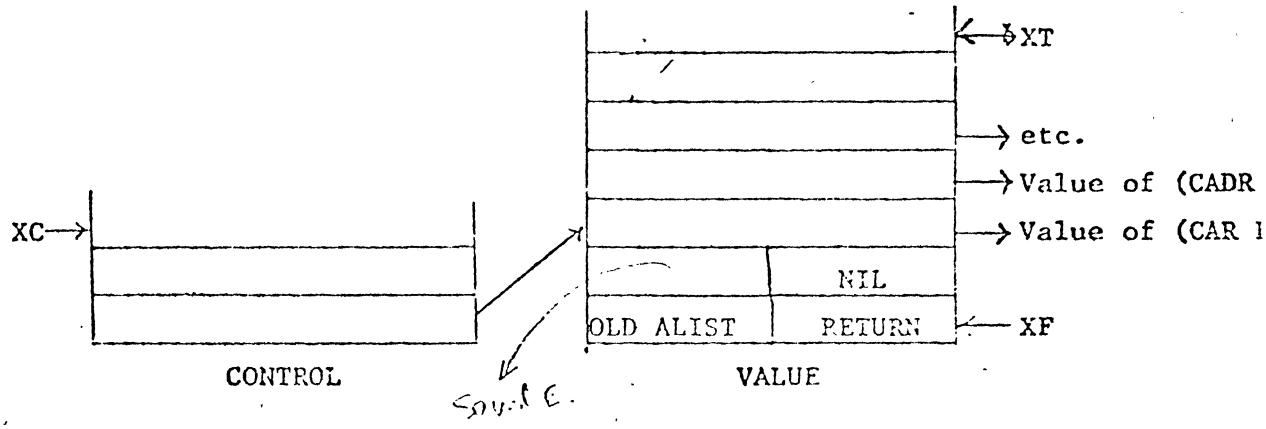
D - I.e. is an expression like (QUOTE x) or (COND (P E) ...), or (LAMBDA (V) E), etc. This is signaled by a negative pointer in the value part of the atomic symbol representing the special form. For instance, the atomic symbol LAMBDA looks like;



where EVLAM is the routine to do whatever we do for LAMBDA.

E - (EVAL (CAR E)) stacks the value of (CAR E), i.e. calls EVAL recursively.

F - when all done evaluating the stacks look like:



Lets look at an example, suppose our expression to evaluate is (PLUS X 3). This expression is not atomic and PLUS does not say special-form, so we are going to apply a function to arguments. We mark the stack and then evaluate each element of the list. So we come back in to evaluate PLUS; it is a constant and its value is the address of the accition function in the system. Now we have to evaluate X; we can assume that we find its value on the association list. Finally we evaluate 3, which evaluates to itself. Now that the function and all its arguments are evaluated, we leap into the function entry routine which pushes up the stack as mentioned above and jumps to the addition routine. The addition routine grabs its arguments from the stack, performs any necessary type conversions, adds them, stores the result in a new node in memory, exits, and we come back into EVAL and we are done.

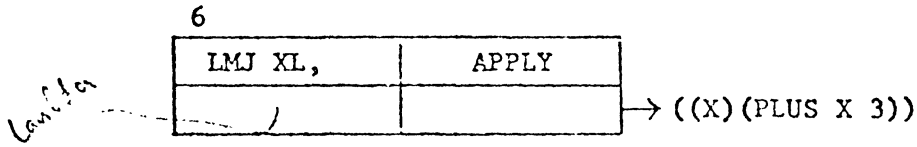
If, for example, our expression were (COND (P_1 E_1) ... (P_n E_n)), we would notice the negative pointer in the atomic symbol COND and scurry off to the routine to handle conditionals. This routine evaluates each predicate in turn until a true one is found, evaluates the associated expression, and exits.

Now, the only exciting thing that remains is to see how a user defined function gets interpreted. Suppose we define a function like so:

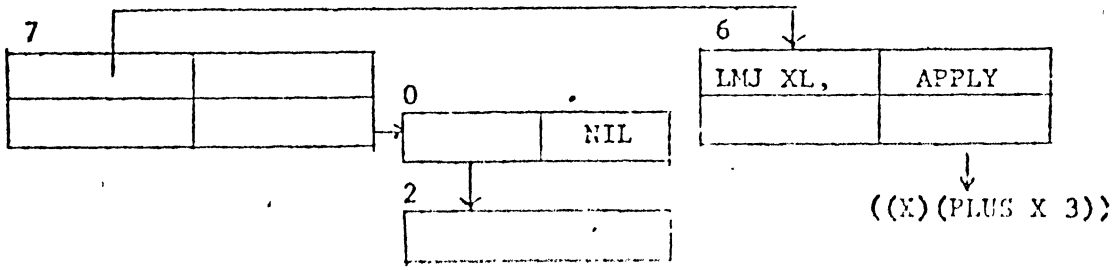
(CSET 'ADD3 (LAMBDA (X) (PLUS X 3))).

To evaluate this we evaluate CSET (system constant), evalute (QUOTE ADD3) (special-form), and finally we have to evaluate (LAMBDA(X) (PLUS X 3)). The value of this must be a function that adds three to its argument. In 1108 LISP, a function is represented by a pointer to executable code. This is the raison d'etre of the linkage node - type 6 above.

What we need here is a little piece of executable code that will get us back into the interpreter whenever the function being defined is applied. Ergo, evaluation of (LAMBDA(X)(PLUS X3)) creates a linkage node that looks like:



When we apply the pseudo-function CSET, this linkage node gets set as the constant value of ADD3, so that the atomic symbol ADD3 now looks like:



Now we can evaluate something like (ADD3 6). We move the value of ADD3 (linkage node above) and 6 into the stack and jump to the function, which jumps to the linkage node, which immediately links back into the system at the APPLY. The APPLY routine has available the definition of the function in the linkage node and the arguments in the stack. It extracts the variable list from the definition list and goes thru a little loop consing each variable to its corresponding argument and adding this pair to the current association list, when done it returns to EVAL giving it the body of the function, (PLUS X 3), and evaluates this in an environment where X means 6, i.e. association list starts with (X,6)..). This evaluation computes the result of applying ADD3 as was required.

The advantages of using the linkage node are twofold. First, it makes the interpreter lightning fast. That is whenever we apply a function, we do not have to go looking down the property list for some indicator like EXPR or SUBR. In 1108 LISP we just go merrily leaping into the code for the function. If the function is one supplied by the system, then we jump into the system. On the other hand, if it is a user function that needs to be interpreted, then there is a linkage node sitting there to route us back into APPLY to do the right thing.

The second beauty of the linkage node is that it solves the problem of linkage between compiled and interpreted code. Namely, the 1108 LISP compiler generates machine code directly into memory and puts the address of this code into the value part of the atomic symbol naming the function. Thereafter, application of the function will jump to compiled code (just like a system function) instead of a linkage node.