Draft Memo #9

DRAFT
S. L. KAMENY
30 September 1965

INTERNAL STORAGE CONVENTIONS FOR Q-32 LISP II

1. GENERAL ORGANIZATION OF CORE STORAGE

The general organization of core storage in LISP II is, as shown in Fig. 1, composed of seven different areas called Fixed Program Space, Character Atoms, Triples, Pushdown Stack, Binary Program Space, Array Space, and List Space. The pushdown pointer, PDP, kept in index register 8, together with a set of fluid variables, defined in section SYS, serves to define the current core map at any point. Each of the fluid variables is of type OCTAL and contains a core address whose meaning is shown in Fig. 1. The core address contained in each variable is just higher than the corresponding boundary of the core map. Thus, Fixed Program Space, which is built toward higher addresses, starts at location FPO and extends to the cell just lower than the location FPP. Similarly, the Pushdown Stack, which is built towards lower addresses, starts at the address just lower than location BPO, and extends approximately to the location PDP. The boundaries FPO, CHO and TRO are fixed in the system and cannot be changed except by reassembling the entire system. Garbage collection reclaims Triple Space, Array Space, and List Space, and repacks Array Space and List Space. Binary Program Space is also reclaimable and reusable.

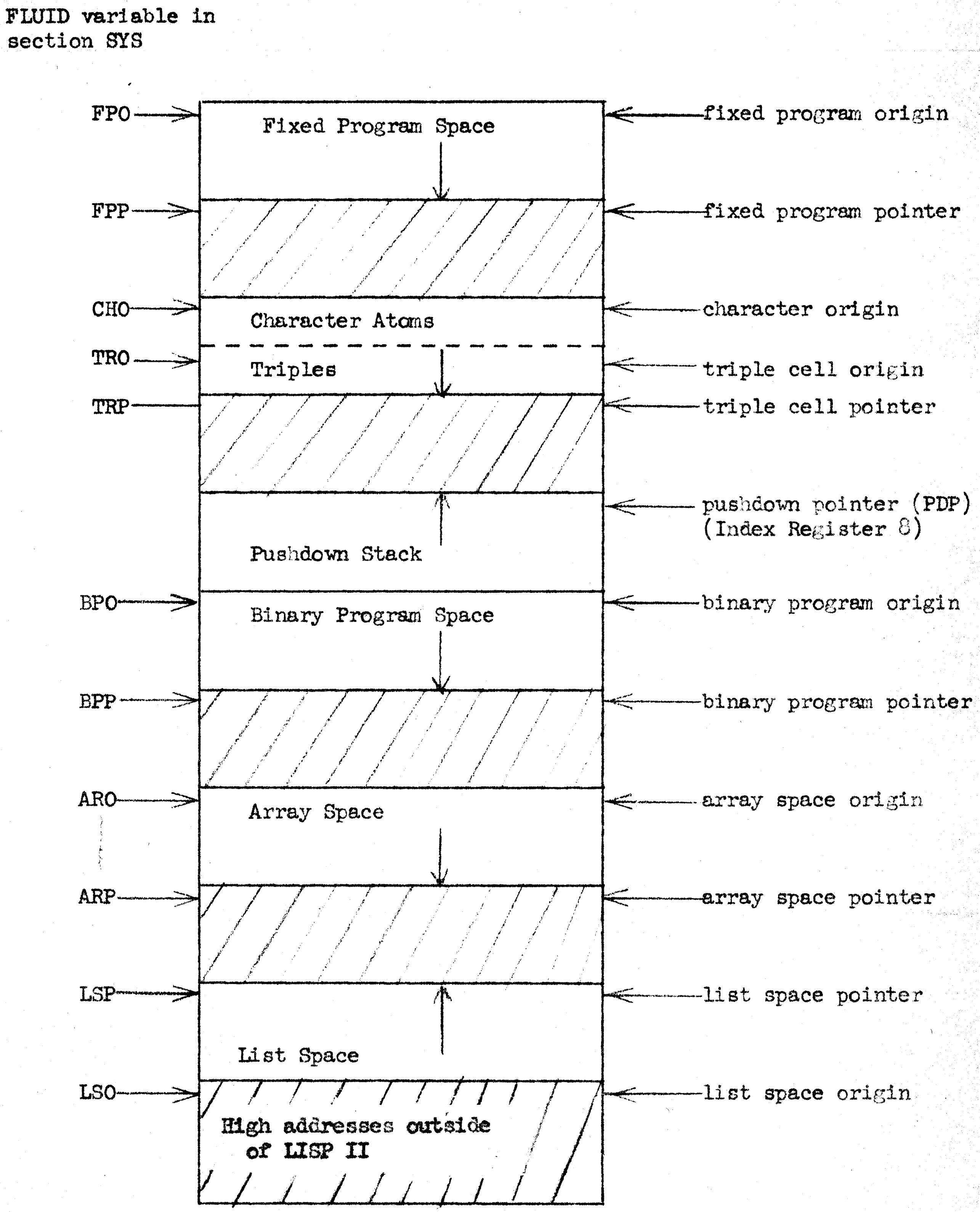


Fig. 1 LISP II Core Allocation

2. CORE AREAS

Fixed Program Space

Fixed Program Space, which starts at the fixed program origin (FPO) is used to hold fixed, non-relocatable constants and subroutines. Only those essential elements of the system which cannot be deleted should be put here, using LAP with an ORG pseudo-instruction (see LAP II memo for further details).

Character Identifier

Character identifiers, which start at the character origin (CHO) are all identifiers whose print names consist of a single character. The character whose print name has the ASCII code aa is located at aa + CHO. The structure of a character identifier is shown in Fig. 2.

Triple Space

Triple cell space, organized in groups of three cells, is used for the storage of identifiers, fluid cells, own cells (including function descriptors) and quote cells. Triple cell space starts at location TRO (immediately after the character identifiers). The triple cell pointer TRP points to the first higher address not occupied by triple cells. Within triple cell space, structures are never moved, but if a triple cell is abandoned it is converted to an empty triple cell and linked onto the (possibly empty) free-triple chain. A new triple cell is taken from the free-triple chain or if the free-triple chain is empty, the triple cell is placed at the end of triple cell space, and the boundary TRO is moved.

Ø7	v-f-list	t Tag	property list

t = 22 for alphabetic characters A-Z (identifier with standard spelling)

t = 32 for other characters (identifier with unusual spelling)

Fig. 2 Character Identifier

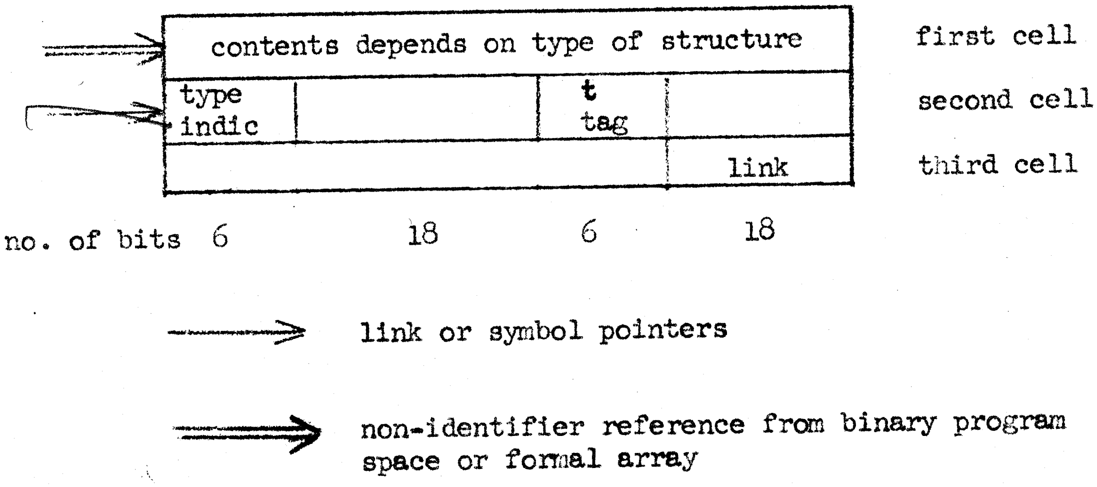

contents depends on type of structure
first cell
type indic
t tag
second cell
link
third cell
no. of bits 6
18
6
18
link or symbol pointers
non-identifier reference from binary program space or formal array

Fig. 3 Triple space structure-general

A triple space structure consists of three consecutive cells in memory, as shown in Fig. 3. Pointers in general go to the second cell of a triple. These pointers include the link pointer and symbol pointers. This is shown by the single arrow → in the figure.

References to non-identifier triples from binary program space, particularly direct or indirect load and store instructions and BUC indirect instructions use the address of the first cell, as shown by the double arrow ⟹ of the figure.

Triple cells other than quote cells are organized into a free-triple chain plus a series of "buckets." The free-triple chain is a chain of empty triples pointed to from variable TRL in section SYS. The chain is tied together through the link pointer of the third cell, with NIL in the last link. Each non-empty "bucket" is a chain of identifiers of which the first identifier is in (pointed to by) the OBLIST, a SYMBOL array in section SYS. The bucket is tied together through the link pointer.

no ¶

Each identifier is tied through the v-f-chain pointer in the left half of its second word to a variable-and-function chain composed of those fluid cells and own cells having it as a name. The v-f-chain is a circular list strung by means of the link pointer, with the last triple in the chain pointing back to the identifier. An identifier also contains a property-list pointer. The property-list is by convention a list containing any mixture of flags and property pairs. A flag is any atom, while a property pair is a dotted pair whose CAR is the property name (an atom) and whose CDR is the property value (any symbol).

The structure of buckets and v-f-chains is shown in Fig. 4.

Quote cells, also contained in triple cell space, have no linked structure.

Genids, or generated identifiers, are generated by the function GENID of no arguments in section NIL. Genids have the same structure as identifiers except that they have a genid indicator bit in the tag. They are strung in buckets from GENLIST, a SYMBOL array in section SYS, instead of OBLIST. Genids are produced by GENID with no p-name, i.e., the first word is all zero. The first time that a specific genid is printed out, it is supplied a name by the function GENPNAME in section SYS, and this name persists for the life of the genid.

Further detail on the contents of triple space is given in section 4.

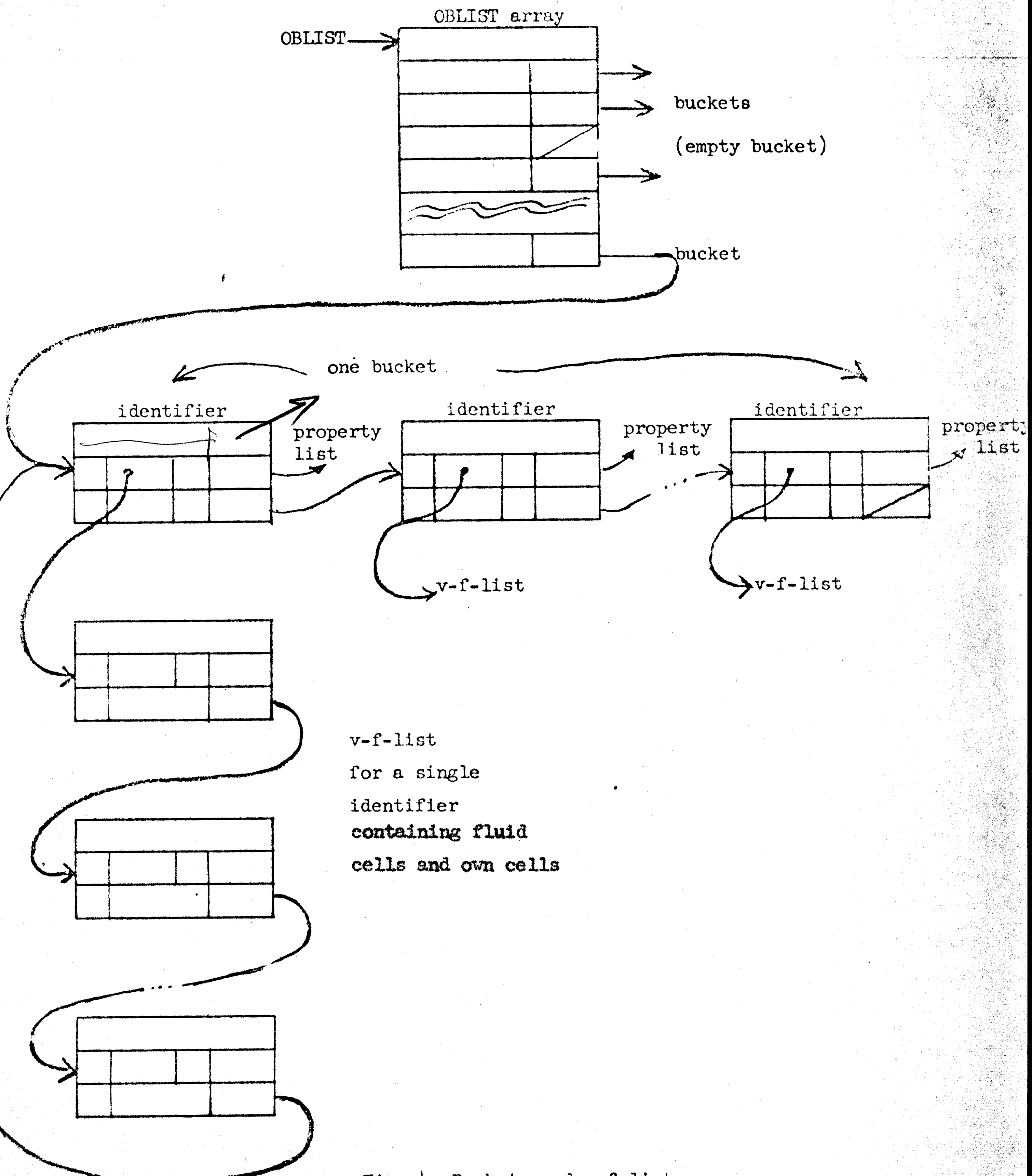


Fig. 4 Buckets and v-f-lists

Pushdown Stack

The pushdown stack and its organization are given in the LAP II document.

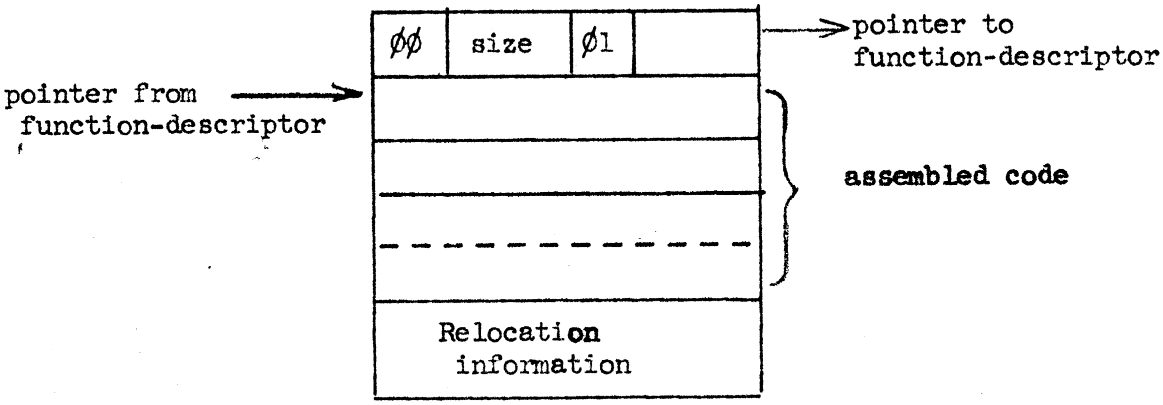
Binary Program Space

Binary program space consists of a packed series of assembled functions each of which consists of a header word, assembled code, and relocation information, as shown in Fig. 5.

The size contained in the header word is the total length of the assembled-function, including the relocation information, which is packed from left to right into a series of words starting with the last word and working toward the header word of the assembled function. The two bits refer to the left half and the right half of words of assembled code, and refer to the assembled code starting with the first word and extending to the cell immediately preceding the relocation information. The coding employed is the following:

ø means no relocation or count

1 means that if the address lies within this assembled-function it is a relocatable address. If the address does not lie within this assembled-function, then it points into triple cell space. The 1 in this bit means that the count in the triple is to be incremented when this assembled-function is loaded and decremented when the function is excised. A triple is excised if its count is zero and there are no symbol or locative references to it.


ØØ
size
Ø1
pointer to function-descriptor
pointer from function-descriptor
assembled code
Relocation information

size = total number of cells

Fig. 5 Assembled Function

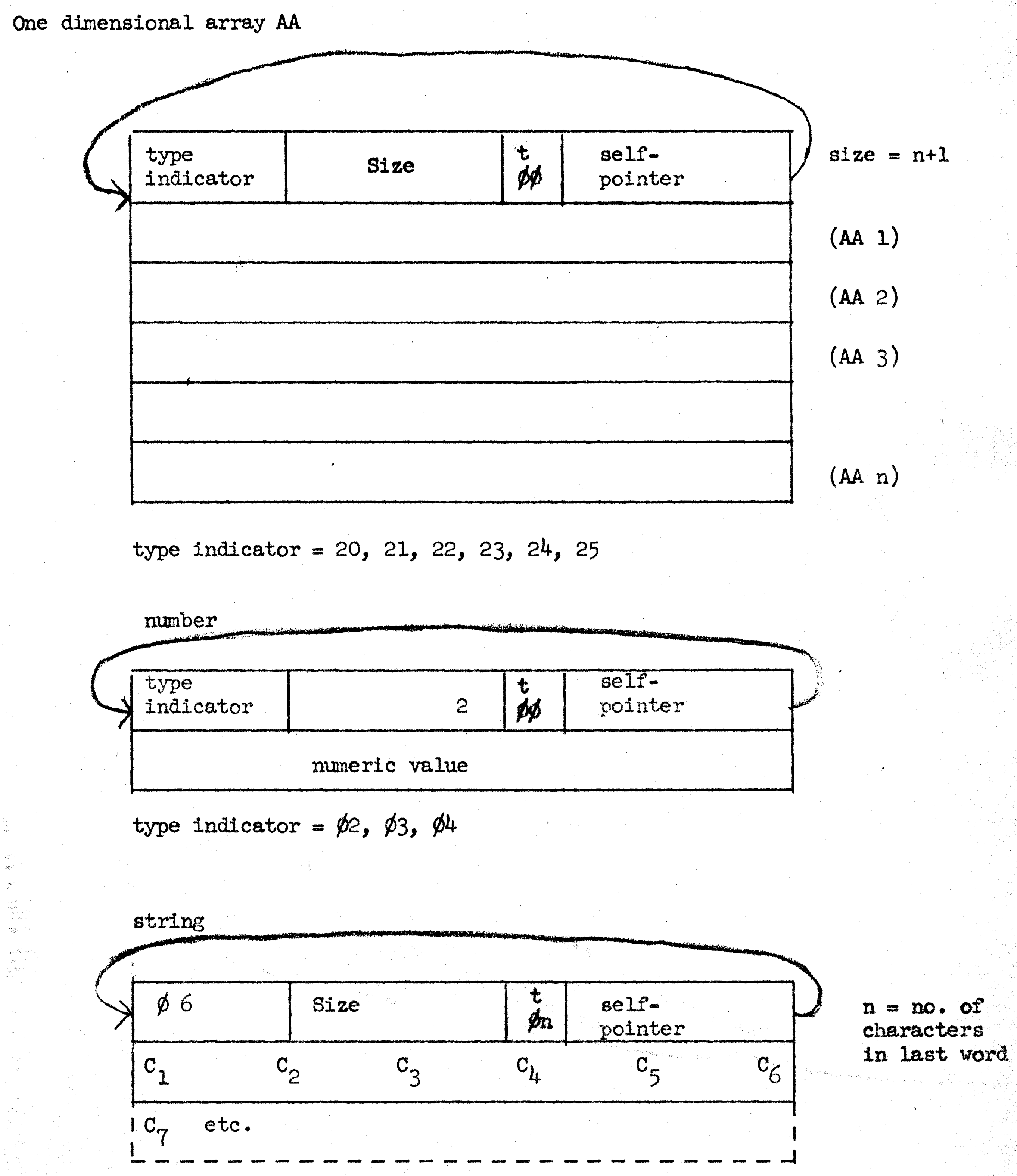
Array Space

Array space is used to hold arrays, numbers, and strings, of which the latter two are regarded as a special case of array. The three structures are shown in Fig. 6. One-dimensional arrays containing n elements have size = n + 1. The self-pointer, which is always contained in an array header word, is used by the garbage collector. Bit t_{24} of the tag portion of the header is used by the garbage collector for marking the array structure during garbage collection, and $t_{24} = \emptyset$ otherwise.

The meaning of the type indicator is given in section 3 below. In particular, the type indicator 2∅ means a symbol array whose elements consist of a single symbol in the address portion (the rightmost 18 bits), the rest of the word being zero. Each element of formal array, type indicator 25, is a pointer to the first word of a function-descriptor with its indirect bit set (tag of 2∅), so that the function can be operated by a branch indirect through the formal array element.

Strings contain six 8-bit ASCII character bytes per word, filled from the left end, as shown by c_1, c_2, c_3 ... in Fig. 6. Unused bytes are filled with the null-character $\emptyset\emptyset_{16}$. The tag portion of the array header shows the number of characters in the last word of the array.

The numerical value contained in a number is a real number for type indicator ∅4, an octal integer for type indicator ∅2 and a signed integer for type indicator ∅3.


One dimensional array AA

type indicator = 20, 21, 22, 23, 24, 25

number

type indicator = Ø2, Ø3, Ø4

string

Fig. 6 Array Space Structures

List Space

List Space consists of a series of list nodes, each of which contains two symbols corresponding to the car and cdr as shown in the following figure:

ØØ	symbol	t tag	symbol
	car		cdr

$t = Ø$ except for t_{24}, used by garbage collector, normally Ø.

A symbol is one of the following:

1. NIL represented by ØØØØØØ
2. TRUE represented by ØØØØØ1
3. An octal number q in the range ØQ ≤ q ≤ 177777Q, represented by q + 2Q5.
4. An integer n in the range -2Q5 ≤ n ≤ 177777Q represented by n + 6Q5
5. A pointer to a list node, an array head, or a character identifier
6. A pointer to an identifier, fluid cell, quote cell, or own cell in triple space. Symbol pointers always point to the second word of a triple.

3. TYPE INDICATORS

A type indicator is a six-bit code which occurs in two distinct contexts. First, it is used as an identification tag to distinguish types of storage structures. Given a legitimate pointer to data structure, one can determine the data structure type by looking at the type indicator. Second, type indicators are used to record type declaration information for variables and functions.

The current assignment of values of type indicators for these two uses is given in Table 1. Dashes indicate type indicator values which are currently unassigned.

In general, within type declaration information, the coding is as follows: the basic range ∅∅ - ∅5 is used for simple types. To each simple type, 1∅ is added to indicate LOC, 2∅ is added to indicate ARRAY, and 4∅ is added to show sub-specification. Since functions and FORMAL variables must be sub-specified, their declarations are described by a sequence of type indicators, as detailed in section 4.

Table 1. Type Indicators

Octal Value	Meaning as Structure Identifier	Meaning as Type Declaration
ØØ	list node	SYMBOL
Ø1	-	BOOLEAN
Ø2	octal	OCTAL
Ø3	integer	INTEGER
Ø4	real	REAL
Ø5	formal	FORMAL
Ø6	string	-
Ø7	identifier	-
1Ø	quote cell	(SYMBOL LOC)
11	fluid cell	(BOOLEAN LOC)
12	function descriptor	(OCTAL LOC)
13	empty triple	(INTEGER LOC)
14	-	(REAL LOC)
15	-	(FORMAL LOC)
16	-	-
17	-	-
2Ø	symbol array	(ARRAY SYMBOL)
21	boolean array	(ARRAY BOOLEAN)
22	octal array	(ARRAY OCTAL)
23	integer array	(ARRAY INTEGER)
24	real array	(ARRAY REAL)
25	formal array	(ARRAY FORMAL)
26	-	-
27	-	-
3Ø	-	(ARRAY SYMBOL LOC)
31	-	(ARRAY BOOLEAN LOC)
32	-	(ARRAY OCTAL LOC)

Octal Value	Meaning as Structure Identifier	Meaning as Type Declaration
33	-	(ARRAY INTEGER LOC)
34	-	(ARRAY REAL LOC)
35	-	(ARRAY REAL LOC)
36	-	-
37	-	NOVALUE or INDEF
40-44	unassigned	unassigned
45	-	FORMAL sub-specified
46-76	unassigned	unassigned
77	-	stop code

4. DETAILED USE OF TRIPLE-SPACE STRUCTURE

Triple space structures include empty triples, quote cells, identifiers, fluid cells, and own cells. Each triple cell structure is identified by the type indicator and the tag portion of its second word. The type indicator values of ∅7, 1∅, 11, 12 and 13 distinguish identifiers, quote cells, fluid cells, own cells, and empty triples, respectively.

The tag occupies bit positions 24 thru 29 in the word (counted from ∅ at the left end), and the tag-bits will be designated t_{24} through t_{29} respectively.

Of these bits, t_{24} is used by the garbage collector only, and is normally ∅. The remaining bits t_{25} through t_{29} are used in differing fashion depending upon the type indicator.

Empty triples

The structure of an empty triple as shown in the following figure, is empty (all ∅) except for the link portion of the third word and the type indicator 13 in the second word.

<table>
<tr><td colspan="4">∅</td></tr>
<tr><td>13</td><td>∅</td><td>∅∅</td><td>∅</td></tr>
<tr><td>∅∅</td><td>∅</td><td>∅∅</td><td>link</td></tr>
</table>

Quote Cell

A quote cell contains a single symbol datum in its first word, a type indicator of 1Ø and a count of Ø1 in its second word, and all zeros in the third word, as shown in the following figure.

<table>
<tr><td colspan="3">Ø</td><td>symbol</td></tr>
<tr><td>1Ø</td><td>Ø</td><td>ØØ</td><td>1</td></tr>
<tr><td>ØØ</td><td>Ø</td><td>ØØ</td><td>Ø</td></tr>
</table>

Identifier Triples

The structure of an identifier triple is shown in Fig. 7. It is a triple whose second word resembles a character identifier, except that t_{25} is Ø. Other bits of the tag are used to designate genids to indicate non-collectability, and to describe the relationship of the first word to the printname (pname) of the identifier. The third word contains a link used, as described in section 2, to chain the identifier buckets together. It also contains a count of the number of direct code references to this identifier. The identifier can be reclaimed by the garbage collector if, at any garbage collection, the count is zero, the property list is NIL, the v-f-chain is empty (self-pointer), the identifier is not pointed to from uncollectable list structure, and bit t_{28} of the tag of the second word is Ø.

If $t_{27} = Ø$, the pname of the identifier is contained in the first word, and the tag of the third word contains the number of characters in the pname. If $t_{27} = 1$, only the first three characters of the pname are in the first word, and the pname is a string pointed to by the first word, as shown in Fig. 8.

character identifier

Ø7	v-f-chain	t tag	property list

t_{24} used by garbage collector, normally Ø

t_{25} = 1 meaning character identifier

t_{26} = Ø for A-Z (identifiers with standard spelling)
1 for other characters (identifiers with unusual spelling)

t_{28} = 1 meaning never collectable by garbage collector

identifier triple

	print-name-information				first word
all pointers →	Ø7	v-f-chain	t tag	property list	second word
	ØØ	count	Øn	link	third word

t_{24} used by garbage collector, normally Ø

t_{25} = Ø meaning not character identifier

t_{26} = Ø for identifier with standard spelling
1 for unusual spelling

t_{27} = Ø pname in triple (no p-name array)
1 p-name array exists (pointer in third word)

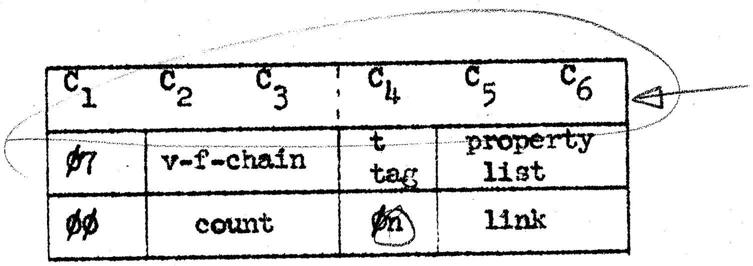
t_{28} = 1 if never collectable by garbage collector
Ø otherwise

t_{29} = 1 for genid (generated identifier)
Ø for normal identifier

Øn = number of characters in first word if t_{27} = Ø

Fig. 7 Identifier

identifier pname ≤ 6 characters



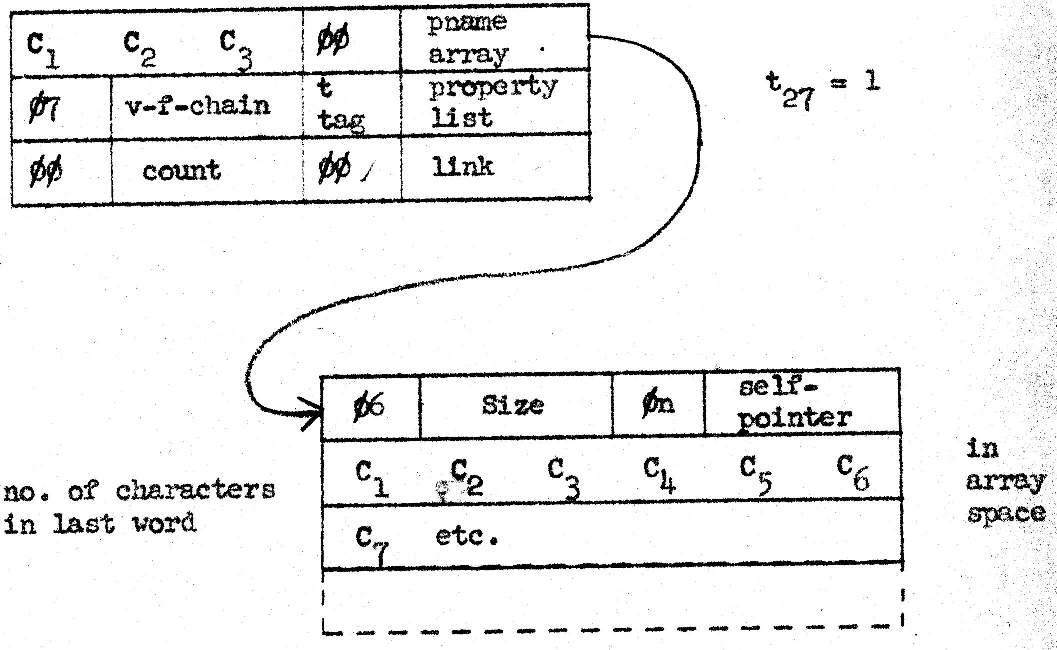
c_1	c_2 c_3	c_4	c_5 c_6
ø7	v-f-chain	t tag	property list
øø	count	øn	link

n = no. of characters in pname

$t_{27} = ø$

c_i for i > n are all ø's

identifier pname > 6 characters



c_1 c_2 c_3		øø	pname array
ø7	v-f-chain	t tag	property list
øø	count	øø	link

$t_{27} = 1$

ø6	Size	øn	self- pointer
c_1 c_2	c_3	c_4	c_5 c_6
c_7 etc.			

n = no. of characters
in last word

in
array
space

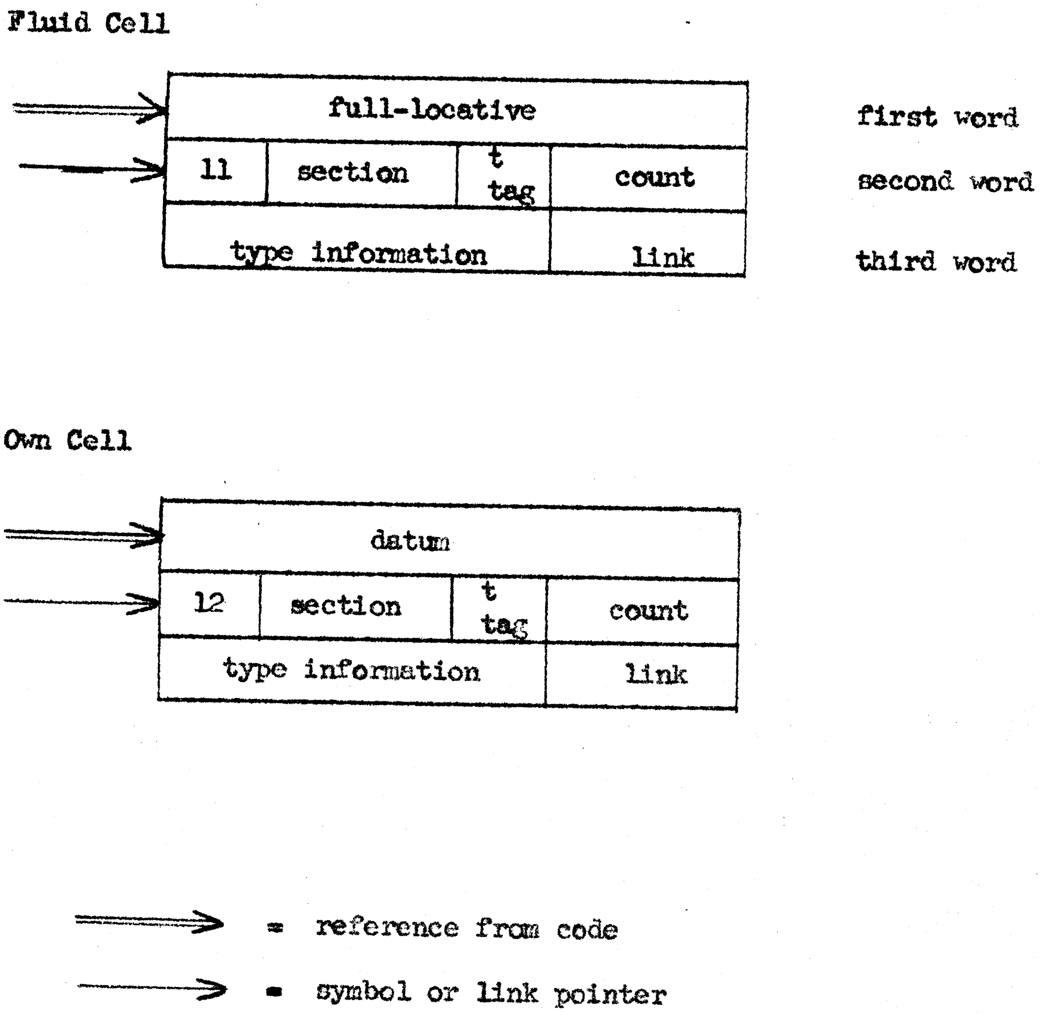
Fig. 8 Pname of identifiers

Fluid Cells and Own Cells

An identifier can have on its v-f-chain at most one fluid cell or own cell for any given section. Fluid cells and own cells are shown in Fig. 9. Fluid cells are used to hold fluid bindings of variables, while own cells are used to hold constant or own settings, particularly function descriptors. A fluid cell contains a full locative in its first word, and a structure indicator of 11 in its second word. An own cell contains its datum directly in its first word, and a structure indicator of 12 in its second word. The contents of the second and third words are similar for both kinds of triples.

The second word contains the structure indicator of 11 or 12, the section name (NIL or an identifier), and a count of the number of code references to this fluid or own cell. The tag bits are not used, except for t_{24}, which is used by the garbage collector, and t_{25}, used to designate a variable for which a top-level FLUID declarative exists.

The third word contains type information and a link.



Fluid Cell

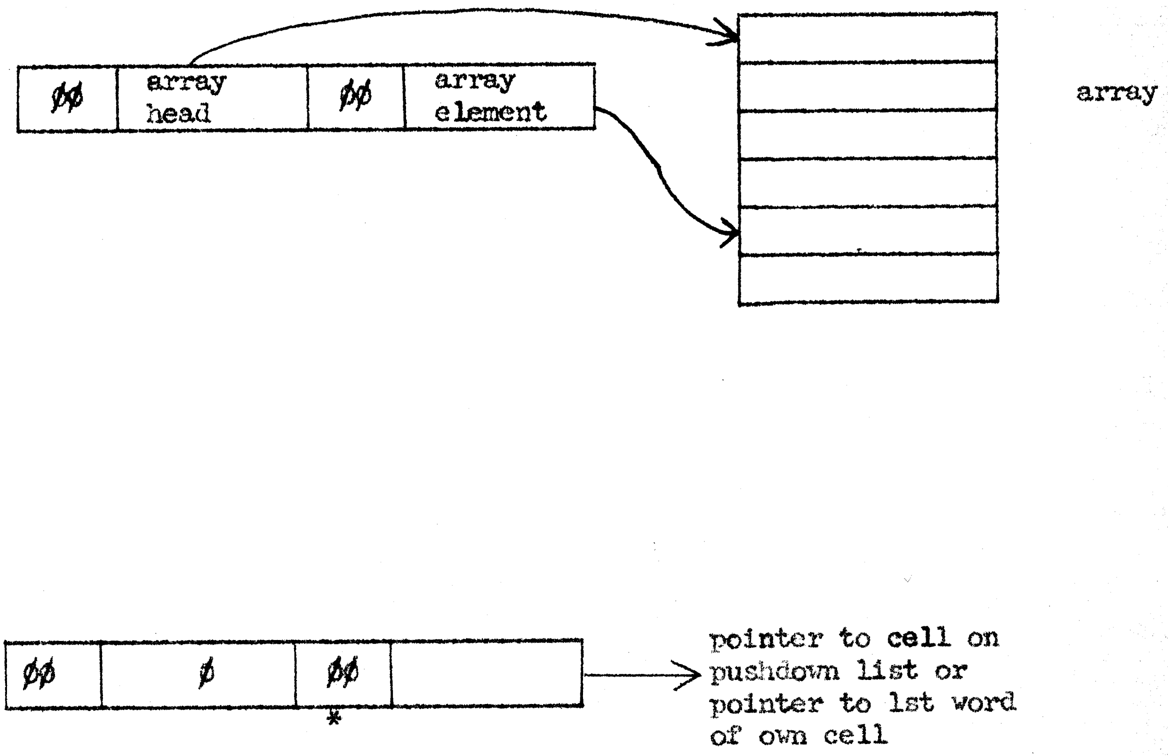
Own Cell

t_{24} used by garbage collector, normally ∅

t_{25} = 1 if FLUID declarative exists, ∅ otherwise

Fig. 9 Fluid Cell and Own Cell

The contents of a full-locative is shown in Fig. 10. A full-locative may point to the pushdown list, an array head, or an identifier, in which case it consists of a single pointer. Alternatively, a full-locative may point into an array, in which case it contains two pointers. In particular, the full locative contained in a fluid cell (of a variable whose transmission mode is not LOC) is initialized to point to the first element of a unique one-element array of the same type as the variable. This array is used to hold top-level free settings of the fluid variable.


ØØ
array head
ØØ
array element
array
ØØ
Ø
ØØ
*
pointer to cell on pushdown list or pointer to 1st word of own cell

* Formal locatives have indirect bit set (tag = 2Ø in this word)

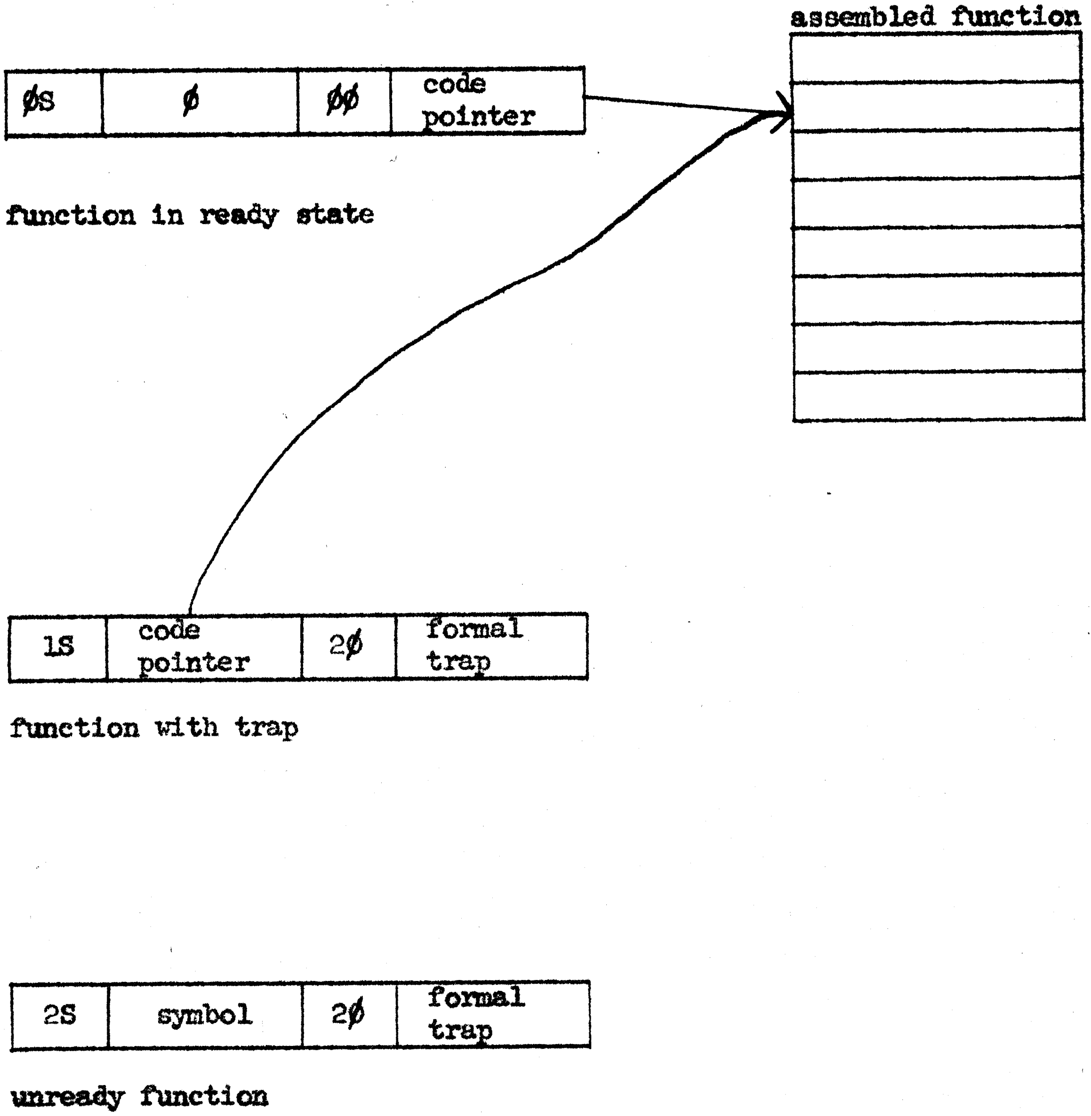
Fig. 10 Full-locative

Function Description

The function descriptor contained in the first word of an own cell can exist in one of three states, as shown in Fig. 11. A normal function in ready state, i.e., one that can be operated directly, contains a code pointer in the address portion of the function descriptor, a tag of Ø, decrement of Ø, and a prefix of Ø for a FUNCTION, 1 for a MACRO, or 2 for INSTRUCTIONS code.

A function with a formal trap, e.g., a function that is being traced, has the code pointer in its decrement, a formal trap in its address portion, and the indirect bit in the tag is set, so that transfers to this function descriptor will go indirectly through the formal trap code. The prefix of 1S indicates this condition, with S having the same meaning as for a ready function.

The third case of a function descriptor, for an unready function, has a prefix of 2S and a symbol in its decrement portion. The prefix 2S is used by the garbage collector to indicate that the left half of this word is to be marked during garbage collection. The symbol is used to hold information as to the location of symbolic code for this function, and the formal trap points to another function which is to be used to obtain or compile the unready function. S has the same meaning as for a ready function.



S = Ø for FUNCTION

    1 for MACRO

    2 for INSTRUCTIONS

Fig. 11 Function Descriptor (first word of own cell)

Type Encoding

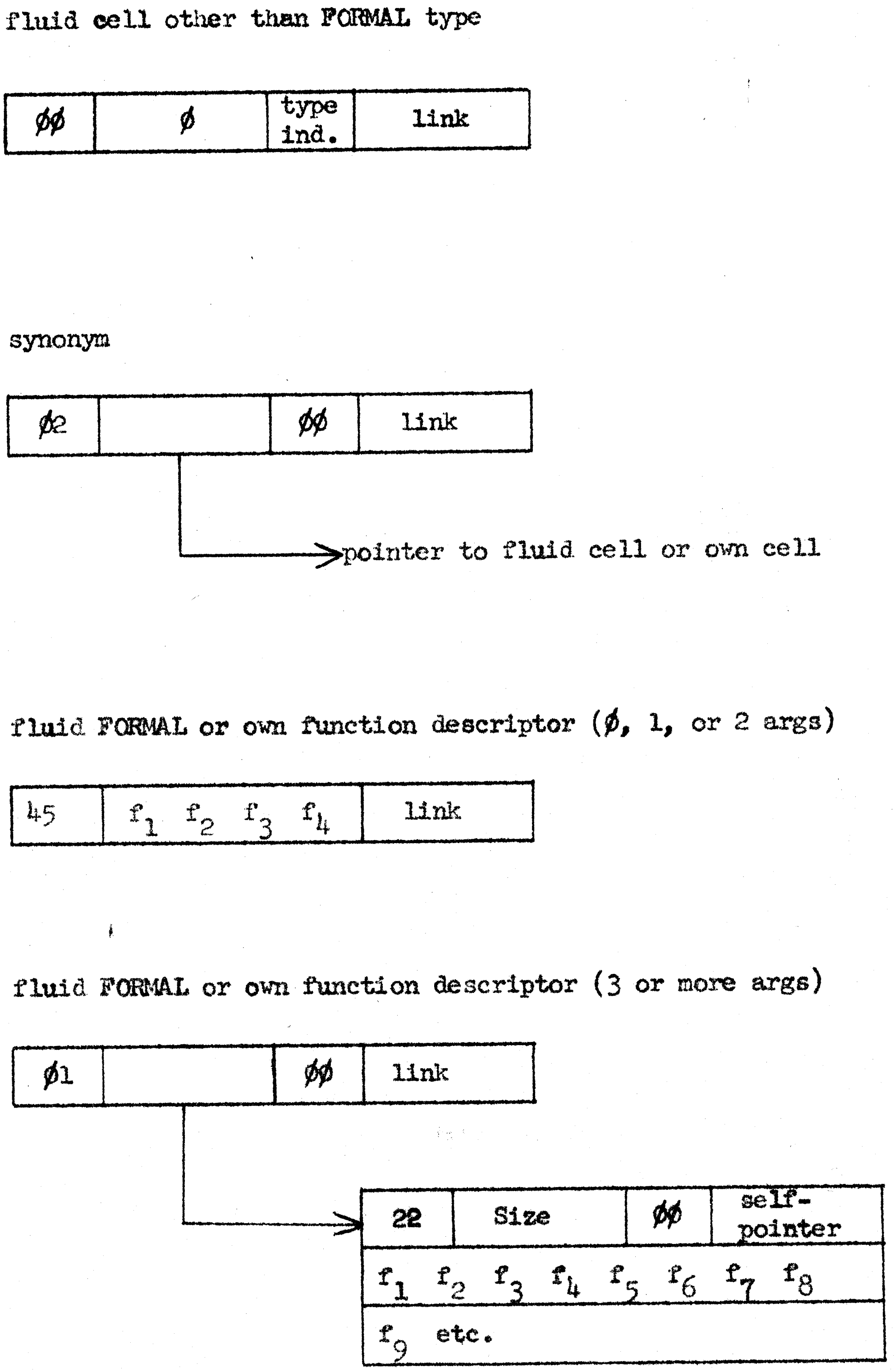
Type encoding is contained either directly in the third word of a fluid or own cell or indirectly in an array or triple cell pointed to by the type information, the various possibilities are distinguished by the value of the prefix, as shown in Fig. 12.

A prefix of ØØ indicates a fluid cell other than FORMAL where no sub-specification is required in the type information. In this case, a single type indicator, contained in the tag of the word, is used. The rest of the word is Ø except for the link.

The prefix Ø2 is used to indicate that the fluid cell or own cell is a synonym. In this case, the decrement of the third cell contains a pointer to another fluid cell or own cell, in which the type and value are to be found.

A fluid FORMAL, or an own cell used as a function descriptor, and used for a function of fewer than 3 arguments has its type information encoded directly in the third word. The prefix of 45 shows that a sub-specified FORMAL is to be represented. The coding f_1 f_2 f_3 f_4 is used to specify the type.

The prefix Ø1 is used for fluid FORMALS and own cells which require more than four type indicators to encode their type information. In this case, the decrement of the third word contains a pointer to an octal array (structure identifier = 22) which then contains the type coding.



fluid cell other than FORMAL type

synonym

fluid FORMAL or own function descriptor (ϕ, 1, or 2 args)

fluid FORMAL or own function descriptor (3 or more args)

Fig. 12 Type Information

The type coding of a formal or function is as follows:

f_1 specifies value-type

37 means NOVALUE

ØØ - Ø5 mean SYMBOL, BOOLEAN, OCTAL, INTEGER, REAL, FORMAL

Other values-types are not implemented at present.

f_2 specifies type of first argument

77 means no arguments

37 means INDEF with type given by f_3

ØØ - Ø5, 1Ø - 15, 2Ø - 25, 3Ø - 35 mean parameter type, according to Table 1.

f_3, f_4 ... may be

ØØ - Ø5, 1Ø - 15, 2Ø - 25, 3Ø - 35 which mean parameter types, according to Table 1.

The stop code 77 means that there are no more arguments. Hence, a function of n arguments requires n + 3 f's if the first argument is INDEF or n + 2 arguments if the first argument is not INDEF. However, the stop code is not required if the type information completely fills its allotted space. Hence, the third word of a triple can encode a formal or function containing up to 3 arguments (2 if INDEF), and an array of n cells can store type information for a function of 6n - 7 arguments (or 6 n - 8 if INDEF).