10/13/66

PRELIMINARY SPECIFICATIONS FOR BBN 940 LISP

Daniel G. Bobrow Daniel L. Murphy

I. Internal Storage

A. Pointers

There will be a maximum of 16 pointer types of objects in the 940 LISP System. These are (numbered in octal)

00. S-expressions (non-atomic)
01. Identifiers (literal atoms)
02. Small Integers
03. Boxed Large Integers
04. Boxed Floating Point Numbers
05. Compiled Function - Lambda Type
06. Compiled Function - Lambda Type - Indef Args
07. Compiled Function - Mu Type - Args Paired
10. Compiled Function - Mu Type - List of Args
11. Compiled Function - Macro
12. Array - Pointers
13. Array - Integers
14. Array - FP #'s
15. Strings - Packed Character Arrays
16.
17. Pushdown List Pointers

Each pointer will be contained in one 940 word of 24 bits. Bits 0 and 1 will be nominally empty, and may in some cases be used by the system (e.g. bit 0 for garbage collection) or perhaps even the user (in S-expressions). The four bits

2-5 will contain the type number for this pointer. The 18 bits 6-23 will contain an effective address (in the LISP drum file) where the referenced information is stored. The structure of each store is described below.

B. Allocation of Storage

Allocation of storage for each entity in the system will be made as it is required. Pages of 256 words will be allocated as necessary. Since the type information is carried along with each pointer, these blocks may be assigned anywhere in the 256K address, with no need to maintain any order to retain the contiguity of types of spaces. A map of the type assignment to pages will be kept for system use. Some compromise will be made, but is not yet specified, between garbage collection and addition of new storage when storage of any kind runs out.

There will be a number of varieties of garbage collection in the system. The first to be implemented will be a standard "stop the world, I want to collect" type. Then we will have a smart compacter using secondary storage. Finally we hope to implement an incremental garbage collector.

C. Identifiers (Literal Atoms)

An identifier has associated with it four canonical cells: 1) a val~~v~~ue cell 2) a property list cell 3) a function definition cell 4) a pname pointer cell. The position of each cell can be computed from the drum address given for the atom. These cells will not be consecutive. Value cells will be collected on separate pages of value cells, and similarly for the other 3 cells. The pointer to an atom will be a pointer to its value cell. As in our current system, the car of atom will yield the contents of the value cell, and cdr the property list. The functions getd and putd will read from and write in the function cell. A function getname will get the string which is the pname.

In a single user system, the position of the prlist, pname and function cells may be able to be computed arithmetically from the address of the value cell. In a multi-user LISP, where p-names are shared, there will probably be a double mapping for atom addresses.

D. S-expressions (list cells)

Each list cell is a consecutive pair of 24 bit words. The first word of each pair contains the car pointer, the second the cdr pointer. Since each pointer only takes up 22 bits, there are two bits (bit 1 in each word) to which the user will have access, ~~th~~ to mark, check and unmark. Bit 0 of each cell will be reserved for system use. The pointers may be of any of the 16 types. Thus one can have lists of pushdown pointers, arrays, functions, etc.

E. Numbers

Small integers between -2^{18} and $2^{18}-1$ will be represented directly by pointers. The 18 bit pointer will be the number - offset by a constant yet to be determined. Boxed large integers $1 x 1 \geq 2^{18}$ will be stored in single words in a page of such numbers. Boxed floating point integers will be stored in double words in standard 940 floating point format.

F. Function Types

There will be a compiler which will compile each of the five types of function in the system. The types of these functions can be recognized from the S-expressions, and therefore will not Need to be marked specially for the interpreter. The types must be marked for compiled code. We will describe these five types in Section II.

All compiled functions, of all types, are in approximately the following format shown in Figure 1.

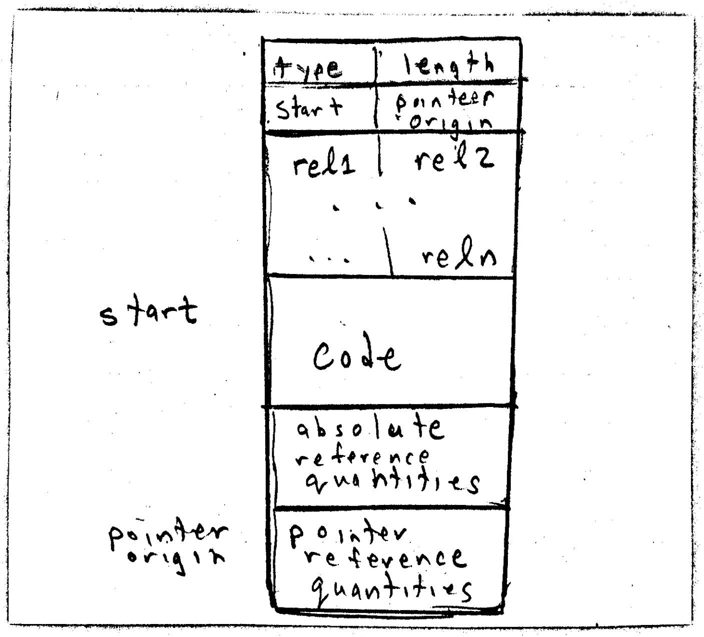

Figure 1: Format of a Compiled Function

The type is one of the five types of fcn.

Length = length of total block (must be $\leq 2^{12}$)

Start = start of code relative to beginning of block

Pointer origin = start of pointer literals referred to from compiled code rell,...,reln = locations relative to the start of the code which must be relocated (address adjusted) when loaded into core

G. Arrays

Arrays must be of uniform type, pointers, integers, or floating point numbers. Pointers may be of any type. Arrays addressed by a two word block which gives their type, length and starting position. Three new functions (Array Length type) will allocate space for an array and return a pointer to it; (ELT array ptr n) will give the value of the nth element of the array pointed to; and (SETA array ptr n value) will store a value in the nth element of the array. Note that arrays do not have names, but are structures which are pointed to like lists. SETA evaluates its arguments. Any contiguous subblock of an array may be an array also. CAR and CDR may be defined for array in the obvious way.

H. Strings

Strings are arrays of packed characters. Basic string functions have not yet been defined but it might be nice if CAR, CDR, RPLACA and RPLACD were extended in the obvious way.

I. Pushdown List Structure

In this system we expect to have four pushdown lists. They will contain 1) Pointers 2) Control Information for function calls, etc. 3) Unboxed Integers and 4) will be used for temporary storage of unboxed numbers for the arithmetic ~~f~~ routines, and for unboxed arguments of functions which are unboxed numbers. We will discuss that in detail later. These two PDL's may be combined to conserve core storage. Let us now consider only PDL 1 and 2.

Figure 2 shows the pushdown list near the beginning of a new page of each pushdown list after the function F has been entered with ~~two~~ three arguments X, ~~and~~ Y and Z; three cells of temporary storage have been used for computing arguments for the function G; and the function H has been entered, which has one argument W. The expression which is the body of H is about to be evaluated. i.e., the function F looks like

(LAMBDA (X Y) ... (G e_1 e_2 e_3 (H e_4))...

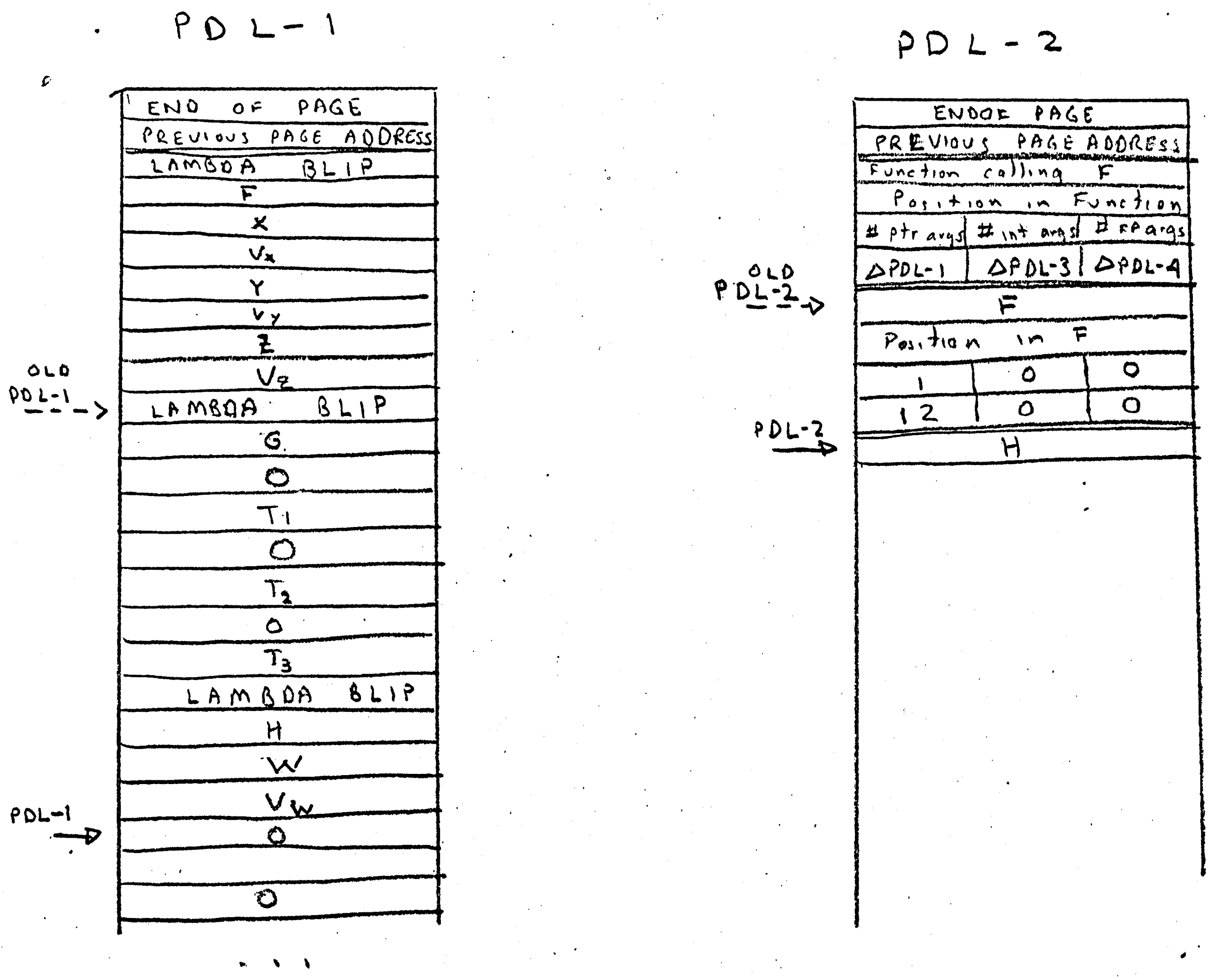

Figure 2 Push Down List Structure - About to evaluate expression in H in F

$F = (\text{LAMBDA } (x \; Y \; z) \; \ldots \; (G \; T_1 \; T_2 \; T_3 \; (H \; v_z))\ldots$

Four system cells contain the pointers to the current pushdown list positions. Compiled code knows the position of its arguments. The interpreter searches the stack for the appropriate variable name and gets the value from the ~~preceding cell~~ other cell in the pair. If it hits ~~a zero~~ the LAMBDA BLIP with a value which is the name of the current function it knows that it is looking for the value of a free variable. The stack is constructed so that there is a ~~zero~~ LAMBDA BLIP guaranteed to be ~~in the~~ at the top of any function area. ~~bottom cell of the stack, and~~ No variable bindings run across page boundaries. The latter is accomplished by moving the bindings to the top of a new page when a function is entered which has cross boundary bindings. The zeroes in the name position of PDL-1 for temporary results are guaranteed by initializing a stack block to zeroes in those positions, and having the function return reset them to zero upon exit (which it can do by scanning ~~down to the next zero~~ up to the LAMBDA BLIP from the current stack position).

J. Free and Special Variables

When a variable is determined to be a free variable (after a search down the PDL in the interpreter, or at compile time for compiled fens), the following action takes place. First, the contents of the value cell are obtained, If the system bit (bit 0) is on, then this variable has been bound SPECIAL and the value is that just found. We describe the syntax for binding ~~b~~ variables as SPECIAL's in Section II. If the system bit is off, then an upward search of the PDL is made to find the value. This is done each time such a variable is referenced in the interpreter. For compiled code a new pair is set up with the name of the free variable, and with value of PDL pointer to the original binding. All references to the value of this variable are ~~in~~ made indirectly through this pointer through the map.

K. The Funarg Device

Functional arguments will be passed ~~son~~ cons-ed with a PDL-1 pointer which preserves its context. Searches for free variables will begin from the point specified on PDL-1 instead of the current list. This skip of part of the PDL will be preserved on PDL-1 by a special BLIP followed by the

PDL pointer which will cause the search procedure to go back to the referenced portion of the stack. It may be that the ENDOFPAGE mark may work as that BLIP.

L. Unboxed Numbers as Args

Unboxed numbers on stacks 3 and 4 may be passed down as arguments to functions, and unboxed numbers may be returned as values. The LISP syntax for this is described in Section II. The names of these arguments are put on PDL-1 bound to PDL pointers to the appropriate places on PDL-3 and PDL-4. Thus these variables may be used free (but not bound SPECIAL).

II. Changes to the LISP Syntax and Semantics

A. Function Types

The function types in the new system are an expansion of the types in current LISP. They are separated into these types to give independence and flexibility in binding arguments to variables, having an indefinite number of arguments for a function, and having arguments evaluated or not

1) Lambda expressions - standard type

```
(LAMBDA (X  Y  Z) . . .)
```

This is the usual lambda expression which expects its arguments to be evaluated and bound to each variable name in the list of variables following the LAMBDA.

2) Lambda expressions with an indefinite number of arguments, e.g.

(LAMBDA N e_1 e_2 . . .)

These arguments are evaluated and put on the pushdown list with no names attached. The atom following the LAMBDA, in this case~~,~~ N is bound to the number of such arguments passed. The function (NTHARG m) will return the value of the nth argument provided $1 \leq m \leq N$.

3) Mu expressions - Args paired

This is one of two types of expression which provide unevaluated arguments

(MU (X Y Z) e_1 e_2 . . .)

The arguments of the above function will be bound (unevaluated) to X Y and Z, and e_1 etc. will be evaluated as usual.

4) ~~Mx~~ Mu expressions - args unpaired

(MU X e_1 e_2 . . .)

The list following function name will be bound to the atom X, with no evaluation.

5) Macro expressions

(MACRO X e_1 e_2 . . .)

Treated exactly as 4) i.e. (MU X . . .) except that the results of the computation are evaluated again. When compiling, macros are expanded at compile time.

B. Number of Arguments and Express

As with PDP-1 LISP, functions which LAMBDA expressions etc. may be given fewer arguments than expected, and the system will fill in NIL, ∅, or ∅, ∅ as appropriate, A LAMBDA etc. expression may be followed by any number of expressions and all will be evaluated - the value of the fen ~~with~~ will be the last evaluated. Conditional expressions have as elements lists of one or more expressions, and if the first is not NIL when evaluated, then all are evaluated in sequence and the value is the last evaluated. The same is done for SELECT and SELECTQ.

C. SPECIAL variables

Following the list of variables following a LAMBDA or PROG, one may insert a call to the pseudofunction SPECBIND. The list of variables following the SPECBIND which must appear

e.g. X and Y in (SPECBIND X Y)

in the current list of LAMBDA or PROG variables are considered to be bound SPECIAL. The current binding of these variables (X and Y in this case) are exchanged with the contents of the value cell, and the system bit in the value cell and in the variable names X and Y on PDL 1 are turned on. In a function return cleanup the bit in the name indicates that a swap back must take place. Since the old value and status of system bit are saved, the swap restores the old status exactly. SPECBIND must be used for each xxxxx occurrence (in each function) in which a variable (say X or Y) are to be bound as SPECIAL.

D. Numerical Arguments and Values

In this system, the onus will be put on the user to provide information about which arguments and/or values are unboxed numbers, and he must be sure he is correct or funny errors are liable to occur. The systemm may correct certain obvious errors, but nothing is guaranteed.

Functions may expect pointer integer or floating point arguments, but only in that order, i.e., you lmay have p pointer arguments and then m integers and n ~~flating~~ floating point numbers, but only in that order ($p \geq 0$ $m \geq 0$ $n \geq 0$). In the function definition you tell the function to expect these numerical arguments by following the list of LAMBDA variables by calls to the functions FIXV and FPV, e.g. if FOO is defined by (LAMBDA (X Y N U V) (FIXV N) (FPV U V). . . then FOO will expect two pointer args X and Y, one integer arg N and two floating point args U and V. After the list of PROG variable, calls to FIXVP and FPVP will create temporary variables of the appropriate type initialized to integer Ø ~~and~~ or Floating Point Ø.Ø. No conversions in arithmetic will be done on unboxed numbers unless explicitly requested.

In addition to telling a called function where to expect its arguments, you must in the cell of such a function tell the calling function where to put the arguments. The pseudo-functions ISFIX and ISFP tell a calling function to put arg on PDL-3 and PDL-4 respectively. Thus a call to FOO might look like

```
(FOO  A  B  (ISFIX T) (ISFP  B) (ISFP  S))
```