An Implementation of J

Roger K.W. Hui

J Version 4.1, 1992 1 27

33 Major Street
Toronto, Ontario
Canada M5S 2K9

Preface

J is a dialect of APL freely available on a wide variety of machines. It is the latest in the line of development known as "dictionary APL". The spelling scheme uses the ASCII alphabet. The underlying concepts, such as arrays, verbs, adverbs, and rank, are extensions and generalizations of ideas in APL\360. Anomalies have been removed. The result is at once simpler and more powerful than previous dialects.

This book describes an implementation of J in C. The reader is assumed to be familiar with J and C. J is specified by the *ISI Dictionary of J*, and introductions to the language are available in *An Introduction to J* and *Programming in J*; C is described in *The C Programming Language*.

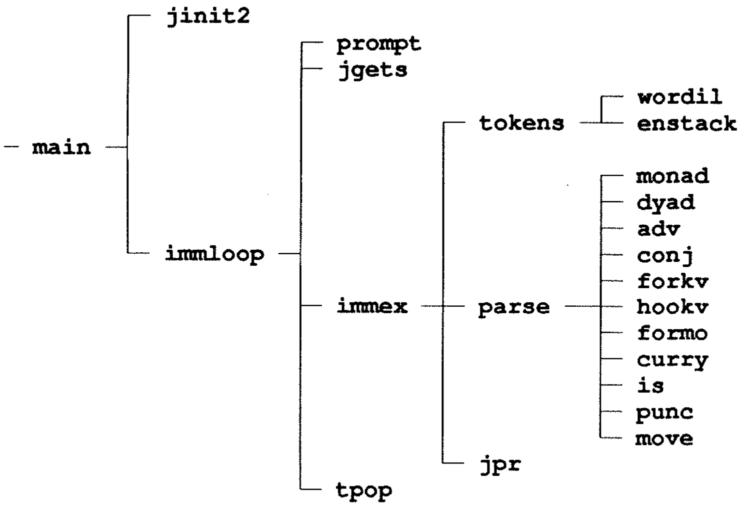
Why "J"? It is easy to type.

Acknowledgment

Ex ungue leonem.

Contents

0. Introduction



The system is organized as above. The main function `main` calls `jinit2` for initializations, then `immloop` ("immediate execution" loop), which repeats the following steps:

`prompt` and `jgets` prompt and accept an input sentence.

`immex` is the heart of the execution loop. The argument is a string of the input sentence. The processing is divided into three parts:

- `tokens` — word formation — applies the rhematic rules to partition the sentence into words. The result is a list of parts of speech: nouns, verbs, adverbs, conjunctions, copulae, and punctuation.
- `parse` interprets the sentence according to the parsing rules. Parsing is controlled by a table of (pattern,action) pairs; the eleven possible actions are embodied as the functions listed under `parse` in the diagram.
- `jpr` displays the result of the sentence.

Finally, `tpop` frees the temporary storage used in an iteration.

The fundamental data structure is the APL array (an object of data type **A**), used to represent all the possible objects in J. Most functions in the implementation accept arrays as argument and return them as result. Functions tend to be short and compact, and functions which implement J primitives are used freely. Extensive use is made of C preprocessor definitions and macros. Although the implementation language is C, the programming style is unmistakably APL.

The book is organized along the lines of the dictionary: Chapter 1 describes the interpretation of a sentence. Chapters 2, 3, and 4 describe nouns, verbs, and adverbs and conjunctions. Chapter 5 presents alternative representations. Chapter 6 describes display. Chapter 7 describes comparisons. Chapter 8, the final chapter, discusses each primitive in detail.

The remainder of the book of contains various useful bits. In particular, Appendix F (on the back cover) provides a means of quickly locating a primitive in the program files, and the Glossary has a short description on every non-local name in those files.

1. Interpreting a Sentence

1.1 Word Formation

Words are expressed in the standard ASCII alphabet. Primitive words are spelled with one or two letters; two letter words end with a period or a colon. The entire spelling scheme is shown on the back cover. The verb `;:` facilitates exploration of the rhematic rules. Thus:

```
   ;: 'sum =.+/_6.95*i.3 4'
```

sum	=.	+	/	_6.95	*	i.	3 4

The source code for word formation is in file w.c. The process is controlled by the function **`wordil`** (word index and length) and the table **`state`**. Rows of **`state`** correspond to 10 states; columns to 9 character classes. Each table entry is a (new state, function) pair. Starting at state `b`, a sentence is scanned from left to right one character at a time; the table entry corresponding to the current state and character class is applied.

NEW STATE/FUNCTION

b	?=	a=	N=	a=	9=	?=	?=	'=
b>	?>	a>	N>	a>	9>	?	?	'>
b>	?>	a	a	a	a	?	?	'>
b>	?>	a	a	B	a	?	?	'>
b>	?>	a	a	a	a	C	?	'>
z	z	z	z	z	z	?	?	z
b>	?>	9	9	9	9	9	?	'>
'	'	'	'	'	'	'	'	"
b>	?>	a>	N>	a>	9>	?>	?>	'
z	z	z	z	z	z	z	z	z
b	?	a	N	B	9	.	:	'

STATES

b	Blank
?	Other
a	Name
N	N
B	NB
C	NB.
9	Number
'	Quote
"	Adjacent Qtes
z	Comment

CLASSES

b	Blank
?	Other
a	Letters excl. NB
N	The letter N
B	The letter B
9	Digits and _
.	Period
:	Colon
'	Quote

FUNCTION

>	`j=.i [ Emit(j,i-1)`
=	`j=.i`

`Emit(j,i-1)` produces a pair of indices delimiting a word in the string. `i` is the current index, and `j` is an internal register; if the current word is a number immediately following a numeric list (one or more numbers), `Emit` combines their indices to form a single word. At the end of the string, `Emit(j,i-1)` is executed.

This process is applied to `sum =.+/_6.95*i.3 4`, the sentence used above to illustrate word formation. In the following table, the columns are: index, character in the string, the (current state, character class) pair, the (new state, function code) pair, and the action. For example, the first step is step 0, the letter is `s`, the current (and initial) state is `b`, and the character class is `a`. From the table, the entry in row `b` and column `a` is `a=`, meaning the new state is `a` and the function code is `=`. The action assigns 0 to `j`.

i	CHAR	STATE/ CHAR CLASS	NEW STATE/ FUNCTION	ACTION
0	s	b a	a=	j=.0
1	u	a a	a	
2	m	a a	a	
3		a b	b>	j=.3 [Emit(0,2)
4	=	b ?	?=	j=.4
5	.	? .	?	
6	+	? ?	?>	j=.6 [Emit(4,5)
7	/	? ?	?>	j=.7 [Emit(6,6)
8	_	? 9	9>	j=.8 [Emit(7,7)
9	6	9 9	9	
10	.	9 .	9	
11	9	9 9	9	
12	5	9 9	9	
13	*	9 ?	?>	j=.13 [Emit(8,12)
14	i	? a	a>	j=.14 [Emit(13,13)
15	.	a .	?	
16	3	? 9	9>	j=.16 [Emit(14,15)
17		9 b	b>	j=.17 [Emit(16,16)
18	4	b 9	9=	j=.18
19				Emit(18,18)

Every primitive word has an ID (a unique byte value) defined in file jc.h. The ID for the first 128 ASCII characters are simply the byte values 0 to 127; other IDs are arbitrary assignments in "dictionary order".

```
  ...
#define CLPAR    '('      /*  40 050 28      */
#define CRPAR    ')'      /*  41 051 29      */
#define CSTAR    '*'      /*  42 052 2a      */
#define CPLUS    '+'      /*  43 053 2b      */
  ...
#define CASGN    '\200'   /* 128 200 80 =.   */
#define CGASGN   '\201'   /* 129 201 81 =:   */
#define CFLOOR   '\202'   /* 130 202 82 <.   */
#define CMIN     '\202'   /* 130 202 82 <.   */
#define CLE      '\203'   /* 131 203 83 <:   */
#define CCEIL    '\204'   /* 132 204 84 >.   */
#define CMAX     '\204'   /* 132 204 84 >.   */
#define CGE      '\205'   /* 133 205 85 >:   */
  ...
```

Using mnemonics such as `CPLUS` and `CASGN` instead of `'+'` and `'\200'` makes the source code more readable and more amenable to automatic manipulation.

The 3-row table `spell` associates letter sequences with IDs. The rows correspond to letters in the range ASCII 32 to 127, those letters inflected by a period, and those letters inflected by a colon; table entries are IDs. Thus:

```
static C spell[3][47]={
 '=',    '<',    '>',    '_',    '+',   '*',    ...,
 CASGN,  CFLOOR, CCEIL,  1,      COR,   CAND,   ...,
 CGASGN, CLE,    CGE,    CUSCO,  CNOR,  CNAND,  ...,
}
```

The first column specifies that `=.` has the ID `CASGN` (assignment) and `=:` the ID `CGASGN` (global assignment).

`spell` is used by functions `spellin` and `spellout`: given a string (*e.g.* `"=."`), `spellin` computes the ID (CASGN); given the ID, `spellout` computes the corresponding string. `spellin` also uses the table `nu`, which contains alternative spellings for the "national use" characters.

Using the information computed by `wordil`, functions `tokens` and `enstack` transform a string into a list of nouns, verbs, adverbs, conjunctions, etc. The next step is to parse this "tokenized" form of the sentence.

1.2 Parsing

Parsing occurs after word formation and is controlled by function **parse** and table **cases** in file p.c. **cases** is a direct translation of the parse table in Section II E of the dictionary:

```
#define EDGE     (MARK+ASGN+LPAR)
#define NOTCONJ  (NOUN+VERB+ADV)
#define RHS      (NOUN+VERB+ADV+CONJ)

PT cases[] = {
 EDGE,          VERB,       NOUN,       ANY,        monad,vmonad,cmonad, 1, 2,
 EDGE+NOTCONJ,  VERB,       VERB,       NOUN,       monad,vmonad,cmonad, 2, 3,
 EDGE+NOTCONJ,  NOUN,       VERB,       NOUN,       dyad, vdyad, cdyad,  1, 3,
 EDGE+NOTCONJ,  NOUN+VERB,  ADV,        ANY,        adv,  vadv,  cadv,   1, 2,
 EDGE+NOTCONJ,  NOUN+VERB,  CONJ,       NOUN+VERB,  conj, vconj, cconj,  1, 3,

 EDGE+NOTCONJ,  VERB,       VERB,       VERB,       forkv,vforkv,cforkv, 1, 3,
 EDGE,          VERB,       VERB,       ANY,        hookv,vhookv,chookv, 1, 2,
 EDGE,          ADV+CONJ,   ADV+CONJ,   ADV+CONJ,   formo,vformo,cformo, 1, 3,
 EDGE,          ADV+CONJ,   ADV+CONJ,   ANY,        formo,vformo,cformo, 1, 2,
 EDGE,          CONJ,       NOUN+VERB,  ANY,        curry,vcurry,ccurry, 1, 2,

 EDGE,          NOUN+VERB,  CONJ,       ANY,        curry,vcurry,ccurry, 1, 2,
 NAME+NOUN,     ASGN,       RHS,        ANY,        is,   vis,   vis,    0, 2,
 LPAR,          RHS,        RPAR,       ANY,        punc, vpunc, vpunc,  0, 2,
};
```

The sentence to be parsed is prefaced with a marker and placed on a queue, and as parsing proceeds words are moved from the right end of the queue onto a stack. The classes of the first four words on the stack are compared to the patterns in columns 0 to 3 of **cases**. The first row matching in all four columns is selected; the action in column 4 is applied to the words on the stack indicated by the inclusive indices in columns 7 and 8, with the result replacing those words. If none of the rows match, the word at the end of the queue is moved onto the stack by the function **move**. Scanning for a matching pattern then begins anew. The process terminates when the queue is empty and none of the rules are applicable. At that time, the stack should have exactly two words: the marker and a noun, verb, adverb, or conjunction; anything else signals syntax error.

This parsing method was first described in Iverson [1983]. The parse table is a compact representation of a large amount of information; it has guided both the evolution of the language and its implementation. The following example illustrates parsing on the sentence `((i.#y)=i.~y)#y` where `y=.'abc'`. (§ denotes the marker.)

QUEUE	STACK	RULE/ ACTION	COMMENT
`§((i.#y)=i.~y)#y`			
`§((i.#y)=i.~y)#`	`'aba'`	13 move	
`§((i.#y)=i.~y)`	`#'aba'`	13 move	
`§((i.#y)=i.~y`	`)#'aba'`	13 move	
`§((i.#y)=i.~`	`'aba')#'aba'`	13 move	
`§((i.#y)=i.`	`~'aba')#'aba'`	13 move	
`§((i.#y)=`	`i.~'aba')#'aba'`	13 move	
`§((i.#y)`	`=i.~'aba')#'aba'`	13 move	
`§((i.#y)`	`=v0 'aba')#'aba'`	3 adv	`v0=.i.~`
`§((i.#y`	`)=v0 'aba')#'aba'`	13 move	
`§((i.#`	`'aba')=v0 'aba')#'aba'`	13 move	
`§((i.`	`#'aba')=v0 'aba')#'aba'`	13 move	
`§((`	`i.#'aba')=v0 'aba')#'aba'`	13 move	
`§(`	`(i.#'aba')=v0 'aba')#'aba'`	13 move	
`§(`	`(i.3)=v0 'aba')#'aba'`	1 monad	`3 -: #'aba'`
`§(`	`(0 1 2)=v0 'aba')#'aba'`	0 monad	`0 1 2 -: i.3`
`§(`	`0 1 2=v0 'aba')#'aba'`	12 punc	
`§(`	`0 1 2=0 1 0)#'aba'`	1 monad	`0 1 0 -: v0 'aba'`
`§`	`(0 1 2=0 1 0)#'aba'`	13 move	
`§`	`(1 1 0)#'aba'`	2 dyad	`1 1 0 -: 0 1 2=0 1 0`
`§`	`1 1 0#'aba'`	12 punc	
	`§1 1 0#'aba'`	13 move	
	`§'ab'`	2 dyad	`'ab' -: 1 1 0#'aba'`

Functions `vmonad, vdyad,` ... in column 5 of `cases` are used by `vtrans` in file pv.c to implement `s : 11`, a tacit verb equivalent to `s : ''` or `'' : s`. As described in Hui, Iverson, and McDonnell [1991], `vtrans` works by parsing `s`. A parallel stack is maintained, and actions on the stack have parallel actions on corresponding objects on the parallel stack. In particular, when an action *applies* a verb to its argument(s), resulting in a noun, the parallel action *composes* the verb with tacit verbs that produce the arguments, resulting in a new tacit verb.

Similarly, functions `cmonad, cdyad,` ... in column 6 of `cases` are used by `ctrans` in file pc.c to implement `s : 12`, a tacit conjunction equivalent to `s : 2`.

1.3 Trains

A *train* is an isolated phrase not interpreted by the parsing rules pertaining to verbs, adverbs, and conjunctions, and (as a matter of language design) may be assigned any meaning whatsoever. Iverson and McDonnell [1989] defined a train of three verbs as a *fork* and a train of two verbs as a *hook*. That is, if `f`, `g`, and `h` are verbs, then so are `(f g h)` and `(g h)`, and:

```
        FORK                          HOOK

   g          g               g            g
  / \        / \             / \          / \
 f   h      f   h           y   h        x   h
 |   |     /\   /\              |            |
 y   y    x  y x  y             y            y
```

Similarly, trains of two or three adverbs and conjunctions can be assigned meanings. The interpretation of trains of two or three adverbs and conjunctions are as follows:

TRAIN	RESULT	DEFINITION
`a0 a1 a2`	adverb	`x a0 a1 a2`
`a0 a1 c2`		undefined
`a0 c1 a2`	conjunction	`(x a0) c1 (y a2)`
`a0 c1 c2`	conjunction	`(x a0) c1 (x c2 y)`
`c0 a1 a2`	conjunction	`x c0 y a1 a2`
`c0 a1 c2`		undefined
`c0 c1 a2`	conjunction	`(x c0 y) c1 (x a2)`
`c0 c1 c2`	conjunction	`(x c0 y) c1 (x c2 y)`
`a0 a1`	adverb	`x a0 a1`
`a0 c1`	adverb	`x a0 c1 x`
`c0 a1`	conjunction	`x c0 y a1`
`c0 c1`		undefined

Finally, a conjunction in isolation with an argument *bonds* (Curries) the argument to the conjunction, producing an adverb.

Parsing rules 5 to 10 deal with trains. (See 1.2 *Parsing*.) A consequence of the rules is that a train of verbs is resolved by repeatedly forming a fork from the *rightmost* three verbs, with a final hook if the train is of even length. Likewise, a train of adverbs and conjunctions is resolved by repeatedly forming a group from the *leftmost* three adverbs or conjunctions, with a final group of two if the train is of even length.

Trains are implemented by the functions and variables in file ct.c. The main routines are:

`folk`	A train of three verbs ("`fork`" conflicts with UNIX usage)
`hook`	A train of two verbs
`forko`	A train of three adverbs and conjunctions
`hooko`	A train of two adverbs and conjunctions
`advform`	A conjunction in isolation with an argument
`gtrain`	The noun case of the adverb \

1.4 Name Resolution

During parsing, words are moved from the queue to the stack (see 1.2 *Parsing*). Suppose a name `xyz` is being moved. If `xyz` is immediately to the left of a copula, it (as a name) is put on the stack. Otherwise, if `xyz` denotes a noun, that noun is put on the stack; if `xyz` denotes a verb, adverb, or conjunction, `'xyz'~` is put on the stack, to be evaluated when the verb, adverb, or conjunction is applied.

Names and their assigned values are stored in symbol tables. A symbol table is an array of type `SYMB` whose atoms are pairs (name,value). (See 2.1 *Arrays*.) Functions and variables in file s.c work with symbol tables. In particular, `symbis(a,w,symb)` assigns the name `a` to `w` in the symbol table `symb`, and `symbrd(w)` "reads" the value of the name `w`.

2. Nouns

2.1 Arrays

The fundamental data structure is the array, that is, an object of the C data type `A` defined in file jt.h:

```
typedef long I;
typedef struct {I t,c,n,r,s[1];} *A;
```

All objects, whether numeric, literal, or boxed, whether noun, verb, adverb, or conjunction (or other), are represented by arrays. For example, the string `'Cogito, ergo sum.'`, the atom `1.61803`, and the table `i.3 4` are represented thus:

t	c	n	r	s[0]					
CHAR	1	17	1	17	Cogi	to,	ergo	sum	.

t	c	n	r	
FL	1	1	0	1.61803

t	c	n	r	s[0]	s[1]					
INT	1	12	2	3	4	0	1	2		
		3	4	5	6	7	8	9	10	11

The parts of an array, and macros for manipulating them, are as follows:

PART	MACRO	DESCRIPTION
`t`	`AT`	type
`c`	`AC`	reference count
`n`	`AN`	number of atoms
`r`	`AR`	rank
`s`	`AS`	shape
	`AV`	"value", atoms in ravelled order

The shape `s` consists of `r` integers whose product equals `n`. The atoms of the array follow immediately after `s`, in ravelled (row major) order. Setting `t`, `n`, `r`, or `s` incorrectly, or exceeding the bounds of the array specified by these quantities, almost always lead to erratic behaviour and catastrophic failure.

The macros `AT`, `AC`, `AN`, and `AR` denote "fullword" integers and may occur on the left or right of an assignment (*i.e.* they are "lvalues"). `AS` is an integer pointer. `AV` is also an integer pointer, and must be *cast* to a C data type appropriate for the type of array. (See 2.2 *Types*.)

All arrays are created using the macro `GA` in file j.h. The statement

```
GA(xyz,t,n,r,s);
```

creates an array named `xyz` of type `t` and rank `r`, having `n` atoms and shape `s`. If the rank is 0, `s` is ignored; if the rank is 1, again `s` is ignored, and the shape is set to `n`; otherwise, if `s` is 0, the shape is not initialized by `GA` (and must be initialized subsequently). `GA` returns zero if the array can not be created.

For example, the arrays diagrammed on the previous page might be created as follows, under the names `ces`, `phi`, and `m`:

```
typedef char   C;
typedef double D;

A ces,m,phi; I j,*s,*v;

GA(ces,CHAR,17,1,0);
memcpy((C*)AV(ces),"Cogito, ergo sum.",(size_t)17);

GA(phi,FL,1,0,0);
*(D*)AV(phi)=1.61803;

GA(m,INT,12,2,0);
s=AS(m); *s=3; *(1+s)=4;
v=AV(m); for(j=0;12>j;++j)*v++=j;
```

The following utilities in file u.c are convenient for creating simple arrays:

`A sc(I k)`	An integer atom with value `k`
`A scalar4(I t,I k)`	An atom of type `t` with 4-byte value `k`
`A scf(D x)`	A floating point atom with value `x`
`A scc(C c)`	A literal atom with value `c`
`A apv(I n,I b,I m)`	The arithmetic progression vector `b+m*i.n`.
`A str(I n,C*s)`	A string (literal list) of length `n` with value the characters pointed to by `s`.
`A cstr(C*s)`	A string with value the characters in the 0-terminated string `s`.

For example, the first two arrays diagrammed on the first page of this chapter might be created by `str(17L,"Cogito, ergo sum.")` or `cstr("Cogito, ergo sum.")` and by `scf(1.61803)`; and `sc(k)` is equivalent to `scalar4(INT,k)`.

A few useful constants are also provided. They are initialized in file i.c.

`zero`	`0`
`one`	`1`
`two`	`2`
`neg1`	`_1`
`pie`	`o.1` ("`pi`" conflicts with C usage)
`a0j1`	`0j1`
`mtv`	`$0`
`jot`	`<$0`
`dash`	`'-'`

2.2 Types

If `x` is an array, its *type* `AT(x)` specifies how the atoms starting at `AV(x)` are to be interpreted. In C programming terms, `AV(x)` must be *cast* to a pointer of the appropriate C data type:

AT(x)	C DATA TYPE	DESCRIPTION
BOOL	B	Boolean
CHAR	C	literal
INT	I	integer
FL	D	floating point
CMPX	Z	complex
BOX	A	boxed
VERB	V	verb
ADV	V	adverb
CONJ	V	conjunction
NAME	C	name
LPAR	I	left parenthesis
RPAR	I	right parenthesis
ASGN	I	assignment
MARK	I	parser marker
SYMB	SY	symbol table

For example, if `x` is literal and `s=(C*)AV(x)`, then `s[i]` is character `i` of `x`. The C data types in the table are all `typedefs` found in file jt.h; the data type `V` is explained in Chapter 4.

Types are fullword integers, and are powers of 2 to permit convenient tests for "composite" types. For example, if:

```
#define NUMERIC        (BOOL+INT+FL+CMPX)
#define NOUN           (NUMERIC+CHAR+BOX)
```

Then the phrase `NUMERIC&AT(x)` tests for numeric arrays, and the phrase `NOUN&AT(x)` tests for nouns. Such comparisons play a key role in the parser (see 1.2 *Parsing*).

A numeric array is accepted as argument by a primitive, regardless of its type, if it is mathematically within the domain of the primitive. For example, a primitive with integral domain would accept integers in an array of type `FL`, `CMPX`, or `BOOL`, or of course `INT`. (This analytic property does not extend to functions internal to the implementation.) Functions in the file k.c convert between numeric types. A converted result is an array of the target type equal to the argument within *fuzz* (see 7 *Comparatives*). The following functions are available:

`cvt(t,x)`	Convert `x` to type `t`; signal error if not possible
`pcvt(t,x)`	Convert `x` to type `t`; return `x` if not possible
`xcvt(x)`	Convert `x` to the "lowest" type

The utility `bp` in file u.c applies to a type, and returns the number of bytes per atom of that type. Thus `bp(INT)` is 4; `bp(AT(x))` is the number of bytes per atom of `x`; and `16+(4*AR(x))+AN(x)*bp(AT(x))` is the number of bytes required by `x` — 4 bytes each for `t`,`c`,`n`,`r`; 4 bytes each for the `AR(x)` elements of the shape; and `bp(AT(x))` bytes each for `AN(x)` atoms.

The atoms of a boxed array are pointers to other arrays, and are accessible through `(A*)AV(x)`, as the following example illustrates. `aib` applies to a boxed array `x`, and returns the number of atoms in each box of `x`:

```
#define R  return

A aib(x)A x;{A*u,z;I j,*v;
 GA(z,INT,AN(x),AR(x),AS(x));                /* 1 */
 u=(A*)AV(x); v=AV(z);                       /* 2 */
 for(j=0;AN(x)>j;++j)*v++=AN(*u++);          /* 3 */
 R z;
}
```

Line 1 creates an integer array `z` having the same rank and shape as `x`. Line 2 initializes pointer variables `u` and `v` for traversing `x` and `z`. Line 3 runs through the atoms of `x`, through `u`, and records the number of atoms in each. Since the data type of `u` is `A*`, the data type of `*u` is `A` and are subject to `AN`, `AT`, `AV`, etc.

2.3 Memory Management

When an array is created, `malloc` is called to obtain the requisite storage; when this storage is no longer needed, `free` is called to return it to the underlying system. No "garbage collection" is done. The performance of this strategy is adequate on modern virtual memory systems. To facilitate the implementation of alternative strategies, the use of `malloc` and `free` is limited to a single instance each, in the file m.c.

The reference count of an array is incremented when it is assigned a name, directly or indirectly, and is decremented when the name is re-assigned or erased; when the reference count of an array reaches 0, its storage is freed.

When an array is created, a pointer to it is entered in a "temp stack" (`tstack` in file m.c). A *temp* is an array on this stack with a reference count of one. The temp stack plays an important role in the main execution loop (see 0 *Introduction*). In an iteration of the loop,

- The top of the temp stack is recorded;
- A line of user-input is executed; and
- Temps from the current top-of-stack to the old top-of-stack recorded above, are freed.

This device permits functions to be written without explicit memory management code. For example, the monad `,:` is written:

```
F1(lamin1){R reshape(over(one,shape(w)),w);}
```

And `lamin1` need not be concerned with temps used in `reshape`, `over`, or `shape`, because they are accounted for in the main loop.

On the other hand, a function *may* account for temps: On entry into the function, the current top-of-stack is recorded; on exit, temps are freed down to the recorded point. (These actions are mediated by the macros `PROLOG` and `EPILOG`.) Whether a function accounts for temps does not affect the logic of functions that it calls, nor functions that call it.

3. Verbs
3.1 Anatomy of a Verb

Verbs are implemented as functions. A verb applies to a noun (if used monadically) or to two nouns (if used dyadically), and produces a noun. The data type `AF` and the macros `F1` and `F2` codify these properties:

```
typedef A(*AF)();

#define F1(f)   A f(w)A w;
#define F2(f)   A f(a,w)A a,w;
```

`AF` is the data type of a function having these properties. `F1` and `F2` are used to specify the headers of functions implementing verbs. (They are also used to specify headers of adverbs and conjunctions.) By far the majority of functions in the implementation are so specified. Verbs are represented by arrays of type `VERB`; the details of this representation are deferred until the next chapter, 4 *Adverbs and Conjunctions*.

The verb `j.` is used here to illustrate the relationship between relevant system components. Recall that `j.` has monad `0j1&*` and dyad `+j.`, with ranks `_ 0 0`. There are three main steps in the implementation:

1. Define and declare functions which implement the monad and dyad.
2. Associate `j.` with the functions and other information.
3. Specify obverses, identity functions, and variants (if any).

The steps are executed as follows:

1. Functions which implement the monad and dyad `j.` are added to file vm.c (or to one of several v*.c files), and declarations are added to je.h:

FILE vm.c

```
F1(jdot1){R times(a0j1,w);}
F2(jdot2){R plus(a,jdot1(w));}
```

FILE je.h

```
extern A jdot1();
extern A jdot2();
```

2. The association between `j.` and `jdot1` and `jdot2` is established in the tables `ps` and `psptr` in file t.c. `psptr[x]` is the index in `ps` for byte value `x`. The ID for `j.` is `CJDOT` (defined in file jc.h; see 1.1 *Word Formation*), therefore the information for `j.` can be found in `ps[psptr[CJDOT]]`. Entries in that locale are as follows:

```
/*199 E. CEBAR */ {VERB, 0,       ebar,    0,    RMAX,RMAX,0     },
/*200 f. CFIX  */ {ADV,  fix,     0,       0,    0,   0,   0     },
/*201 i. CIOTA */ {VERB, iota,    indexof,1,     RMAX,RMAX,0     },
/*202 j. CJDOT */ {VERB, jdot1,   jdot2,   RMAX,0,    0,   0     },
/*203 o. CCIRC */ {VERB, pitimes,circle,   RMAX,0,    0,   0     },
/*204 p. CPOLY */ {VERB, poly1,   poly2,   1,    1,   0,   CPOLY},
```

The entry for `j.` indicates that it is a verb, with monad `jdot1`, dyad `jdot2`, monadic rank `RMAX`, left dyadic rank 0, right dyadic ranks 0, and a non-primitive inverse (if it has an inverse at all). The information in `ps` and `psptr` are used by the utility `ds` ("define symbol") in file au.c. `ds` applies to an ID and produces the corresponding primitive. Thus, `ds(CJDOT)` is `j.`.

3. A verb may have additional parts which can not be specified as static data structures. (`ps` and `psptr` are static data structures.) Such information is embodied in functions `inv` and `invamp` (obverses) in file ai.c, `iden` (identities) in ai.c, and `fit` (variants) in cf.c. See 3.4 *Obverses, Identities, and Variants*.

The obverse for `j.`, `n&j.`, and `j.&n` are as follows:

```
j.          %&0j1
n&j.        %&0j1@(-&n)   or   (j.^:_1)@(-&n)
j.&n        -&(j.n)
```

The obverse of `j.` is implemented as `case CJDOT` in `inv`; those for `n&j.` and `j.&n` are implemented as `case CJDOT` in `invamp`. The identity function of `j.` is `$&0@}.@$`, and is implemented as `case CJDOT` in `iden`. `j.` has no variants; the implementation of a variant would have required a `case` in `fit`.

3.2 Rank

The ranks of a verb are three integers of the monadic rank, left rank, and right rank. A verb need only be defined on arguments of rank bounded by its ranks; the extension to higher-ranked arguments is uniform for all verbs. The intrinsic (default) ranks of a verb `u` may be augmented by the rank conjunction, thus: `u"n`, which may be modelled as follows:

```
rank   =. #@$
rep    =. 3&$&.|.
cellax =. 0&>.@(+rank)`(<.rank) @. (0&<:@[)
enl    =. ]`(<@$ , [ $: */@[ }. ]) @. (*@#@])
enc    =. -@cellax ((}.$) $ ({.$) enl ,@]) ]
sfx    =. -@<.&rank
agree  =. (sfx {. $@[) -: (sfx {. $@])
frame  =. ('err'&+)`($@([`]@.(<&rank))) @. agree

r      =. rep n
mcell  =. (0{r)&enc
lcell  =. (1{r)&enc@[    [.  lframe =. frame ($,) [
rcell  =. (2{r)&enc@]    [.  rframe =. frame ($,) ]

u"n y        is   u&> mcell y
x u"n y      is   x (lcell (lframe u&> rframe) rcell) y
```

The utility `rank` returns the rank of its argument, and `rep` replicates one or two ranks into three. `r cellax y` computes the number of cell axes for rank `r` and noun `y`; `s enl,y` boxes the first cell of `y` for cell shape `s`; and `r enc y` boxes the cells of `y` for rank `r`. `lcell` (`rcell`) builds an array of boxed left (right) argument cells; `lframe` and `rframe` check these arrays for agreement (*viz.*, shapes must match in suffix), then reshape the lower-ranked array to the shape of the other. In the expression for the dyad, `u&>` applies to left and right arguments of the same shape; in both expressions, `u` applies to cells with rank bounded by `n`.

(The preceding text borrows extensively from Hui [1987] A.2.)

The model is implemented by functions `rank1ex` and `rank2ex` ("rank execution") in file cr.c. A function `f` has access to the entire arguments of the verb that it implements, regardless of the ranks of the verb. Within `f`, rank effects can be achieved by invoking `rank1ex` and `rank2ex`, mediated by the macros `F1RANK` and `F2RANK`:

```
A rank1ex(    A w,A self,I m,    AF f1);
A rank2ex(A a,A w,A self,I l,I r,AF f2);

F1RANK(m,  f1,self);
F2RANK(l,r,f2,self);
```

`a` and `w` are the left and right arguments of the verb; `f1` and `f2` are functions which implement the monad and dyad; `m`, `l`, `r` are ranks; and `self` is an array representing the verb (see 4 *Adverbs and Conjunctions*). For example, the dyad `#` has ranks `1 _` and is implemented by the function `repeat`, which uses `F2RANK` as follows:

```
F2(repeat){A z;C*v,*x;I c,j,k,m,p=0,n,r,t,*u;
 F2RANK(1,RMAX,repeat,0);
 RZ(a=vi(a)); ...
}
```

If the argument ranks are not greater than the verb ranks, then `F2RANK` (`F1RANK`) does nothing, and execution proceeds to the statement following the macro; if the argument ranks *are* greater, then `F2RANK` (`F1RANK`) invokes `rank2ex` (`rank1ex`), and on return therefrom exits `f` with the result obtained therefrom. In this scheme, `rank2ex` (`rank1ex`) invokes `f` repeatedly, but with arguments of rank bounded by the verb ranks.

A function may implement rank by other means. For example, the dyad `{` has ranks `0 _` and is implemented by the function `from`, which eschews `rank2ex` on numeric left arguments when rank effects are rather simple. (`from` does use `rank2ex` on boxed left arguments.) Atomic verbs also implement rank independently to exploit the special properties of such verbs. See the next section, 3.3 *Atomic (Scalar) Verbs.*

Verbs derived from adverbs and conjunctions are *always* invoked with `self`. The macros `PREF1` and `PREF2` are used in such cases, wherein `rank1ex` and `rank2ex` are invoked with ranks extracted from `self`, and not with "hard-wired" numbers as in the use of `F1RANK` and `F2RANK` for primitive verbs.

3.3 Atomic (Scalar) Verbs

An atomic verb is a primitive verb of the form `f"_ : g` (that is, a verb whose monad is `f"_` and whose dyad is `g`), where `f` and `g` have zero argument and result ranks. (These are the *scalar functions* in APL.) The *shape* of the result is therefore determined by the shapes of the arguments alone: For monads, the shape of the result is simply the shape of the argument; for dyads, it is the shape of the higher-ranked argument (and the shape of the other argument must be a suffix of this shape). The *type* of the result is determined by the types of the arguments.

Mechanisms described in the previous section (3.2 *Rank*) suffice to implement atomic verbs. However, the special properties of atomic verbs can be exploited to effect more efficient computation, as follows:

In the implementation, the definition of an atomic verb begins by specifying the computation on atoms of each data type, in the form of *kernels*. A kernel is a function defined by the macros `SF1` or `SF2` (in file v.h). For example, the kernels for the dyad `+` are as follows (in file ve.c):

```
static SF2(bplus,B,I,  *u+*v)
static SF2(iplus,I,D,  *u+(D)*v)
static SF2(dplus,D,D,  *u+*v)
static SF2(jplus,Z,Z,  zplus(*u,*v))
```

As the examples illustrate, `SF2` has four arguments, `f`, `Tv`, `Tx`, and `exp`. `f` is the name of the function being defined; `Tv` is the data type of the arguments; `Tx` is the data type of the result; and `exp` is an expression for computing the result from the arguments, wherein the left argument is available as a pointer of data type `Tv` named `u` and the right argument a pointer of data type `Tv` named `v`. The definition of the macro is rather shorter than the preceding description:

```
#define SF2(f,Tv,Tx,exp)      \
    B f(u,v,x)Tv*u,*v;Tx*x;{*x=(exp); R!jerr;}
```

The formal result of a kernel (*i.e.* the result as far as C is concerned) is Boolean, and is 1 if no errors are encountered. (The variable `jerr` is explained in Section 3.5 *Error Handling*.)

`SF1` is similarly defined. In the expression `exp`, the right (and only) argument is available as a pointer of data type `Tv` named `v`.

The logic for applying kernels is embodied in functions `sex1` and `sex2` ("scalar execution") in file cr.c, with the following prototypes:

```
A sex1(    A w,I zt,SF f1)
A sex2(A a,A w,I zt,SF f2)
```

`a` and `w` are the usual array arguments of a verb; `zt` is the type of the result (`BOOL`, `INT`, `FL`, etc.); and `f1` and `f2` are kernels. `sex1` and `sex2` first allocate space for the result, then invoke `f1` and `f2` repeatedly with pointers to the arguments and result.

The definition of an atomic verb is completed by specifying a "cover" function which first coerces the arguments to the same type (or to some type depending on the arguments), then invokes `sex1` or `sex2` with the appropriate result type and kernel. Thus, `+` is implemented by `plus`:

```
F2(plus){
 switch(coerce2(&a,&w,BOOL)){
  case BOOL: R sex2(a,w,INT ,bplus);
  case INT:  R pcvt(INT,sex2(a,w,FL,iplus));
  case FL:   R sex2(a,w,FL  ,dplus);
  case CMPX: R sex2(a,w,CMPX,jplus);
  default:   R 0;
}}
```

`plus` is the function put into the table `ps` in file t.c, as described in Section 3.1.

3.4 Obverses, Identities, and Variants

Verbs have additional parts — obverse, identity, and variants — which can not be not be specified as static data structures. Such information is embodied in functions.

• Obverses

A verb `u` is an obverse (usually the inverse) of a verb `v` if `x=u v x` for a significant subdomain of `v`. The obverse is used in the conjunctions *under* (`&.`) and *power* (`^:`). For example, exponential `^` and logarithm `^.` are obverses, and:

```
3+&.^.4   is  ^ (^.3)+^.4          ^ ^:_1   is  ^.
3*&.^ 4   is  ^.(^ 3)*^ 4          ^.^:_1   is  ^
```

Obverses are produced by the function `inv` in file ai.c. (`inv` implements `^:_1`.) The logic is a combination of table look-up and nested branch tables (`switch`-es).

PRIMITIVES. If the obverse of a primitive verb is itself primitive, the information is recorded in the table `ps` in file t.c. For example, the ID for `^` is `CEXP` and that for `^.` is `CLOG`; therefore `ps[CEXP].inv` is `CLOG` and `ps[CLOG].inv` is `CEXP`. (`ps[].inv` is zero otherwise.)

BONDED VERBS. Bonding (Currying) is fixing an argument of a dyad to derive a monad: `n&v` or `v&n`. For example, `10&^.` is *base-10 log* and `^&0.5` is *square root*. The obverse of a bonded verb is computed by the subfunction `invamp` in file ai.c, invoked by `inv` as appropriate.

PREFIX AND SUFFIX. Sum prefix `+/\` and sum suffix `+/\.` can be expressed as pre-multiplication by matrices obtained by applying `+/\` and `+/\.` on the identity matrix. The obverse is therefore pre-multiplication by the *matrix inverse* of these matrices. (The actual obverse is a more efficient equivalent derived therefrom.) Similar reasoning applies to `-`, `*`, and `%`, and to `=` and `~:` on Boolean arguments. The logic is embodied as a sub-`switch` in `inv`, under `case CBSLASH` and `case CBSDOT`.

VERBS DERIVED FROM ~. The monad `v~` computes `y v y`. For example, `+~` is *double*. The obverses of a few such verbs are implemented by a sub-`switch` in `inv`, under `case CTILDE`.

ASSIGNED OBVERSE. A verb may be assigned an obverse with the *obverse conjunction* (`:.`). `f=.u :. v` is like `u` but its obverse is `v`.

OTHER VERBS. `inv` applies to a few other verbs, including `u@v` and `u&v`, whose obverse are `(v inv)@(u inv)` and `(v inv)&(u inv)`.

DEFAULT OBVERSE. Verbs which would otherwise be non-obvertable are assigned an obverse `%.@:v@(=@i.@#) +/ .* ]` (function `invdef` in ai.c). The reasoning is similar to that under PREFIX AND SUFFIX.

• Identities

`u/y` applies the dyad `u` between the items of `y`. When `y` has *zero* items, the result of `u/y` obtains by applying the *identity function* of `u` to `y`, so-called because `u/(iu y),y` or `u/y,(iu y)` is `y` for a significant subdomain of `u`.

Identity functions are computed by the function `iden` in file ai.c. `iden` behaves like an adverb, applying to verbs and producing verbs. The logic is implemented as a branch table (a `switch`). Not all verbs have identity functions; `iden` signals error in such cases (*i.e.* in `u/''` when `u` does not have an identity).

• Variants

Variants of a verb are produced by the *fit* conjunction (`!.`), and are used to effect tolerant comparison (`= < <. <: > >. >: +. * *. -: ~. ~: | #: e. i.`), formatting to a specific precision (`":` and `5!:3`), shifts (`|.`), and factorial polynomials (`^`).

`!.` is implemented by the function `fit` in file cf.c. The logic is implemented as a branch table (a `switch`). Not all verbs have variants; `fit` signals error in such cases.

3.5 Error Handling

When an error is encountered in a function, the global variable `jerr` is set to an error number, and zero is returned. Therefore, when calling a function that can not have zero as a valid result (but does return a result), the returned value must be checked for zero; when calling a "void" function or one whose range includes zero, `jerr` must be inspected.

Error numbers range between 1 and `NEVM`, and are referenced by the `EV*` names ("event" names, defined in file j.h). The function `jsignal` (u.c) applies to an error number, sets `jerr` to this number, and (if the global variable `errsee` is 1) displays the appropriate error message; `jsignal` exits immediately if `jerr` is already nonzero. `qevm` is a list of the error messages. These messages are initialized in function `jinit2` (i.c), and may be inspected and changed by the user through `9!:8` and `9!:9`.

The macro `ASSERT` (j.h) is used extensively in argument validation. It applies to a proposition and an error number. For example, the following statements check whether `w` is a literal atom:

```
ASSERT(!AR(w),EVRANK);
ASSERT(CHAR&AT(w),EVDOMAIN);
```

If the proposition is nonzero, execution proceeds to the next statement; otherwise, the indicated error is `jsignal`-ed and a zero is returned. The macros `RZ` and `RE` (j.h) are used in function calls. `RZ` returns zero if its argument is zero; `RE` evaluates its argument, and returns zero if `jerr` is nonzero. For example, the function `iota` (implementing the monad `i.`) uses `RZ` to check the results of functions that it calls, as follows:

```
F1(iota){A z;I m,n,*v;
 F1RANK(1,iota,0);
 RZ(w=vi(w));
 n=AN(w); v=AV(w); m=prod(n,v);
 RZ(z=reshape(mag(w),apv(ABS(m),0L,1L)));
 DO(n*!!m, if(0>v[i])RZ(z=ranklex(z,0L,n-i,reverse));); 
 R z;
}
```

The arguments of a function may be the result of another function; the convention is that a function checks its arguments for zero and returns zero immediately in such cases. Thus, in `iota` above:

```
RZ(z=reshape(mag(w),apv(ABS(m),0L,1L)));
```

If `reshape` did not check for zero arguments, the statement would have to be elaborated:

```
RZ(t0=mag(w));
RZ(t1=apv(ABS(m),0L,1L));
RZ(z=reshape(t0,t1));
```

A *conventional function* is a function that follows the conventions described herein — return zero on zero arguments and on errors. The data type `AF` (jt.h) typifies a conventional function. Most functions in the system are conventional; in particular, all functions implementing primitives are conventional. Expressions and statements that use only conventional functions need not employ `RZ` or `RE`, and the resulting programs are neater. For example, consider the functions `lamin1` and `lamin2` (vs.c), implementing *laminate* (; .):

```
F1(lamin1){R reshape(over(one,shape(w)),w);}
F2(lamin2){RZ(a&&w); R over(AR(a)?lamin1(a):a,
                 AR(w)?lamin1(w):AR(a)?w:table(w));}
```

`lamin2` must check for zero arguments `RZ(a&&w)`, because it applies the *un*conventional macro `AR` to the arguments. In contrast, `lamin1` applies only conventional functions to *its* argument and to results of conventional functions on that argument.

4. Adverbs and Conjunctions

An adverb is monadic, applying to a noun or verb argument on its *left*; a conjunction is dyadic, applying to noun or verb arguments on its *left and right*. The result is usually a verb, but can also be a noun, adverb, or conjunction.

The conjunction `&` is used here to illustrate the relationship between relevant system components. (The implementation of adverbs is similar.) Recall that `&` derives a verb depending on whether the arguments are nouns (`m` and `n`) or verbs (`u` and `v`):

`m&n`	undefined
`m&v`	`m&v y` is `m v y`
`u&n`	`u&n y` is `y u n`
`u&v`	`u&v y` is `u v y`; `x u&v y` is `(v x)u(v y)`

A verb derived from `&` is (internally) an array of type `VERB` whose value is interpreted according to the data type `V`, defined in file jt.h as follows:

```
typedef struct {AF f1,f2;A f,g,h;I mr,lr,rr;C id;} V;
```

`f1`	monad	`mr`	monadic rank
`f2`	dyad	`lr`	left dyadic rank
`f`	left conj. or adverb argument	`rr`	right dyadic rank
`g`	right conj. argument	`id`	identification
`h`	auxiliary argument		

If `fn=.%.&|:`, the internal array for `fn` is:

t	c	n	r
VERB	1	1	0

f1	f2	f	g	h	mr	lr	rr	id
on1	on2	%.	|:	0	_	_	_	&

Access to fields in `fn` is by name and by macros defined in jt.h and a.h, and *never* by offsets and indices. Thus, `AV(fn)` points to the "value" of `fn`; and if `v=(V*)AV(fn)`, then `v->f1` is `on1`; `v->f2` is `on2`; `v->f` is the array for `%.`; `v->g` is the array for `|:` (that is, `v->f` and `v->g` are arrays similar to `fn`); `v->mr` is `_` (indicating that `fn` has infinite monadic rank); and so on. The macro `VAV(f)` — `((V*)AV(f))` — is useful for working with adverbs and conjunctions.

To introduce `&` into the system, functions which implement `&` are added to file c.c (or to one of several c*.c files), and declarations of global objects are added to file je.h:

FILE C.C:

```
static DF1(withl){DECLFG; R g2(fs,w,gs);}
static DF1(withr){DECLFG; R f2(w,gs,fs);}
static CS1(on1,  f1(g1(w,gs),fs))
static CS2(on2,  f2(g1(a,gs),g1(w,gs),fs))

F2(amp){
 RZ(a&&w);
 switch(CONJCASE(a,w)){
  case NN: ASSERT(0,EVDOMAIN);
  case NV: R CDERIV(CAMP,withl,0L,RMAXL,RMAXL,RMAXL);
  case VN: R CDERIV(CAMP,withr,0L,RMAXL,RMAXL,RMAXL);
  case VV: R CDERIV(CAMP,on1,on2,mr(w),mr(w),mr(w));
}}
```

FILE je.h:

```
extern A amp();
```

Corresponding to the four possibilities, `amp` defines four cases, which either signal error or return a verb; the functions `withl`, `withr`, `on1`, and `on2` are invoked when a verb derived from `&` is applied. For example, `%.&|: m=.?4 4$100` first branches to the case `VV` in `amp`, and subsequently applies `on1` to `m`. Consider a partial macro expansion of `on1` and the values of its local variables for this example:

MACRO EXPANSION:

```
static A on1(w,self)A w,self;{PROLOG;V*v=VAV(self);
 A fs=v->f; AF f1=fs?VAV(fs)->f1:0, f2=fs?VAV(fs)->f2:0;
 A gs=v->g; AF g1=gs?VAV(gs)->f1:0, g2=gs?VAV(gs)->f2:0;
 PREF1(on1);
 z=f1(g1(w,gs),fs);
 EPILOG(z);
}
```

LOCAL VARIABLES:

`w`	m
`self`	`fn` above
`v`	pointer to the value part of the array `fn`

`fs`	`%.`	`f1`	monad of `%.`	`f2`	dyad of `%.`
`gs`	`|:`	`g1`	monad of `|:`	`g2`	dyad of `|:`

The initialization of `v`, `fs`, `f1`, and so on are the same for all adverbs and conjunctions. (The details of such initialization are normally suppressed by the use of macros.) If an argument to `&` (*i.e.* `fs` or `gs`) is itself a result of adverbs and conjunctions, expressions such as `g1(w,gs)` or `f1(xx,fs)` engender further executions as occurs in `on1`. The macro `PREF1` implements rank (see 3.2 *Rank*), and the macros `PROLOG` and `EPILOG` manage memory (see 2.3 *Memory Management*).

The association between `&` and `amp` is established in the tables `ps` and `psptr` in file t.c. `psptr[x]` is the index of the entry in `ps` for byte value `x`. The ID for `&` is `CAMP` (defined in file jc.h; see 1.1 *Word Formation*), so `ps[psptr[CAMP]]` contains the information for `&`:

```
/* 38 26 &  CAMP */ {CONJ, 0, amp, 0, 0, 0, 0 },
```

The entry specifies that `&` is a conjunction and has monad 0 (none), dyad `amp`, ranks 0, and inverse 0 (none). The information in `ps` and `psptr` are used by the utility `ds` ("define symbol") in file au.c. `ds` applies to an ID, and produces the corresponding primitive. Thus, `ds(CAMP)` is `&`.

The utilities `ac1` and `ac2` in file au.c enable non-primitive functions (those *not* put into `ps` and `psptr`) to participate in phrases involving adverbs and conjunctions. Suppose `f1` and `f2` are functions which apply to array arguments and return array results. That is, the prototypes of `f1` and `f2` are:

```
A f1(A w);
A f2(A a, A w);
```

Then `ac1(f1)` is a monadic verb and `ac2(f2)` is a dyadic verb, and are in the domain of adverbs and conjunctions. These verbs have infinite ranks; other ranks can be specified through the function `qq` (which implements `"`). Thus, `qq(ac1(f1),sc(1L))` is a verb with rank 1. An ambivalent verb obtains by a further application of the function `colon` (which implements `:`). Thus: `colon(ac1(f1),ac2(f2))` is a verb whose monad is `ac1(f1)` and whose dyad is `ac2(f2)`.

The utilities `df1` and `df2` in file au.c apply the monad or the dyad of a verb. For example,

```
df1(w,ds(CPOUND))
df1(   w,amp(ds(CPOUND),ds(COPE)))
df2(a,w,amp(ds(CPOUND),ds(COPE)))
df1(w,qq(ac1(f1),sc(1L)))
```

The phrase `ds(CPOUND)` is the verb `#`, `ds(COPE)` is the verb `>`, and `amp(ds(CPOUND),ds(COPE))` is `#&>`; the examples compute `#w`, `#&>w`, `a#&>w`, and `f1` on the lists of `w`. Finally, `df1(w,ac1(f1))` is equivalent to `f1(w)`, and `df2(a,w,ac2(f2))` is equivalent to `f2(a,w)`.

5. Representation

5.1 Atomic Representation

`5!:1` is a verb that applies to a boxed name, and produces the *atomic representation* of the named object. Gerunds (results of the ` conjunction) are arrays of atomic representations. The adverb `5!:0` defines an object from its representation.

The atomic representation is a boxed list of two boxes:

noun	ID	value
verb	ID	arguments
adverb	ID	arguments
conjunction	ID	arguments

The ID is a string computed by the function `spellout` in file w.c. For a primitive with an assigned word (for example `+` or `/.`), the ID is simply that word; for those without, the ID is one of the following:

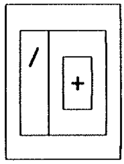
`'0'`	noun
`'2'`	hook
`'3'`	fork
`'4'`	bonded conjunction
`'5'`	2-element a-train or c-train
`'6'`	3-element a-train or c-train

The "value" in the representation of a noun is just the noun itself; arguments in the representation of a verb, adverb, or conjunction are themselves atomic representations. If an object is uniquely identified by the ID alone, then the second field is elided, and the representation is the boxed ID alone.

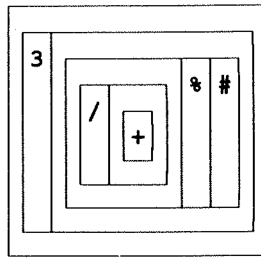
The following examples illustrate atomic representation:

```
ar =. 5!:1
plus =. +
ar <'plus'
```

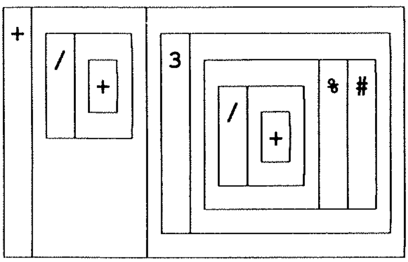
```
sum =. +/
ar <'sum'
```



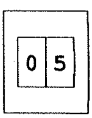
```
mean =. +/ % #
ar <'mean'
```



```
+`(+/)`(+/ % #)
```

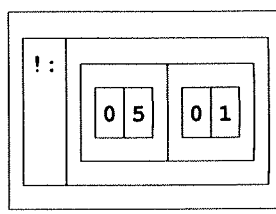


```
a=.5
ar <'a'
```

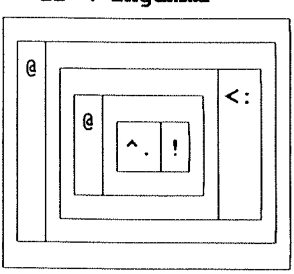


```
xenos=.!:
ar <'xenos'
```

```
ar <'ar'
```



```
lngamma =. ^.@!@<:
ar <'lngamma'
```



5.2 Display Representation

`5!:2` is a verb that applies to a boxed name, and produces the *display representation* of the named object. (This is what is displayed if the result of an input line is a verb, adverb, or conjunction.) The representation can be modelled as follows:

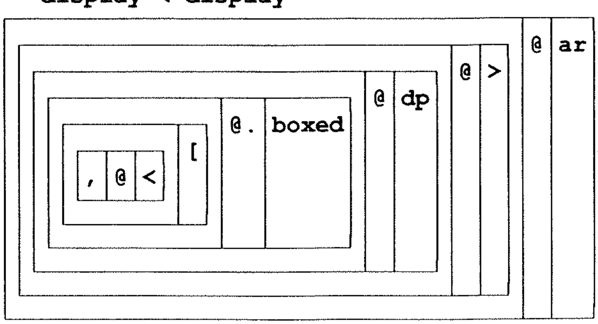
```
ar      =. 5!:1
type    =. 3!:0
boxed   =. 32&=@type
oarg    =. >@(1&{)

root    =. (<1 0)&C.@,`] @. (e.&(,&.>'0123456789')@[)

dpx     =. {.root dp&.>@oarg
dpgl    =. {.root (dpx&.>@{. , dp &.>@}.)@oarg
dpgr    =. {.root (dp &.>@{. , dpx&.>@}.)@oarg
dpg     =. dpgr`dpgl`dpx @. (i.&(<,'`')@oarg)
dptil   =. dpx`(oarg@>@{.@oarg) @. ((<,'0')&=@{.@>@{.@oarg)
dpcase  =. oarg`dpgl`dpgl`dpg`dptil`dpx @.
                                     ((;:'0@.`:4~')&i.@{. )
dp      =. ]`dpcase @. boxed

display =. ,@<`[@.boxed @ dp @ > @ ar

   display <'display'
```



The model is divided into groups of verbs. The first group are utilities:

`ar`	atomic representation
`type`	type
`boxed`	1 if boxed
`oarg`	open the second element of the list argument

`root` produces an infix representation from a root `r` and its list of arguments `a`. If `r` is a digit, it denotes a primitive without an assigned word (e.g. `'3'` denotes a *fork*; see 5.1 *Atomic Representation*), and the result of `root` is `a`; otherwise, `r root a` produces:

`a,r`	one argument
`({.a),r,(}.a)`	two arguments
`r`	no arguments (primitive)

The verbs named with the `dp` prefix apply to the opened atomic representation, and embody logic to effect "nice" displays for various special cases. The agenda items in `dpcase` are:

ID	AGENDA	
`0`	`oarg`	noun (leaf)
`@.`	`dpgl`	gerundial left subtree
`` `: ``	`dpgl`	gerundial left subtree
`4`	`dpg`	bonded conjunction; gerundial left or right subtree
`~`	`dptil`	possible instance of *evoke*
other	`dpx`	none of the above

`display` is a model of `5!:2`.

5.3 String Representation

`5!:3` is a verb that applies to a boxed name, and produces a literal list of the *string representation* of the named object. The representation conforms to the Workspace Interchange Standard (Bernecky *et al.* [1981]), and facilitates exchange of data and programs between disparate systems.

```
   sr  =. 5!:3
   str =. 'Cogito, ergo sum.'
   sr <'str'
27cstr 1 17 Cogito, ergo sum.

   ] ces =. ;: str
+------+-+----+----+
|Cogito|,|ergo|sum.|
+------+-+----+----+
   sr <'ces'
60xbces 1 4 13c- 1 6 Cogito8c- 1 1 ,11c- 1 4 ergo11c- 1
4sum.

   sum =. +/
   sr <'sum'
38xvsum 1 2 8c- 1 1 /17xb- 1 1 8c- 1 1 +
```

The string representation is the catenation of the following parts:

length Digits representing the length of the representation (excluding the length itself).

type One or two letters denoting the type of object

- `c` literal (character) array
- `n` numeric array
- `xb` boxed array
- `xv` verb
- `xa` adverb
- `xc` conjunction

The representation of a verb, adverb, or conjunction is the representation of its opened atomic representation.

name The name of the object, or `-` if anonymous.

blank	A single blank.
rank	Digits representing the rank.
blank	A single blank.
shape	Digits and blanks representing the shape, terminating in a blank.
elements	The ravelled elements. For a literal or numeric array, this is the display of the ravelled array; for a boxed array (hence for a verb, adverb, or conjunction), the elements themselves are recursively so represented.

String representation can be modelled by the following verbs:

```
ar     =. 5!:1
type   =. 3!:0
boxed  =. 32&=@type
nc     =. 4!:0

nt     =. >@({&(;:'n c n n n xb'))@(1 2 4 8 16 32&i.)@type
rs     =. ('  '&,)@(,&'  ')@":@($@$,$)
elem   =. (":@,)`(;@('-'&sn&.>"1)@,)@.boxed
sn     =. (,~ ":@#) @ (nt@],>@[,(rs,elem)@])
st     =. ('x'&,@({&'   vac')@nc)`(nt@".@>)@.(2&=@nc)
val    =. (>@ar)`(".&>)@.(2&=@nc)
upfx   =. }.~ >:@(<./)@(i.&'cnb')
sr     =. (,~ ":@#) @ (st , ] upfx@sn val)
```

The first group are utilities:

`ar`	atomic representation
`type`	type
`boxed`	1 if boxed
`nc`	name class

`nt` computes the *type* part of the representation for a noun; the result is `n`, `c`, or `xb`, depending on whether the argument is numeric, literal, or boxed.

`rs` computes *rank* and *shape*; `rs y` is `' ',(":($$y),$y),' '`, the formatted result of the rank and shape, surrounded by blanks.

`elem` computes *elements*. If the argument `y` is open, this is simply `":,y`, the format of the ravel of `y`; if boxed, it is the catenation of the representations of the boxed elements.

`sn` computes the representation of a noun whose boxed name is the left argument, and whose value is the right argument. The parts of the representation correspond to readily identifiable phrases in the definition: *length*, `":@#`; *type*, `nt`; *name*, `>@[`; *rank* and *shape*, `rs`; and *elements*, `elem`.

`st` computes the *type* part of the representation. The argument is a boxed name; the result is `n`, `c`, `xb`, `xv`, `xa`, or `xc`, according on whether the named object is numeric, literal, boxed, verb, adverb, or conjunction.

`val` computes the value to be represented, given a boxed name. If the named object is a noun, the value is simply the noun itself (execute the open of the boxed name); otherwise it is the opened atomic representation.

`upfx`, "unprefix", drops the *length* and *type* parts of the representation of a noun. The amount to be dropped is one plus the minimum index of `c`, `n`, or `b` in the argument.

`sr` is a model of `5!:3`.

5.4 Tree Representation

`5!:4` is a verb that applies to a boxed name, and produces a literal table of the *tree representation* of the named object.

```
   tree =. connect @ > @ (,.&.>/) @ (> (root;]) tr@>@ar)
   5!:4 <'tree'
                           ┌─ connect
                     ┌─ @ ─┴─ >
              ┌─ @ ──┤                ┌─ ,.
              │      └─ / ─── &. ─────┴─ >
- tree ── @ ──┤
              │      ┌─ >
              │      │     ┌─ root
              │      ├─────┼─ ;
              └──────┤     └─ ]
                     │                ┌─ tr
                     │     ┌─ @ ──────┴─ >
                     └─ @ ─┴─ ar

ar       =. 5!:1
type     =. 3!:0
boxed    =. 32&=@type
mt       =. 0&e.@$
oarg     =. >@(1&{)
shr      =. |.!.''
shl      =. 1&(|.!.'')
mat      =. (1 1&}.)@(_1 _1&}.)@":@<
boxc     =. 9!:6 ''
dash     =. 10{boxc

extent   =. (+./\ *. +./\.) @ (' '&~:) @: ({."1)
limb1    =. 1&|.@$ 1&~: }. (10 6 0{boxc)&,@($&(9{boxc))
limb     =. -@(i.&1)@[ |. #@[ {. limb1@]
pfx      =. (limb +/)@extent ,. ]
pad      =. [{.] ,. dash&=@({:"1)@] { ' '&,:@($&dash)@(-&{:$)
take     =. pad`({.&(,.' ')@[) @. (mt@])
rc       =. #@>@{."1 ; >./@:({:@$@>)
kernt    =. (0{boxc)&=@shl@[ *. ' '&~:@]
```

```
kernb   =. (6{boxc)&=@] *. ' '&~:@shl@[
kern    =. (<0 0)&{&>"2 (kernt+./"1@:+.kernb) (<_1 0)&{&>"2
gap     =. ,&.>"_1 {&((0 1$' ');1 1$' ')@kern
graft   =. (pfx&.>@{.}]) @ (,&.>/) @ gap @ ({@rc take&.> ])

lab     =. ,: @ (2&|.) @ ((' ',dash,dash,' ')&,)
label   =. lab`((,.dash)&[) @. (e.&'0123456789'@{.)
center  =. ((i.&1) -@+ <.@-:@(+/))@] |. #@] {. [
root    =. label@[ center extent@>@{.@]

leaf    =. ,@<@(((,:dash,' ')&[ center $&1@#) ,. ])@mat@":

trx     =. >@{. (root ; ]) graft@:(tr@>)@oarg
trgl    =. >@{. (root ; ]) graft@:(trx@>@{. , tr @>@}.)@oarg
trgr    =. >@{. (root ; ]) graft@:(tr @>@{. , trx@>@}.)@oarg
trg     =. trgr`trgl`trx @. (i.&(<,'`')@oarg)
trtil   =. trx`(leaf@oarg@>@{.@oarg) @.
                                  ((<,'0')&=@{.@>@{.@oarg)
trcase  =. (leaf@oarg)`trgl`trgl`trg`trtil`trx @.
                                  ((;:'0@.`:4~')&i. @{.)
tr      =. leaf`trcase @. boxed

rep     =. [. & (((# i.@#)@,@)(@]))})
right   =. (5{boxc) rep (e.&(9{boxc) *. shr"1@(e.&dash))
cross   =. (4{boxc) rep (e.&(5{boxc) *. shl"1@(e.&dash))
left    =. (3{boxc) rep (e.&(9{boxc) *. shl"1@(e.&dash))
bot     =. (7{boxc) rep (e.&(6{boxc) *. shr"1@(e.&dash))
connect =. bot @ left @ cross @ right

tree    =. connect @ > @ (,.&.>/) @ (> (root;]) tr@>@ar)
```

The model is divided into groups of definitions (which are verbs unless indicated otherwise). The first group are utilities:

`ar`	atomic representation
`type`	type
`boxed`	1 if boxed

mt	1 if empty
oarg	open the second element of the list argument
shr	shift right
shl	shift left
mat	a literal matrix image of the argument
boxc	(noun) box drawing characters
dash	(noun) the "dash" in the set of box drawing characters

A "generational tree" (GT) is a list of boxed literal tables having the same number of rows, such that nodes at the same depth are in the same box. For example, the GT for `tree` is:

```
┌──────────┬──────┬────────┬────────┬────────────┬──────┐
│          │      │        │        │┌─ connect  │      │
│          │      │        │┌─ @ ─  │└─ >        │      │
│          │      │┌─ @ ─  ││       │            │┌─ ,. │
│          │      ││       │└─ / ─  │── &. ───── │└─ >  │
│─ tree ─  │─ @ ─ ││       │┌─ >    │            │      │
│          │      ││       ││       │┌─ root     │      │
│          │      ││       ││────── │├─ ;        │      │
│          │      │└────── ││       │└─ ]        │      │
│          │      │        ││       │            │┌─ tr │
│          │      │        ││       │┌─ @ ────── │└─ >  │
│          │      │        │└─ @ ─  │└─ ar       │      │
└──────────┴──────┴────────┴────────┴────────────┴──────┘
```

`graft` is the main verb in the next group. The argument is a table whose rows are GTs for the nodes at the same depth. The result is a GT.

`root` accepts a string left argument and a GT right argument. The result is a literal matrix with the string centered relative to the GT.

`leaf` computes a unitary (single-element) GT from its argument.

`tr` applies to the opened atomic representation of an object and produces a GT. The verbs named with the `tr` prefix embody logic to effect "nice" displays for various special cases. The agenda items in `trcase` are:

ID	AGENDA	
`0`	`leaf@oarg`	noun (leaf)
`@.`	`trgl`	gerundial left subtree
`` `: ``	`trgl`	gerundial left subtree
`4`	`trg`	bonded conjunction; gerund left or right
`~`	`trtil`	possible instance of *evoke*
other	`trx`	none of the above

`rep` is a conjunction whose left argument is a single literal `c` and whose right argument is a proposition `p`, deriving a verb such that the phrase `c rep p y` replaces with `c` the positions in `y` marked by `p y`. **`connect`** substitutes ⊥ (`bot`), ├ (`left`), ┼ (`cross`), and ┤ (`right`) at nexuses of the tree.

`tree` is a model of `5!:4`.

6. Display

If the last operation in a line of user input is not assignment, the result of the line is displayed. More specifically, if the global variable `asgn` is zero at the end of executing an input line, and the line had no errors, `jpr` is invoked to display the result. `jpr` first applies `thornl` (the monad `":`) to compute the *display* of `y`, then writes the lines to the screen.

In all cases, the display of an object is a literal array. The display of a literal array is itself. The display of a verb, adverb, or conjunction is that of its display representation `5!:2` (a boxed array; see Section 5.2). The display of a numeric array is discussed in Section 6.1; that of a boxed array, in Section 6.2; and *format* (the dyad `":`) is discussed in Section 6.3.

Display is implemented by functions and variables in file f.c.

6.1 Numeric Display

The display of a numeric array `y` is a literal array having the same rank as `y` (but at least one), such that the shape of `":y` matches the shape of `y` in all but the last axis. Columns are right-justified and are separated by one space. The conversion from numeric to literal can be modelled as follows:

```
sprintf =. ":
type    =. 3!:0
real    =. {.@+.
imag    =. {:@+.

minus   =. $&'_'@('-'&=@{.)
ubar    =. >@({&(<;._1 ' _ _ _. _.'))@('iInN'&i.@{.)
afte    =. minus , (i.&0@(e.&'-+0') }. ])
efmt    =. >:@(i.&'e') ({.,afte@}.) ]
finite  =. ]`efmt@.('e'&e.)
massage =. finite`ubar@.(e.&'iInN'@{.)
fmtD    =. (minus,massage@(e.&'-+'@{. }. ])) @ sprintf

fmtB    =. {&'01'
fmtI    =. sprintf
fmtZ    =. fmtD@real , 'j'&,@fmtD@imag
fmt     =. (fmtB&.>)`(fmtI&.>)`(fmtD&.>)`(fmtZ&.>) @.
                                           (1 4 8&i.@type)

sh      =. (*/@}:,{:@(1&,))@$ ($,) ]
width   =. (<:@{. 0} ])@>:@(>./)@sh@:(#&>)
th      =. (-@width ;@:(({.&.>)"1 ]) @ fmt
```

The model is divided into groups of verbs. The first group are utilities:

`sprintf`	a function in the C library
`type`	type
`real`	the real part of a complex number
`imag`	the imaginary part of a complex number

`fmtD` formats a real number. Its constituents transform the result of `sprintf` to follow J conventions in the treatment of negative signs (`minus`), exponential notation (`efmt` and `afte`), and infinities and indeterminates (`ubar`).

`fmt` formats a numeric array into an array of boxed strings. It invokes formatters specialized for the different types: `fmtB` (Boolean), `fmtI` (integer), `fmtD` (floating point), and `fmtZ` (complex).

`sh` shapes an array into a table having the same number of rows. `width` computes the maximum width in each column of an array of boxed strings. `th` is a model of `":` on numeric arrays.

6.2 Boxed Display

The display of a boxed array `b` is a literal array `d=.":b` such that:

- The rank of `d` is the greater of 2 or the rank of `b`.
- Excluding the last two axes, the shape of `d` matches the shape of `b`.
- The frame (formed by ┌┬┐ ├┼┤ └┴┘ │─) is the same in all the planes.

Boxed display can be modelled as follows:

```
type     =. 3!:0
boxed    =. 32&= @ type
mt       =. 0&e.@$
boxc     =. 9!:6 ''
tcorn    =. 2  0{boxc
tint     =. 1 10{boxc
bcorn    =. 8  6{boxc
bint     =. 7 10{boxc

sh       =. (*/@}: , {:)@(1&,)@$ $ ,
rows     =. */\.@}:@$
bl       =. }.@(,&0)@(+/)@(0&=)@(|/ i.@{.@(,&1))
mask     =. 1&,. #&, ,.&0@>:@i.@#
mat      =. mask@bl@rows { ' '&,@sh

edge     =. ,@(1&,.)@[ }.@# +:@#@[ $ ]
left     =. edge&(3 9{boxc)@>@(0&{)@[ , "0 1 ]
right    =. edge&(5 9{boxc)@>@(0&{)@[ ,~"0 1 ]
top      =. 1&|.@(tcorn&,)@(edge&tint)@>@(1&{)@[ ,"2  ]
bot      =. 1&|.@(bcorn&,)@(edge&bint)@>@(1&{)@[ ,"2~ ]
perim    =. [ top [ bot [ left [ right ]

topleft =. (4{(boxc)&(0}) @ ((_2{(boxc)&,.) @ ((_1{boxc)&,)
inside  =. 1 1&}. @ ; @: (,.&.>/"1) @: (topleft&.>)
take    =. [ {. ]`(]&' ')@.mt@]
frame   =. [ perim {@[ inside@:(take&.>)"2 ]
rc      =. (>./@sh&.>) @: (,.@|:"2@:(0&{"1);1&{"1) @: ($&>)

thorn1  =. ":`thbox @. boxed
thbox   =. (rc frame ]) @: (mat@thorn1&.>)
```

The model is divided into groups of definitions (which are verbs unless indicated otherwise). The first group are utilities:

`type`	type
`boxed`	1 if boxed
`mt`	1 if empty
`boxc`	(noun) box drawing characters
`tcorn`	(noun) the characters ┐ ┌
`tint`	(noun) the characters ┬ ─
`bcorn`	(noun) the characters ┘ └
`bint`	(noun) the characters ┴ ─

`mat` is the main verb of the next group of definitions. The argument is a literal array; the result is a literal matrix image of the array — a literal table that "looks like" the argument array.

`perim` draws a perimeter around each plane of the right argument: According to the information in the left argument (a result of `rc`), `perim` puts ┐ ┌ ┬ ─ (`top`), ┘ └ ┴ ─ (`bot`), ├│ (`left`), or ┤│ (`right`) at appropriate positions on the perimeter of each plane.

`topleft` catenates the characters ┼ │ ─ on the top and left edges of a literal table. `inside` produces the inside (excluding perimeter) of a plane of the display. `take` is `{.` if the right argument is non-empty, and is an array of blanks otherwise. `frame` applies to an array of boxed tabular displays, and computes the overall display. `rc` computes the number of rows and columns in the display of the atoms in a plane.

`thornl` models `":`; `thbox` models `":` on a boxed array.

The following examples illustrate the inner workings of the model:

```
y =.(i.2 3);'abc';(i.4 1);(<2 2$'ussr');12;<+&.>i.2 2 3
y =. 2 3$y
x=.mat@th&.>y
```

```
   $&.>x                       rc x                   {rc x
┌────┬────┬────┐           ┌────┬─────┐          ┌─────┬─────┬─────┐
│2 5 │1 3 │4 1 │           │4 11│5 3 9│          │4 5  │4 3  │4 9  │
├────┼────┼────┤           └────┴─────┘          ├─────┼─────┼─────┤
│4 4 │1 2 │11 9│                                 │11 5 │11 3 │11 9 │
└────┴────┴────┘                                 └─────┴─────┴─────┘
```

```
   a =. 2 3 4$'abcdefghijklmnopqrstuvwx'
   a                        mat a                $a
abcd                     abcd                 2 3 4
efgh                     efgh
ijkl                     ijkl                    $ mat a
                                              7 4
mnop                     mnop
qrst                     qrst
uvwx                     uvwx
```

```
   topleft 3 4$'a'          (2 3;4 5) perim 6 10$'a'
┼─────                   ┌─────┬─────┐
│aaaa                    │aaaaaaaaaa │
│aaaa                    │aaaaaaaaaa │
│aaaa                    ├aaaaaaaaaa ┤
                         │aaaaaaaaaa │
                         │aaaaaaaaaa │
                         │aaaaaaaaaa │
                         └─────┴─────┘
```

```
   t=.({rc x)inside@:(take&.>)"2 x         (rc x) perim t
   t                                    ┌─────┬───┬──────────────┐
0 1 2│abc│0                             │0 1 2│abc│0             │
3 4 5│   │1                             │3 4 5│   │1             │
     │   │2                             │     │   │2             │
     │   │3                             │     │   │3             │
─────┼───┼──────────                    ├─────┼───┼──────────────┤
┌──┐ │12 │┌─┬──┬──┐                     │┌──┐ │12 │┌─┬──┬──┐     │
│us│ │   ││0│1 │2 │                     ││us│ │   ││0│1 │2 │     │
│sr│ │   │├─┼──┼──┤                     ││sr│ │   │├─┼──┼──┤     │
└──┘ │   ││2│3 │5 │                     │└──┘ │   ││3│4 │5 │     │
     │   │└─┴──┴──┘                     │     │   │└─┴──┴──┘     │
     │   │┌─┬──┬──┐                     │     │   │┌─┬──┬──┐     │
     │   ││6│7 │8 │                     │     │   ││6│7 │8 │     │
     │   │├─┼──┼──┤                     │     │   │├─┼──┼──┤     │
     │   ││9│10│11│                     │     │   ││9│10│11│     │
     │   │└─┴──┴──┘                     └─────┴───┴──────────────┘
```

6.3 Formatted Display

`x":y` is a literal representation of `y` specified by `x`. Positive elements of `x` specify fixed-point notation, while negative elements specify exponential notation. The left and right ranks are one; that is, lists in the arguments are independently formatted. The computation can be modelled as follows:

```
fmtexp   =. {&'++-'@* , _3&{.@('00'&,)@":@|
cexp     =. >:@(i.&'e') ({. , fmtexp@".@}.) ]
cminus   =. '-'&((e.&'_' # i.@#)@]})
larg     =. (+_20&*@(0&=))@-@(1&|)@|@".@(-.&' %e')
nsprintf =. larg@[ cexp@cminus@": ]
psprintf =. ".@(-.&' %f')@[ ($&' '@(0&=)@<.@[ , cminus@":) ]
sprintf  =. nsprintf`psprintf@.('f'&e.@[)

wd       =. <.@|
npstr    =. ' %- '&,@(,&'e')@(0.1&":)@(-1&<)@|
ppstr    =. *@wd }. ' %'&,@(,&'f')@(0.1&":)
pstr     =. npstr`ppstr@.(0&<:)

jexp     =. >:@(i.&'e') ({. , ":@".@(-.&' +')@}.) ]
jminus   =. '_'&((e.&'-' # i.@#)@]})
stars    =. ]`{.@.(*@[)`($&'*'@[)@.(*@[*.(<#))
c2j      =. stars ]`jexp@.('e'&e.)@jminus

lb       =. (0&=@wd *. 0&<:)@{.
thcell   =. (wd@[ <@c2j pstr@[ sprintf ])"0
thorn2   =. (lb@[ }. ;@:thcell) " 1
```

The model is divided into groups of verbs.

`sprintf` is a limited model of `sprintf` in the C library, applying to a string containing a single `%e` or `%f` conversion specification and to a single number. Thus, if `embrace=.('{'&,)@(,&'}')`, then:

```
embrace ' %0.3f' sprintf ^5               { 148.13}
embrace '%9.3f' sprintf ^_5               {     0.007}
embrace ' %- 0.3e' sprintf ^_5            {  6.738e-003}
embrace ' %- 9.3e' sprintf -^5            { -1.484e+002}
embrace ' %- 6.3e' sprintf -^_5           { -6.738e-003}
```

`pstr` applies to the left argument of `":` and produces the necessary left argument to `sprintf`. For example:

```
 x         embrace pstr x

_12        { %- 11.0e}
_7.3       { %- 6.3e}
_0.3       { %- 0.3e}
0          { %0.0f}
0.3        { %0.3f}
7.3        {%7.3f}
12         {%12.0f}
```

`c2j` and its constituents transform the result of `sprintf` to follow J conventions, in the treatment of negative signs (`jminus`), exponential notation (`jexp`), and overflow (`stars`).

`thorn2` is a model of the dyad `":`. It works by applying `thcell` to corresponding atoms of the arguments, producing a list of boxes; the leading blank of the razed result is then dropped or not, according to the value of `lb` on left argument.

7. Comparatives

Comparisons between finite numbers are *tolerant*, as defined in Bernecky [1977]:

```
x = y   if  (|x-y) <:!.0 qct * (|x)>.(|y)
```

(`<:!.0` means *exact* less than or equal.) That is, `x` and `y` are tolerantly equal if the smaller is on or within the circle centered at the larger, having radius `qct` times the magnitude of the larger. `qct`, comparison tolerance, is a real number between `0` and `2^_34` with a default value of `2^_44`; a non-default tolerance may be specified using the *fit* conjunction (`!.`). Tolerant relations can be modelled as follows:

```
teq     =. |@- <:!.0 qct&*@>.&|          in file ut.c
tlt     =. < !.0 *. ~:                   ut.c
tle     =. <:!.0 +. =                    ut.c
tfloor  =. <.!.0 ([ - -.@tle) ]          ut.c
tceil   =. <.!.0 ([ +    tlt) ]          ut.c

dsignum =. qct&<@| * 0&< - 0&>           ve.c
jsignum =. qct&<@| * (%|)                ve.c
```

`teq`, `tlt`, and `tle` model tolerant equal, less than, and less than or equal. `tfloor` and `tceil` model tolerant floor and ceiling. `dsignum` computes the tolerant signum of a real number; `jsignum` that of a complex number.

Additionally, some comparisons internal to the system are *fuzzy*. Fuzzy comparisons are like tolerant comparisons, but depend on the parameter `qfuzz`, having fixed value `2^_44`. Such comparisons are used to decide whether arguments are in the domain of certain verbs; for example, `(2 3+1e_14)$'abc'` is valid but `(2 3+1e_12)$'abc'` is not. Fuzzy comparisons can be modelled as follows:

```
int     =. (-2^31)&<: *. <&(2^31)
real    =. {.@+."0

feq     =. |@- <:!.0 qfuzz&*@>.&|              in file ut.c
freal   =. >:!.0/@((qfuzz,1)&*)/@|@+.          ut.c

BfromD =. ]`(1&=)@>(feq 1&=)                   k.c
IfromD =. ]`<.@.(int *. (feq<.))               k.c
DfromZ =. ]`real @. (feq real)                 k.c
```

The utility `int` tests for membership in the interval `-2^31` to `_1+2^31` inclusive. `real` produces the real part of a complex number. `feq` is 1 if its real arguments are equal within fuzz; `freal` is 1 if its complex argument is within fuzz of real. `BfromD`, `IfromD`, and `DfromZ` convert between types: boolean from real ("double"), integer from real, and real from complex.

8. Primitives

This chapter describes the primitives. Each entry has the spelling, class, program file, C object name, and notes on the implementation. The notes are mostly in the form of models written in J; the models are not necessarily optimal but are presented here because they are close to the implementation. The entries are ordered as in the dictionary, an order also shown in Appendix F on the back cover.

The following conventions and definitions apply:

adv	Adverb
conj	Conjunction
`m`	Left noun argument to an adverb or a conjunction
`n`	Right noun argument to a conjunction
`u`	Left verb argument to an adverb or a conjunction
`v`	Right verb argument to a conjunction

`pi`	π, 3.14159265358979...
`qct`	Comparison tolerance (default: `2^_44`)
`qrl`	Random link (initial value: `7^5`)
`rk`	An adverb that produces the ranks of its verb argument. For example, `%.rk` is `2 _ 2`

```
rank    =. #@$                 Rank
mt      =. 0&e.@$              1 if empty
id      =. [.+                 The identity adverb
type    =. 3!:0                Type
complex =. 16&=@type           1 if complex
boxed   =. 32&=@type           1 if boxed
pind    =. ]`]`+@.(*@])"0      n pind i are integers in i.n
pfill   =. [ ((i.@[-.]) , ]) pind
```

`n pfill p` converts `p` to a permutation of order `n` in the standard form; *i.e.* `(i.n) -: /:~ n pfill p`

`=` monad vb.c `sclass` `=. (((i.@#=])#]) =/ ])@,@(i.~)`

`=` dyad vb.c `eq`

For atoms `x` and `y`, `x=y` is 1 if `x` equals `y`; tolerant equality `|@- <:!.0 qct&*@>.&|` is used for numeric arguments. See 7 *Comparatives* and 3.3 *Atomic Verbs.*

`=.` p.c `isl` See 1.4 *Name Resolution.*

`=:` p.c `isg` See 1.4 *Name Resolution.*

`<` monad v.c `box`

`<y` has the following properties:

`0 = rank <y`	atomic
`y -: ><y`	*open* is the inverse of *box*
`y ~: <y`	*box* y differs from y
`32 = type <y`	the type is encoded as 32

`<` dyad vb.c `lt` `=. <!.0 *. ~:`

`<.` monad ve.c `floor1`

```
floor  =. <.!.0         NB. a function in the C library
dfloor =. (] - <) floor@(0.5&+)
zfl    =. floor@+.
inc    =. (1&<:@(+/) * 1 0&=@(>:!.0/)) @ (+. - zfl)
zfloor =. zfl j./@:+ inc
floor1 =. dfloor`zfloor @. complex " 0 " _
```

See also 3.3 *Atomic Verbs.*

`<.` dyad ve.c `minimum` See 3.3 *Atomic Verbs.*

`<:` monad ve.c `decrem` `=. -&1`

`<:` dyad vb.c `le` `=. <:!.0 +. =`

`>`	monad	v.c	`ope`	

```
mrk    =. >./@:(rank@>)
crk    =. mrk (-@[{.$&1@[,$@])&.> ]
crank  =. crk ($,)&.> ]
msh    =. >./@:($@>)
cshape =. <@msh {.&.> ]
mtp    =. >./@:((type*-.@mt)@>)
fill   =. >@({&(' ';(<$0);0))@(2 32&i.)
ctype  =.  (msh <@$ fill@mtp) (]`[@.(mt@]))&.> ]
ope    =. > @ cshape @ ctype @ crank
```

See Section II.B of the dictionary.

`>`	dyad	vb.c	`gt`	`=. -.@<:`
`>.`	monad	ve.c	`ceil1`	`=. <.&.-`
`>.`	dyad	ve.c	`maximum`	`=. <.&.-"0`
`>:`	monad	ve.c	`increm`	`=. 1&+`
`>:`	dyad	vb.c	`ge`	`=. -.@<`
`_`	noun	w.c	`coninf`	See 1.1 *Word Formation.*
`_.`	noun	w.c	`coninf`	See 1.1 *Word Formation.*
`_:`	monad	v.c	`inf1`	`=. _"_`
`_:`	dyad	v.c	`inf2`	`=. _"_`
`+`	monad	ve.c	`conjug`	See 3.3 *Atomic Verbs.*
`+`	dyad	ve.c	`plus`	See 3.3 *Atomic Verbs.*

`+.` monad vm.c `rect` `=. 9 11&o."0"_`

`+.` dyad ve.c `gcd` See 3.3 *Atomic Verbs.*

`+:` monad ve.c `duble` `=. +~`

`+:` dyad vb.c `nor` `=. -.@+.`

`*` monad ve.c `signum` `=. (%|) * >!.0&qct@|`

`*` dyad ve.c `tymes` See 3.3 *Atomic Verbs.*

`*.` monad vm.c `polar` `=. 10 12&o."0"_`

`*.` dyad ve.c `lcm` See 3.3 *Atomic Verbs.*

`*:` monad ve.c `square` `=. *~`

`*:` dyad vb.c `nand` `=. -.@*.`

`-` monad ve.c `negate` `=. 0&-`

`-` dyad ve.c `minus` See 3.3 *Atomic Verbs.*

`-.` monad ve.c `not` `=. 1&-`

`-.` dyad v.c `less`

```
dr   =. rank@] - 0&>.@<:@rank@[
res  =. (dr (*/@{. , }.) $@]) $ ,@]
less =. [`(([ -.@e. res) # [)@.((<: >:)&rank)
```

`-:` monad ve.c `halve` `=. %&2`

`-:` dyad vb.c **`match`**

`x -: y` if

`-:&(#@,)`	numbers of elements match; and
`-:&rank`	ranks match; and
`-:&$`	shapes match; and
`*./@(=&,)`	corresponding atoms match

`%` monad ve.c **`recip`** `=. 1&%`

`%` dyad ve.c **`divide`** See 3.3 *Atomic Verbs.*

`%.` monad vi.c **`minv`**

`minv` has two main constituents: **`qr`** computes the QR decomposition; **`rinv`** computes the inverse of a square upper triangular matrix.

```
pdt  =. +/ . *
en   =. 1&{@(,&1 1)@$

t=.0 0$''
t=.t, 'n  =. en y.'
t=.t, 'm  =. >.-: n'
t=.t, 'a0 =. m{."1 y.'
t=.t, 'a1 =. m}."1 y.'
t=.t, 't0 =. qr a0'
t=.t, 'q0 =. >@{. t0'
t=.t, 'r0 =. >@{: t0'
t=.t, 'c  =. (+|:q0) pdt a1'
t=.t, 't1 =. qr a1 - q0 pdt c'
t=.t, 'q1 =. >@{. t1'
t=.t, 'r1 =. >@{: t1'
t=.t, '(q0,.q1);(r0,.c),(-n){."1 r1'
q2   =. t : ''
norm =. (%:@pdt +)@,
qr   =. q2`((% ;&,. ,~@en@[ $ ]) norm) @. (1&>:@en)
```

```
x=.0 0$''
x=.x, 'n =. #y.'
x=.x, 'm =. >.-: n'
x=.x, 'ai =. rinv (m,m){.y.'
x=.x, 'di =. rinv (m,m)}.y.'
x=.x, 'b  =. (m,m-n){.y.'
x=.x, 'bx =. - ai pdt b pdt di'
x=.x, '(ai,.bx),(-n){."1 di'
r4   =. x : ''
rinv =. r4`% @. (1&>:@#)

minv =. (|.@$ ($,) (rinv@] pdt +@|:@[)&>/@qr) " 2
```

%. dyad vi.c `mdiv` `=. (%.@] +/ . * [) " _ 2`

%: monad vm.c `sqroot` `=. 2&%:`

%: dyad vm.c `root` `=. (] ^ %@[)"0`

^ monad vm.c `expn1`

```
exp    =. ^         NB. a function in the C library
sin    =. 1&o.
cos    =. 2&o.
zexp   =. ((^@[ * cos@])j.(^@[ * sin@]))/@+.
expn1 =. exp`zexp @. complex " 0 " _
```

^ dyad vm.c `expn2` `=. ^@(^.@[ * ])"0`

^. monad vm.c `logar1`

```
atan2  =. 12&o.@j.     NB. a function in the C library
logar1 =. (^.@| j. atan2/@+.)"0"_
```

^. dyad vm.c `logar2` `=. %~&^."0`

^: conj. cp.c `powop`

`$`	monad	vs.c	**shape**	See 2.1 *Arrays*.
`$`	dyad	vs.c	**reitem**	`=. ((, }.@$) ($,) ])"1 _`
`$.`		cx.c	**ensuite**	
`$:`	monad	p.c	**self1**	
`$:`	dyad	p.c	**self2**	
`~`	adverb	a.c	**swap**	

`m~` is a reference to the verb named by `m`. See 1.4 *Name Resolution*.
`u~` is `(] u [)"(_,2 1{u rk)`.

`~.`	monad	v.c	**nub**	`=. ~: # ]`
`~:`	monad	vb.c	**nubsieve**	`=. i.@# = i.~`
`~:`	dyad	vb.c	**ne**	`=. -.@=`
`|`	monad	ve.c	**mag**	`=. (>.-)`(%:@*+)@.complex`
`|`	dyad	ve.c	**residue**	See 3.3 *Atomic Verbs*.
`|.`	monad	vs.c	**reverse**	`=. {~ i.@-@#`
`|.`	dyad	vs.c	**rotate**	

```
rotate =. ]`(((i.@]-]-|~)#){])@.(*@rank@])"0 _
```

`|:`	monad	vs.c	**cant1**	`=. i.@-@rank |: ]`
`|:`	dyad	vs.c	**cant2**	

```
mask  =. =/ i.@>:@(>./)
vec   =. >@{@:(i.&.>)@((<./ .+) _&*@-.)
ind   =. vec +/ .*  (#. |:)
```

```
canta =. ($@] ind mask@[) { ,@]

en    =. - #@;
ci    =. (/:@pfill ;) { i.@en , en + (#@> # i.@#)@]
cant2 =. ((rank@] ci [) canta ]) " 1 _
```

See Hui [1987] 3.1. `canta` is dyadic transpose in APL.

. conj. c.c `dot`

```
minors =. (0 0 1&}.) @ (1&([\.))
col    =. {:@(1&,)@$
```

monad `v/@,`(u@,)`({."1 u . v$:@minors)@.(0 1&i.@col)"2`

dyad `x u . v y` is

`x u@(v"(lv,lv>.<:#$y)"(1+lv,_)) y [ lv=.1{v rk`

See Hui [1987] 3.3.

.. conj c.c `even` `=. [.`(-:@:+[.)`& \`

.: conj c.c `odd` `=. [.`(-:@:-[.)`& \`

: conj. cx.c `colon`

:. conj. c.c `obverse` See 3.4 *Obverses*.

, monad vs.c `ravel`

`,y` has the following properties:

```
1       -: rank ,y
(*/$y)  -: # ,y
y       -: ($y)$ ,y
```

, dyad vs.c `over`

,. monad vs.c `table` `=. (#, */@}.@$) $ ,`

,. dyad vs.c `overr` `=. ,"_1`

```
,:  monad   vs.c   lamin1       =. 1&,@$ $ ,
```

```
,:  dyad    vs.c   lamin2
```

```
v00 =.     [ ,  ,.@]
v01 =.     [ ,  ,:@]
v10 =.  ,:@[ ,     ]
v11 =.  ,:@[ ,  ,:@]
lamin2 =. v00`v01`v10`v11 @. (#.@*@,&rank)
```

```
;   monad   v.c    raze
```

The monad `;` is `>@(,&.>/)@,` . This is an O(n^2) algorithm. The implementation uses a faster method — copying items from the argument into a pre-allocated space — when `1&>:@#@~.@:(type@>)` and `1&>:@(>./ - <./)@:(rank@>)`.

```
;   dyad    v.c    link         =. <@[ , <`]@.boxed@]
```

```
;.  conj.   cc.c   cut
```

```
cut_1 =. (&}.) (;.1)
cut2  =. (&|.) (;.1) (&.|.)
cut_2 =. (&}:) (;.2)
```

See Hui [1987] 3.2.

;: monad w.c `words` See 1.1 *Word Formation.*

```
#   monad   vs.c   tally        =. {.@(,&1)@$
```

```
#   dyad    vs.c   repeat       =. ;@(<@($,:)"_1) " 1 _
```

```
#.  monad   ve.c   base1        =. 2&#."1
```

`#.` dyad ve.c `base2`

```
ext   =. (#@] # [)`[ @. (*@rank@[)
base2 =. (*/\.@}.@(,&1)@ext +/ . * ]) " 1
```

`#:` monad ve.c `abase1`

```
max    =. >./@|@,
bits   =. >:@<.@(2&^.)@(1&>.)
abase1 =. #:~ $&2@bits@max
```

`#:` dyad ve.c `abase2` `=. ([ | ([%~]-|)/\.@}.@,)"1 0`

`!` monad vm.c `fact`

`!` dyad vm.c `outof`

```
case  =. #. @ (0&>*.(=<.)) @ ([,],-~)
f000  =. !@] % !@[ * !@-~
f001  =. 0:
f010  =. 'domain error'"0
f011  =. _1&^@[ * [ ! (->:)
f100  =. 0:
f101  =. 'can not happen'"0
f110  =. _1&^@-~ * !&|&>:~
f111  =. 0:
outof =. f000`f001`f010`f011`f100`f101`f110`f111 @.case"0
```

See *SHARP APL Reference Manual*, pp. 131-133 (Berry [1979]) and 3.3 *Atomic Verbs*.

`!.` conj. cf.c `fit` See 3.4 *Variants*.

`!:` conj. x.c `foreign`

The `!:` conjunction takes integer scalar left and right arguments, and produces verbs. (One exception: `5!:0` is an adverb.) These verbs behave like other verbs; in particular, they have intrinsic ranks, may be assigned names, and may serve as arguments to adverbs and conjunctions. Where these verbs take names as arguments (file names, workspace names, or object names), the names are always boxed, and the verb rank is 0.

See Appendix E for the names of functions which implement the various cases of `m!:n`.

`/` adverb a.c **`slash`**

`/.` adverb ap.c **`sldot`**

```
key      =. (@#) (=@[`) (`]) \
osub     =. >@]`(>@[ >@:{ ]) @. (*@#@])
oind     =. (+/&i./ </.&, i.)@(2&{.)@(,&1 1)@$
oblique =. (@(osub"0 1)) (oind`) (`(,@(<"_2))) \
sldot    =. id (oblique : key)
```

`/:` monad vg.c **`grade1`**

```
qsort     NB. a function in the C library
arg     =. <"_1 ,. ]&.>@i.@#
grade1 =. >@{:"1 @ qsort @ arg
```

`/:` dyad vg.c **`grade2`** `=. {~ /:`

`\` adverb ap.c **`bslash`**

```
base   =. 1&>.@-@[ * i.@em
iind   =. base ,. |@[ <. en - base
seg    =. ((+i.)/@[ { ])"1 _
infix  =. (@seg) (iind `) (`]) \ ("0 _)
prefix =. (@{.) (>:@,.@i.@#`) (`]) \
bslash =. id (prefix : infix)
```

`\.` adverb ap.c **`bsdot`**

```
en     =. #@]
em     =. (en >.@% 1&>.@|@[)`(en 0&>.@>:@- [) @. (0&<:@[)
```

```
kay     =. en`em @. (0&<@[)
omask   =. (em,en) $ ($&0@|@[ , $&1@kay)
outfix =. (@#) (omask`) (`]) \ ("0 _)
suffix =. (@}.) ~ (`(,.@i.@#)) \
bsdot   =. id (suffix : outfix)
```

\: monad vg.c `dgrade1`

```
qsort        NB. a function in the C library
darg     =. <"_1 ,. -&.>@i.@#
dgrade1 =. |.@- @: (>@{:"1) @ qsort @ darg
```

\: dyad vg.c `dgrade2 =. {~ \:`

[monad v.c `left1`

[dyad v.c `left2`

[. conj. c.c `lev`

] monad v.c `right1`

] dyad v.c `right2`

]. conj. c.c `dex`

{ monad vs.c `catalog`

```
count    =. */@$@>
prod     =. */\.@}.@(,&1)
copy     =. */@[ $&> prod@[ (#,)&.> ]
catalog =. (;@:($&.>) $ count <"1@|:@copy ]) " 1
```

See Hui [1987] 2.1.

`{` dyad vs.c **from**

```
ifrom =. (#@] pind [) >@{ <"_1@]
afi   =. pind`(i.@[-.(pind>)) @. (boxed@])
afrom =. ($@] #. $@] >@{@:(afi&.>) >@[) ifrom ,@]
from  =. ifrom`afrom @. (boxed@[) " 0 _
```

See Hui [1987] 2.2.

`{.` monad vs.c **head** `=. 0&{`

`{.` dyad vs.c **take**

```
fill  =. >@({&(' ';(<$0);0)) @ (2 32&i.@(type*-.@mt))
pad   =. fill@] $~ (|@[ - #@]) 0} $@]
ti    =. i.@-@[ + [ + #@]
case  =. 0&<:@[ #.@, |@[ > #@]
itake =. (ti{])`(],~pad)`(i.@[{])`(],pad) @. case
taker =. '':'({.x.) itake"({:x.) y.'
raise =. (1"0@[ $ ])`]@.(*@rank@])
larg  =. <@,"(0) _&(0})@-@i.@#
targ  =. larg@[ , <@raise
take  =. >@(taker&.>/)@targ " 1 _
```

`{:` monad vs.c **tail** `=. _1&{`

`}` adverb a.c **rbrace**

```
m}          m"_}
monad u}     , {~ i.@}.@$ + */@}.@$ * # pind u
dyad  u}     (i.@$@] (|.@,&,i.[) pind@u) { ] |.@,&, $@u $ [
```

See Hui [1987] 2.4.

`}.` monad vs.c **behead** `=. 1&}.`

}. dyad vs.c `drop`

```
pi   =. 0&< @[ * 0&<.@-
ni   =. 0&>:@[ * 0&>.@+
di   =. ({.~ rank) (pi + ni) $@]
drop =. (di {. ])"1 _
```

}: monad vs.c `curtail` `=. _1&}.`

" conj cr.c `qq` See 3.2 *Rank.*

". monad v.c `exec1`

". dyad v.c `exec2`

": monad f.c `thorn1` See 6 *Display.*

": dyad f.c `thorn2` See 6.3 *Formatted Display.*

` conj. cg.c `tie`

`ar`	an adverb that produces atomic representation of a verb
`` m`n ``	`m,n`
`` m`v ``	`m,(v ar)`
`` u`n ``	`(u ar),n`
`` u`v ``	`(u ar),(v ar)`

`: conj. cg.c `evger`

@ conj. c.c `atop`

@. conj. c.c `agenda`

For argument cells `x` and `y` of `m@.v`:

`m@.v y` is ``((v y){m)`:0 y``

`x m@.v y` is ``x ((x v y){m)`:0 y``

`@:` conj c.c **atco** `=. [.@("_)`

`&` conj. c.c **amp**

`m&v`	Empty dyadic domains; infinite monadic rank.
`u&n`	Empty dyadic domains; infinite monadic rank.
`u&v`	`u@v : (v@[ u v@]) " ({.v rk)`

`&.` conj. c.c **under** `=. (] . (^:_1))@&`

`&:` conj. c.c **ampco** `=. [.&("_)`

`?` monad v.c **roll**

```
tick =. [ <.@%~ (* 'qrl=:(<:2^31)|(7^5)*qrl':'')@]
roll =. (<:2^31)&tick"0
```

See *SHARP APL Reference Manual*, p. 126 (Berry [1979]).

`?` dyad v.c **deal**

```
tick =. [ <.@%~ (* 'qrl=:(<:2^31)|(7^5)*qrl':'')@]
step =. <@~.@((+ (2^31)&tick)/\)@[ C. ]
arg  =. <@i.@-@] ,~ i.@-@[ ([ ,&.> -~) ]
deal =. ([ {. >@(step&.>/)@arg)"0
```

See *SHARP APL Reference Manual*, p. 178 (Berry [1979]).

`)` cx.c **label**

`a.` noun j.c **alp** The 256-letter ASCII alphabet.

`A.` monad vp.c **adot1**

```
ord   =. >:@(>./)
base  =. >:@i.@-@#
rfd   =. +/@({.>}.)\.
dfr   =. /:^:2@,/

adot1 =. (base #. rfd)@((ord pfill ])`C.@.boxed) " 1
```

```
A.  dyad      vp.c   adot2        =. dfr@(base@] #: [) { ]

b.  adverb    a.c    bool

    tt   =. i.@rank |: {&(#:i.16)
    bool =. '0&$: : (+:@[ {&(tt x.)@+ ]) " _ 0 0' : 1

c.  monad     vm.c   eig1         Not yet available

c.  dyad      vm.c   eig2         Not yet available

C.  monad     vp.c   cdot1

    ac    =. (, i. ]) { 1&|.@[ , ]
    dfc   =. >@(ac&.>/)@(pind&.> , <@i.@[)
    bc    =. <@(([ i. >./) |. ])@~.
    cfd   =. ~.@(/: {.&>)@:(bc"1)@|:@({/\)@(,~@[ $ pfill)
    cdot1 =. (ord cfd ])`(ord@; dfc ])@.boxed " 1

C.  dyad      vp.c   cdot2

    cdot2 =. ((#@] pfill`dfc@.(boxed@]) [) { ]) " 1 _

e.  monad     vb.c   razein       =. e.~&> <@;

e.  dyad      vb.c   eps          =. i.~ < #@]

E.  dyad      vb.c   ebar         Not yet available

f.  adverb    a.c    fix

i.  monad     v.c    iota

    rev  =. '' : '|."x. y.'
    ineg =. # - 0&> # i.@#
    iota =. > @ (rev&.>/) @ (<"0@ineg , (<@$ i.@(*/))@|) " 1
```

`i.` dyad vh.c `indexof`

If `x` and `y` are literal lists, then `x i. y` is `x ciof y`:

```
map  =. '(i.-#y.) (a.i.|.y.)}256$#y.' : ''
ciof =. a.&i.@] { map@[
```

Otherwise, if `x` and `y` are not floating point or complex numbers, or if the comparison tolerance `qct` is zero, a straightforward hashed algorithm is used.

Otherwise, if `x` or `y` are floating point numbers and `qct` is nonzero, an algorithm due to Arthur Whitney is used:

`bit x` Convert a floating point `x` into a Boolean vector
`tib b` Convert a Boolean vector `b` into a floating point number
`hash b` Hash function on a Boolean vector `b`

There exists a Boolean `mask` with a minimum number of ones such that `tib mask*.bit x` is within `qct` of `x`; the actual mask used in the algorithm may have fewer number of ones. For each `xi` of `x`, compute `hash mask*.bit xi`; for each `yj` of `y`, compute:

```
hl =. hash mask*.bit yj*1-qct
hr =. hash mask*.bit yj*1+qct
```

Look for `hl` and `hr` in the list of hashed `xi`'s. In other words, if `hash` were a perfect hash, then for `th=.hash@(mask&*.)@ bit"0`, `x i. y` is `((th x)i.th y*1-qct)<.((th x)i.th y*1+qct)`.

`j.` monad vm.c `jdot1` `=. 0j1&*`

`j.` dyad vm.c `jdot2` `=. (+ j.)"0`

`NB.` w.c `wordil` See 1.1 *Word Formation*.

`o.` monad vm.c `pix` `=. pi&*`

```
o. dyad      vm.c   circle
```

```
sin     =. 1&o.          NB. a function in the C library
cos     =. 2&o.          NB. a function in the C library
sinh    =. 5&o.          NB. a function in the C library
cosh    =. 6&o.          NB. a function in the C library

cir0    =. 1&+    %:@* 1&-
zp4     =. -&0j1 %:@* +&0j1
zp8     =. 0j1&+ %:@* 0j1&-
zm4     =. +&1 * -&1 %:@% +&1
real    =. -:@(++)
imag    =. %&0j2@(-+)
zarc    =. 0j_1&*@^.@*`0: @. (0&=)

zsin    =. ((sin@[ * cosh@]) j. (  cos@[ * sinh@]))/@+.
zcos    =. ((cos@[ * cosh@]) j. (-@sin@[ * sinh@]))/@+.
ztan    =. zsin % zcos
zsinh   =. zsin&.j.
zcosh   =. zcos@j.
ztanh   =. ztan&.j.

zasin   =. zasinh&.j.
zacos   =. (-:pi)&-@zasin
zatan   =. zatanh&.j.
zasinh  =. (^.@+ zp4)`($:&.-) @. (0&>@real)
zacosh  =. ]`(j.@|@imag)@.(0&>@real) @ (^.@+ zm4)
zatanh  =. 1&+ -:@^.@% 1&-

cirp    =. (cir0@])`(zsin@])`(zcos@])`(ztan@])`(zp4@])`
           (zsinh@])`(zcosh@])`(ztanh@])`(zp8@])`
           (real@])`(|@])`(imag@])`(zarc@]) @. [
cirm    =. (cir0@])`(zasin@])`(zacos@])`(zatan@])`(zm4@])`
           (zasinh@])`(zacosh@])`(zatanh@])`(-@zp8@])`
           ]`(+@])`(j.@])`(r.@]) @. (|@[)
circle =. cirp`cirm @. (0&>@[) " 0
```

See *Handbook of Mathematical Functions*, Chapter 4 (Abramowitz and Stegun [1964]).

`p.`	monad	vm.c	`poly1`	Not yet available.
`p.`	dyad	vm.c	`poly2`	Not yet available.
`r.`	monad	vm.c	`rdot1`	`=. ^@j.`
`r.`	dyad	vm.c	`rdot2`	`=. (* r.)"0`
`x.`		cx.c	`xd`	
`y.`		cx.c	`xd`	
`0:`	monad	v.c	`zero1`	`=. 0"_`
`0:`	dyad	v.c	`zero2`	`=. 0"_`
`1:`	monad	v.c	`one1`	`=. 1"_`
`1:`	dyad	v.c	`one2`	`=. 1"_`

Appendix A. Incunabulum

One summer weekend in 1989, Arthur Whitney visited Ken Iverson at Kiln farm and produced — on one page and in one afternoon — an interpreter fragment on the AT&T 3B1 computer. I studied this interpreter for about a week for its organization and programming style; and on Sunday, August 27, 1989, at about four o'clock in the afternoon, wrote the first line of code that became the implementation described in this book.

Arthur's one-page interpreter fragment is as follows:

```
typedef char C;typedef long I;
typedef struct a{I t,r,d[3],p[2];}*A;
#define P printf
#define R return
#define V1(f) A f(w)A w;
#define V2(f) A f(a,w)A a,w;
#define DO(n,x) {I i=0,_n=(n);for(;i<_n;++i){x;}}
I *ma(n){R(I*)malloc(n*4);}mv(d,s,n)I *d,*s;{DO(n,d[i]=s[i]);}
tr(r,d)I *d;{I z=1;DO(r,z=z*d[i]);R z;}
A ga(t,r,d)I *d;{A z=(A)ma(5+tr(r,d));z->t=t,z->r=r,mv(z->d,d,r);
 R z;}
V1(iota){I n=*w->p;A z=ga(0,1,&n);DO(n,z->p[i]=i);R z;}
V2(plus){I r=w->r,*d=w->d,n=tr(r,d);A z=ga(0,r,d);
 DO(n,z->p[i]=a->p[i]+w->p[i]);R z;}
V2(from){I r=w->r-1,*d=w->d+1,n=tr(r,d);
 A z=ga(w->t,r,d);mv(z->p,w->p+(n**a->p),n);R z;}
V1(box){A z=ga(1,0,0);*z->p=(I)w;R z;}
V2(cat){I an=tr(a->r,a->d),wn=tr(w->r,w->d),n=an+wn;
 A z=ga(w->t,1,&n);mv(z->p,a->p,an);mv(z->p+an,w->p,wn);R z;}
V2(find){}
V2(rsh){I r=a->r?*a->d:1,n=tr(r,a->p),wn=tr(w->r,w->d);
 A z=ga(w->t,r,a->p);mv(z->p,w->p,wn=n>wn?wn:n);
 if(n-=wn)mv(z->p+wn,z->p,n);R z;}
V1(sha){A z=ga(0,1,&w->r);mv(z->p,w->d,w->r);R z;}
V1(id){R w;}V1(size){A z=ga(0,0,0);*z->p=w->r?*w->d:1;R z;}
pi(i){P("%d ",i);}nl(){P("\n");}
```

```
pr(w)A w;{I r=w->r,*d=w->d,n=tr(r,d);DO(r,pi(d[i]));nl();
 if(w->t)DO(n,P("< ");pr(w->p[i]))else DO(n,pi(w->p[i]));nl();}

C vt[]="+{~<#,";
A(*vd[])()={0,plus,from,find,0,rsh,cat},
 (*vm[])()={0,id,size,iota,box,sha,0};
I st[26]; qp(a){R  a>='a'&&a<='z';}qv(a){R a<'a';}
A ex(e)I *e;{I a=*e;
 if(qp(a)){if(e[1]=='=')R st[a-'a']=ex(e+2);a= st[ a-'a'];}
 R qv(a)?(*vm[a])(ex(e+1)):e[1]?(*vd[e[1]])(a,ex(e+2)):(A)a;}
noun(c){A z;if(c<'0'||c>'9')R 0;z=ga(0,0,0);*z->p=c-'0';R z;}
verb(c){I i=0;for(;vt[i];)if(vt[i++]==c)R i;R 0;}
I *wd(s)C *s;{I a,n=strlen(s),*e=ma(n+1);C c;
 DO(n,e[i]=(a=noun(c=s[i]))?a:(a=verb(c))?a:c);e[n]=0;R e;}

main(){C s[99];while(gets(s))pr(ex(wd(s)));}
```

Appendix B. Program Files

a.c	adverbs
ai.c	adverbs — inverse and identity
ap.c	adverbs — partitions
au.c	adverbs — utilities
c.c	conjunctions
cc.c	conjunctions — cuts
cf.c	conjunctions — fit
cg.c	conjunctions — gerunds
cp.c	conjunctions — power
cr.c	conjunctions — rank
ct.c	conjunctions — trains
cx.c	conjunctions — explicit definition
f.c	format (display)
i.c	initialization
io.c	input/output
j.c	`main` and global variables
k.c	conversion
m.c	memory management
p.c	parsing
pc.c	parsing — tacit conjunction translator
pv.c	parsing — tacit verb translator
r.c	representation
rt.c	tree representation
s.c	symbol table
t.c	tables
u.c	utilities
ut.c	utilities — tolerant and fuzzy comparison
v.c	verbs
vb.c	verbs — boolean
ve.c	verbs — elementary functions

vg.c	verbs — grades
vh.c	verbs — hashed indexing
vi.c	verbs — matrix inverse and matrix divide
vm.c	verbs — mathematical functions
vp.c	verbs — permutation
vs.c	verbs — selection & structural
vz.c	verbs — complex functions
w.c	word formation
x.c	external, experimental, and extra
xf.c	external — files
xs.c	external — scripts
xw.c	external — workspaces

a.h	adverbs and conjunctions
io.h	input/output
j.h	global definitions
jc.h	character definitions
je.h	`extern` declarations
jt.h	types
p.h	parsing
v.h	verbs
x.h	external, experimental, and extra

lj.c	LinkJ
lj.h	LinkJ
main.c	LinkJ example

Appendix C. The LinkJ Interface

LinkJ is a set of object modules which together offer the full capability of J while allowing links to other compiled routines and libraries. It is possible to call J from C and to call C from J. The interface consists of the following definitions, functions, and variables:

```
typedef char B;
typedef char C;
typedef long I;
typedef struct{I t,c,n,r,s[1];}*A;
typedef A (*AF)();
```

```
C jinit(void);                    B asgn;
A jx(C*s);                        C jerr;
A jpr(A x);
A jma(I t,I n,I r);
C jfr(A x);
A jset(C*name,A x);
C jc(I k,AF*f1,AF*f2);
```

`A` is the C data type of an array. The parts are the type, reference count, number of elements in the array, rank, shape, and the array elements, in a contiguous segment of memory. (Array types are boolean, literal, integer, floating point, complex, and boxed. See file lj.h.) `AF` typifies a function which accepts one or more array arguments, and returns an array result; that is, `AF` is the C data type of a verb.

`jinit` initializes J. `jx` applies to a 0-terminated string representing a sentence, and returns the array result of executing the sentence; the global variable `asgn` is 1 if the last operation is assignment. When an error is encountered in an interface function, the result is 0, and the global variable `jerr` contains an error number as defined in file lj.h. For example:

```
jinit();
p=jx("a=.i.3 4");
q=jx("+/,a");
```

`p` is a 3 by 4 table of the integers from 0 to 11, and `q` is the atom 66. The space occupied by the result of `jx` is reused the next time `jx` is called.

`jpr(x)` prints array `x` on the standard output; the result is `x` itself.

`jma(t,n,r)` allocates memory for an array of type `t` having `n` elements and rank `r`. (The shape and the elements must then be filled.) `jfr` frees an array previously allocated by `jma`. Array arguments and results must use space managed by `jma` and `jfr`.

`jset(name,x)` assigns a value to a global name (as in the copula `=:`). `name` is a 0-terminated string; `x` is an array. The result of `jset` is `x` itself. `jx(name)` returns the referent of a name.

The preceding functions allow calling J from C. The following facilities allow calling C from J. A new case of the `!:` foreign conjunction is defined: `10!:k` is a verb whose definition is controlled by `jc`, a function written by the user, as follows:

```
C jc(I k,AF*f1,AF*f2){
 switch(k){
 /* k:  index                                       */
 /* f1: pointer to monad (or NULL if no monad)      */
 /* f2: pointer to dyad (or NULL if no dyad)        */
 /* result is 0 if there is an error, nonzero if no error */
}}
```

`10!:k` invokes `jc(k,&f1,&f2)`, wherein (presumably depending on `k`) `*f1` is assigned a pointer to a monadic function and `*f2` a pointer to a dyadic function. The result of `10!:k` is a verb like any other; in particular, it may be assigned a name and may serve as argument to adverbs and conjunction; and when it is invoked with arguments the functions assigned to `*f1` and `*f2` are invoked with those arguments.

File main.c contains an example of using LinkJ. It has a `main` function which repeats the following steps, *ad infinitum*:

- Get a line of input from the terminal;
- Execute the line;
- Print the error number if an error occurred, or the result if the last operation was not assignment.

(To terminate, enter CTRL D or execute `0!:55 ''`.) As well, main.c has an example of using `jc`: `10!:0 y` computes `#,y`, the number of elements in array `y`; and `x 10!:0 y` computes `x{.,y`, the first `x` elements of integer array `y`.

Appendix D. Compiling

There is a single set of program files; machine and compiler dependencies are handled by conditional compilation (`#if` preprocessor statements). The following names must be defined in file j.h:

```
#define SYS            SYS_???

#define LINKJ          0
#define WATERLOO       0

#define SYS_ANSILIB    0
#define SYS_LILENDIAN  0
#define SYS_SESM       0
#define SYS_UNIX       0
```

`SYS` identifies the current system, and must be one of the `SYS_*` names defined at the beginning of j.h — `SYS_PCAT`, `SYS_MACINTOSH`, `SYS_SUN4`, etc. The inclusion of a system name in the list of `SYS_*` names does not imply that the program files would compile in that system, nor that the compiled result would work. (The file status.doc has a list of working systems.)

`LINKJ` and `WATERLOO` are Boolean flags. `LINKJ` is set to 1 to generate the LinkJ modules. (See Appendix C *The LinkJ interface*.) `WATERLOO` is set to 1 when compiling on machines at the University of Waterloo using the MFCF library organization.

The other `SYS_*` names are Boolean masks, used as `(SYS & SYS_UNIX)`. In compiling on a machine from the existing list, these masks can remain unchanged; otherwise, in porting to a new system, it is easiest just to set a mask to 0 or 1 as appropriate. `SYS_ANSILIB` selects systems using the ANSI C library organization. `SYS_LILENDIAN` selects "little endian" (reverse byte order) systems. The PC (Intel 80x86) line of machines are little endian. `SYS_SESM` selects systems using the J session manager. (The session manager is not publicly available, so `SYS_SESM` should be set to 0.) `SYS_UNIX` selects UNIX systems.

Object modules must be linked with the C math library to generate an executable module. The procedure varies from system to system; the command `cc *.o -lm -o j` works under UNIX.

Bibliography

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, 1964 6.

Bernecky, R., *Comparison Tolerance*, SHARP APL Technical Notes 23, 1977 6 10.

Bernecky, R., G. Bezoff, and M. Symes, *APL Workspace Transfer*, SHARP APL Technical Notes 22, Revision 4, 1981 7 15.

Bernecky, R., and R.K.W. Hui, *Gerunds and Representations*, APL91, APL Quote-Quad, Volume 21, Number 4, 1991 8.

Berry, P.C., *SHARP APL Reference Manual*, I.P. Sharp Associates, 1979 3; Additions and Corrections, 1981 6.

Falkoff, A.D., and K.E. Iverson, *APL\360 User's Manual*, IBM Corporation, 1968 8.

Falkoff, A.D., and K.E. Iverson, *The Design of APL*, IBM Journal of Research and Development, Volume 17, Number 4, 1973 7.

Falkoff, A.D., and K.E. Iverson, *The Evolution of APL*, ACM SIGPLAN Notices, Volume 13, Number 8, 1978 8.

Hui, R.K.W., *Some Uses of* { *and* }, APL87, APL Quote-Quad, Volume 17, Number 4, 1987 5.

Hui, R.K.W., *An Implementation of J*, Iverson Software Inc., 1992 1 27.

Hui, R.K.W., K.E. Iverson, and E.E. McDonnell, *Tacit Definition*, APL91, APL Quote-Quad, Volume 21, Number 4, 1991 8.

Hui, R.K.W., K.E. Iverson, E.E. McDonnell, and A.T. Whitney, *APL\?*, APL90, APL Quote-Quad, Volume 20, Number 4, 1990 7.

Iverson, K.E., *A Programming Language*, Wiley, 1962 5.

Iverson, K.E., *Operators and Functions*, Research Report #RC7091, IBM Corporation, 1978 4 26.

Iverson, K.E., *Notation as a Tool of Thought*, Communications of the ACM, Volume 23, Number 8, 1980 8.

Iverson, K.E., *Rationalized APL*, I.P. Sharp Associates, 1983 1 6.

Iverson, K.E., *APL87*, APL87, APL Quote-Quad, Volume 17, Number 4, 1987 5.

Iverson, K.E., *A Dictionary of APL*, APL Quote-Quad, Volume 18, Number 1, 1987 9.

Iverson, K.E., *Arithmetic*, Iverson Software Inc., 1991.

Iverson, K.E., *ISI Dictionary of J*, Version 4, Iverson Software Inc., 1991.

Iverson, K.E., *Programming in J*, Iverson Software Inc., 1991.

Iverson, K.E., *Tangible Math*, Iverson Software Inc., 1991.

Iverson, K.E., *A Personal View of APL*, IBM Systems Journal, Volume 30, Number 4, 1991 12.

Iverson, K.E., *An Introduction to J*, Iverson Software Inc., 1992 1 8.

Iverson, K.E., and E.E. McDonnell, *Phrasal Forms*, APL89, APL Quote-Quad, Volume 19, Number 4, 1989 8.

Iverson, K.E., and A.T. Whitney, *Practical Uses of a Model of APL*, APL82, APL Quote-Quad, Volume 13, Number 1, 1982 9.

Kernighan, B.W., and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

McDonnell, E.E., and J.O. Shallit, *Extending APL to Infinity*, APL80, North-Holland Publishing Company, 1980.

McIntyre, D.B., *Mastering J*, APL91, APL Quote-Quad, Volume 21, Number 4, 1991 8.

McIntyre, D.B., *Language as an Intellectual Tool: From Hieroglyphics to APL*, IBM Systems Journal, Volume 30, Number 4, 1991 12.

Glossary and Index

An explanation is provided for every name in the program files. Each entry consists of a name, a program file name, a section number in this book (if any), and an explanation. The following conventions and abbreviations apply:

Names spelled with majuscules denote defined types (**typedefs**) or **#defined** constants and macros; those spelled with minuscules denote C functions and variables. Names localized in functions are omitted.

*	A sequence of letters; everywhere
adv	Adverb
arg	Argument
char	Character
conj	Conjunction
m	Left noun argument to an adverb or a conjunction
n	Right noun argument to a conjunction
u	Left verb argument to an adverb or a conjunction
v	Right verb argument to a conjunction
x	Left argument
y	Right argument

a	*		left argument
A	jt.h	2.1	typedef array
AA	p.h		typedef translat. action
abase1	ve.c	8	monad #:
abase2	ve.c	8	dyad #:
ABS	j.h		absolute value
AC	jt.h	2.1	A reference count
ACTION	p.h	1.2	parser action header
ac1	au.c	4	monad from C function
ac2	au.c	4	dyad from C function
ADERIV	a.h	4	derive verb from adverb
adot1	vp.c	8	monad A.
adot2	vp.c	8	dyad A.
adv	p.c	1.2	parser action
ADV	jt.h	2.2	A type
advform	ct.c	1.3	derive an adv from a conj
aeq	vb.c		dyad = A subcase
AF	jt.h	3.1	typedef APL function
afi	vs.c	8	dyad { standard index
aform	ct.c		bonded conjunction
afrom	vs.c	8	dyad { A subcase
agenda	cg.c	8	@.
AH	jt.h	2.1	A no. of header words
aii	u.c		no. of atoms in an item
all1	u.c		1 if all ones
alp	j.c	8	a.
alt	ai.c		$&1 _1@#
amp	c.c	4	&
ampco	c.c	8	&:
AN	jt.h	2.1	A number of atoms
ane	vh.c		fne A subcase
ANY	jt.h	2.2	A composite type
appf	xs.c		application file handler
appf1	xs.c		appf subfunction
apv	u.c	2.1	arithmetic progression
AR	jt.h	2.1	A rank
arep	r.c	5.1	atomic representation
aro	r.c	5.1	atomic rep, opened
arx	x.c	5.1	monad 5!:1
AS	jt.h	2.1	A shape
asgn	j.c	6	last op was assignment
ASGN	jt.h	2.2	A type
ASSERT	j.h	3.5	argument validation
ASSERTVV	c.c	4	verb-verb case of conj
AS1	a.h	4	adverb-derived monad

CN	jc.h	1.1	char type: N
coerce1	u.c	3.3	coerce monad argument
coerce2	u.c	3.3	coerce dyad arguments
colon	cx.c	8	:
colorq	x.c		PC monad 8!:4
colors	x.c		PC monad 8!:5
compD	vg.c		/: comparator
compI	vg.c		/: comparator
compn	vg.c		/: item size
compUC	vg.c		/: comparator
coninf	w.c		_ __ _. input handler
conj	p.c	1.2	parser action
CONJ	jt.h	2.2	A type
CONJCASE	a.h	4	encode as NN,NV,VN,VV
conjug	ve.c	8	monad +
conname	w.c		convert to name
connect	rt.c	5.4	5!:4 subfunction
connum	w.c		convert to numeric list
consa	pc.c		:12 translator subfn
constr	w.c		convert to string
continue	C		
con1	cg.c		monad m`:0
con2	cg.c		dyad m`:0
copy1	xw.c		monad 2!:4
copy1f	xw.c		copy1 subfunction
copy2	xw.c		dyad 2!:4
copy2f	xw.c		copy2 subfunction
cos	C		
cosh	C		
CPINF	f.c	6	printf _
CPMINUS	f.c	6	printf minus sign
CPNAN	f.c	6	printf _.
CPPLUS	f.c	6	printf plus sign
CQ	jc.h	1.1	char type: quote
CS	jc.h	1.1	char type: space or tab
cstr	u.c	2.1	C string into J string
CS1	a.h	4	conj-derived monad
CS2	a.h	4	conj-derived dyad
ctrans	pc.c	1.2	:12 translator
ctype	t.c	1.1	character type
curry	p.c	1.2	parser action
curtail	vs.c	8	monad }:
cut	cc.c	8	;.
cut01	cc.c	8	monad v;.0
cut02	cc.c	8	dyad v;.0

cut1	cc.c	8	monad u;.n
cut2	cc.c	8	dyad u;.n
CVCASE	k.c		convert: case encoding
cvt	k.c	2.2	convert among B, I, D, Z
CX	jc.h	1.1	char type: other
c2j	f.c	6.3	printf C to J format
C9	jc.h	1.1	char type: digit or sign
D	jt.h	2.2	typedef real
dash	j.c	2.1	'-'
dbin	vm.c		dyad ! D subcase
ddiv	ve.c		dyad % D subcase
deal	v.c	8	dyad ?
DECLF	a.h		declarations for x adv
DECLFG	a.h		declarations for x conj y
decrem	ve.c	8	monad <:
default	C		
define	C		
depth	p.c		depth of function calls
deq	vb.c		dyad = D subcase
det	c.c		monad u/ . v
dex	c.c	8	].
dexp	vm.c		monad ^ D subcase
dfact	vm.c		monad ! D subcase
dfc	vp.c	8	direct from cycle
dfloor	ve.c		monad <. D subcase
dfr	vp.c	8	direct from reduced
DfromZ	k.c	7	convert: D from Z
dfs1	p.c		invoke named monad
dfs2	p.c		invoke named dyad
df1	au.c	4	apply monad
DF1	a.h		derived monad header
df2	au.c	4	apply dyad
DF2	a.h		derived dyad header
dgcd	ve.c		dyad +. D subcase
dgrade1	vg.c	8	monad \:
dgrade2	vg.c	8	dyad \:
divide	ve.c	8	dyad %
dl	x.c		monad 6!:3
dlcm	ve.c		dyad *. D subcase
dle	vb.c		dyad <: D subcase
dlt	vb.c		dyad < D subcase
dmin	ve.c		dyad <. D subcase
dminus	ve.c		dyad - D subcase
dne	vh.c		fne D subcase

do	C		
DO	j.h		do n times under index i
domerr	au.c		verb with empty domain
dot	c.c	8	.
dotprod	c.c	8	dyad u/ . v
double	C		
dplus	ve.c		dyad + D subcase
DR	cr.c		derived rank
dren	ve.c		dyad | D subcase
drep	r.c	5.2	display representation
drop	vs.c	8	dyad }.
drr	r.c	5.2	5!:2 subfunction
drx	x.c	5.2	monad 5!:2
ds	au.c	3.1	define symbol
dsignum	ve.c	7	monad * D subcase
dtymes	ve.c		dyad * D subcase
duble	ve.c	8	monad +:
DxB	k.c		convert: D from B case
DxI	k.c		convert: D from I case
DxZ	k.c		convert: D from Z case
dyad	p.c	1.2	parser action
ebar	vb.c	8	dyad E.
EDGE	p.c	1.2	A composite type
edit	x.c		PC monad 8!:9
efr	cr.c		effective rank
EI	w.c	1.1	word formation fn code
eig1	vm.c	8	monad c.
eig2	vm.c	8	dyad c.
else	C		
EN	w.c	1.1	word formation fn code
encell	f.c	6.2	boxed display subfn
endif	C		
enframe	f.c	6.2	boxed display subfn
ENGAP	f.c		monad ": insert gap
enstack	w.c	1.1	tokens subfunction
ensuite	cx.c		$. handler
eo	c.c		u .. v and u .: v
EPILOG	j.h	2.3	temps clean-up
eps	vb.c	8	dyad e.
eq	vb.c	8	dyad =
errsee	j.c	3.5	1 if display event msgs
EV*	j.h	3.5	event codes
even	c.c	8	..
every	a.c		"each" operator variant

evger	cg.c	8	`:
evmq	x.c	3.5	monad 9!:8
evms	x.c	3.5	monad 9!:9
evoke	u.c		1 if in the form m~
ex	x.c		monad 4!:55
exec1	v.c	8	monad ".
exec2	v.c	8	dyad ".
exit	C		
exp	C		
expn1	vm.c	8	monad ^
expn2	vm.c	8	dyad ^
extern	C		
fa	m.c		free array
fabs	C		
fac	vm.c		monad ! subfunction
fact	vm.c	8	monad !
factp1	cf.c	3.4	monad ^!.n
factp2	cf.c	3.4	dyad ^!.n
FAPPEND	x.h		C file opcode
fclose	C		
fdef	au.c		derive verb/adv/conj
feq	ut.c	7	equal within fuzz
ferror	C		
fgetc	C		
fgets	C		
fh	vh.c	8	dyad i. hasher
fi	u.c		string to integer
fibon	cp.c	8	dyad u^:n
FILL	jt.h		fill value
filler	u.c		fill value
FINDC	vh.c	8	dyad i. find in hash
fit	cf.c	3.4	!.
fitct1	cf.c	3.4	monad u!.n
fitct2	cf.c	3.4	dyad u!.n
fitpp1	cf.c	3.4	monad ":!.n
fix	a.c	8	f.
fixa	a.c		u f. subfunction
fixi	a.c		u f. fn call depth
fixpath	a.c		u f. fn call path
fixpv	a.c		u f. fixpath value
FL	jt.h	2.2	A type
floor	C		
floor1	ve.c	8	monad <.
fmod	C		

fmtB	f.c	6.1	monad `":` B subcase
fmtD	f.c	6.1	monad `":` D subcase
fmtI	f.c	6.1	monad `":` I subcase
FMTX	f.c	6.1	header for formatting fns
fmtZ	f.c	6.1	monad `":` Z subcase
fne	vh.c		1 if items not equal
folk	ct.c	1.3	f g h
fontq	x.c		MAC monad 8!:16
fonts	x.c		MAC monad 8!:17
fopen	C		
for	C		
foreign	x.c	8	!:
forko	ct.c		size 3 a-train or c-train
forkv	p.c	1.2	parser action
fork1	ct.c		monad f g h
fork2	ct.c		dyad f g h
formo	p.c	1.2	parser action
fputc	C		
fputs	C		
fr	m.c		free memory
fram	f.c		boxed display subfn
FREAD	x.h		C file opcode
fread	C		
freal	ut.c	7	within fuzz of real
free	C		
FREE	m.c	2.3	free cover
from	vs.c	8	dyad {
fseek	C		
fsize	C		
ftell	C		
FUNC	jt.h		A composite type
FUPDATE	x.h		C file opcode
FWRITE	x.h		C file opcode
fwrite	C		
fx	r.c		5!:1 inverse
fxr	r.c		5!:1 inverse subfn
fxx	x.c		5!:0
F1	j.h	3.1	primitive monad header
F1RANK	j.h	3.2	monad rank handler
F2	j.h	3.1	primitive dyad header
F2RANK	j.h	3.2	dyad rank handler
ga	m.c		generate array
GA	j.h	2.1	ga cover
gam5243	vm.c		monad ! subfunction

GAPPEND	a.h		`: opcode
gc	m.c		temps: purge; keep arg
gcd	ve.c	8	dyad +.
gc3	m.c		temps: purge; keep args
ge	vb.c	8	dyad >:
GGA	i.c		initialize constant
GG4	i.c		initialize 4-byte constant
GG8	i.c		initialize 8-byte constant
GINSERT	a.h		`: opcode
global	s.c		global symbol table
gnl	xw.c		global names in 4!:1
grade	vg.c		/: subfunction
grade1	vg.c	8	monad /:
grade2	vg.c	8	dyad /:
graft	rt.c	5.4	5!:4 subfunction
gt	vb.c	8	dyad >
gtrain	ct.c	1.3	m\
GTRAIN	a.h	1.3	`: opcode
gt1	pc.c		translator: :12 subfn
gt2	pc.c		translator: :12 subfn
halve	ve.c	8	monad -:
head	vs.c	8	monad {.
HOMO	jt.h	2.1	1 if homogeneous
hook	ct.c	1.3	f g
hooko	ct.c	1.3	size 2 a-train or c-train
hookv	p.c	1.2	parser action
hook1	ct.c		monad f g
hook2	ct.c		dyad f g
host	xf.c		monad 0!:0
hostne	xf.c		monad 0!:1
htab	vh.c		dyad i. hash indices
hypoth	vz.c		complex: monad |
I	jt.h	2.2	typedef integer
IC	jt.h		item count
ID	jt.h	3.1	ID field of verb/adv/conj
iden	ai.c	3.4	identity function adverb
idiv	ve.c		dyad % I subcase
ieq	vb.c		dyad = I subcase
if	C		
ifdef	C		
ifndef	C		
ifrom	vs.c	8	dyad { I subcase
IfromD	k.c	7	convert: I from D

IfromZ	k.c		convert: I from Z
igcd	ve.c		dyad +. I subcase
ii	u.c		i.@#
iind	ap.c	8	dyad u\ subfunction
ilcm	ve.c		dyad *. I subcase
ile	vb.c		dyad <: I subcase
ilt	vb.c		dyad < I subcase
imin	ve.c		dyad <. I subcase
iminus	ve.c		dyad - I subcase
immex	v.c	0	immediate execution
immloop	j.c	0	immex loop
inbuf	io.c		user input buffer
include	C		
increm	ve.c	8	monad >:
indexof	vh.c	8	dyad i.
inf	j.c		_
infile	j.c		input file handle
infix	ap.c	8	dyad u\
inf1	v.c	8	monad _:
inf2	v.c	8	dyad _:
initevm	i.c		initialize event msgs
insert	cg.c		monad m/
INT	jt.h	2.2	A type
int	C		
inv	ai.c	3.4	^:_1
invamp	ai.c	3.4	inv m&v u&n subcase
invdef	ai.c	3.4	inv default
invl	cp.c		monad u^:_1
iota	v.c	8	monad i.
iplus	ve.c		dyad + I subcase
ir	x.c		monad 3!:1
irem	ve.c		dyad | I subcase
is	p.c	1.2	parser action
isatty	C		
isg	p.c		=:
isl	p.c		=.
IS1BYTE	jt.h		A composite type
itake	vs.c	8	dyad {. atomic left arg
itymes	ve.c		dyad * I subcase
IxB	k.c		convert: I from B case
IxD	k.c		convert: I from D case
IxZ	k.c		convert: I from Z case
i0	u.c		value of integer atom

jc	x.c	C	LinkJ interface to C fn
jcir	vm.c		dyad o. Z subcase
jconjug	ve.c		monad + Z subcase
jdiv	ve.c		dyad % Z subcase
jdot1	vm.c	3.1	monad j.
jdot2	vm.c	3.1	dyad j.
jeq	vb.c		dyad = Z subcase
jerr	j.c	3.5	error number
jexp	vm.c		monad ^ Z subcase
JF	io.h		typedef applic. file
jfappend	xf.c		dyad 1!:3
jfdir	xf.c		monad 1!:0
jferase	xf.c		monad 1!:55
jfloor	ve.c		monad <. Z subcase
jfopen	xf.c		open file for processing
JFOPEN	io.h		application file opcode
JFPRINT	io.h		application file opcode
JFPROF	io.h		application file opcode
jfr	lj.c	C	LinkJ cover for fa
jfread	xf.c		monad 1!:1
JFSAVE	io.h		application file opcode
jfsize	xf.c		monad 1!:4
jfwrite	xf.c		dyad 1!:2
jgcd	ve.c		dyad +. Z subcase
jgets	io.c		get a line of user input
jinit	lj.c	C	LinkJ initializations
jinit2	i.c	0	initializations
jiread	xf.c		monad 1!:11
jiwrite	xf.c		dyad 1!:12
jlcm	ve.c		dyad *. Z subcase
jlog	vm.c		dyad ^. Z subcase
jma	lj.c	C	LinkJ cover for ma
jmag	ve.c		monad | Z subcase
jminus	ve.c		monad - Z subcase
jne	vh.c		fne Z subcase
joff	io.c		sign-off
jot	j.c	2.1	<$0
jouts	io.c		display a line
jplus	ve.c		dyad + Z subcase
jpow	vm.c		dyad ^ Z subcase
jpr	f.c	6	display on screen
jputc	io.c		display a character
jputs	io.c		display a string
jrem	ve.c		dyad | Z subcase
jset	lj.c	C	LinkJ copula

jsignal	u.c	3.5	display event msg
jsignum	ve.c	7	monad `*` z subcase
JSPR	j.h		print short string
jsqrt	vm.c		monad `%:` z subcase
jstf	xs.c		pointer to `jstfrec`
jstfrec	xs.c		application file info
jsti	io.h		sesm input
jstinit	io.h		sesm initializations
jstkiav	io.h		sesm key input available
jsto	io.h		sesm output
jstratts	x.c		PC rd display attributes
jstref	x.c		PC refresh display
jstsatts	x.c		PC set display attributes
jstslow	x.c		PC set slow display
jststop	io.h		sesm terminate
jtan2	vz.c		Atari GNU C kludge
jtymes	ve.c		dyad `*` z subcase
jx	lj.c	C	LinkJ execute
key	ap.c	8	dyad `u/.`
KF1	k.c		convert: function header
L_tmpnam	C		
label	cx.c		`m : n` initialize labels
lamin1	vs.c	8	monad `,:`
lamin2	vs.c	8	dyad `,:`
lc	u.c		last character
lcm	ve.c	8	dyad `*.`
le	vb.c	8	dyad `<:`
left1	v.c	8	monad `[`
left2	v.c	8	dyad `[`
less	v.c	8	dyad `-.`
lev	c.c	8	`[.`
link	vs.c	8	dyad `;`
LINKJ	j.h	D	1 if LinkJ
local	s.c		local symbol table
localtime	C		
log	C		
logar1	vm.c	8	monad `^.`
logar2	vm.c	8	dyad `^.`
long	C		
LONG_MAX	j.h		max `I` value
LONG_MIN	j.h		min `I` value
LPAR	jt.h	2.2	`A` type
lr	u.c		left rank

lt	vb.c	8	dyad `<`
ma	m.c		`malloc` cover
mag	ve.c	8	monad `|`
main	j.c	0	`main()`
MALLOC	m.c	2.3	`malloc` cover
malloc	C		
mark	j.c	1.2	marker
MARK	jt.h	2.2	`A` type
mat	f.c	6.2	matrix image
match	vb.c	8	dyad `-:`
math1	vm.c		math monad executor
math1z	vm.c		math monad executor
math2	vm.c		math dyad executor
math2z	vm.c		math dyad executor
matth1	f.c	6.2	`1 1&).@(_1_1&|.)@":@<`
MAX	j.h		maximum
maxbytes	m.c		max bytes used in line
maximum	ve.c	8	dyad `>.`
mdiv	vi.c	8	dyad `%.`
memchr	C		
memcmp	C		
memcpy	C		
memset	C		
merge1	a.c	8	monad `u}`
merge2	a.c	8	dyad `u}`
MIN	j.h		minimum
minimum	ve.c	8	dyad `<.`
minors	c.c	8	monad `u/ . v` subfn
minus	ve.c	8	dyad `-`
minv	vi.c	8	monad `%.`
MMM	vz.c		rotate `z` so that `>:/|+.z`
MOD2	vm.c		`2&|`
monad	p.c	1.2	parser action
move	p.c	1.2	parser action
mr	u.c		monadic rank
mtv	j.c	2.1	`$0`
MTYOIN	io.h		`jsto` opcode
MTYOUT	io.h		`jsto` opcode
mv	u.c		move
mv1	u.c		move one atom
NALP	j.h		size of alphabet
NAME	jt.h	2.2	`A` type
nan	j.c		`_.`

nand	vb.c	8	dyad *:
nc	s.c		4!:0 subfunction
ncases	p.c	1.2	no. of rows in cases
ncx	x.c		monad 4!:0
NDEPTH	p.c		max depth of fn calls
ndig	w.c		AMIGA LatticeC kludge
ne	vb.c	8	dyad ~:
negate	ve.c	8	monad -
negl	j.c	2.1	_1
neq	vb.c		dyad = subfunction
NEVM	j.h	3.5	no. of event messages
NFKD	io.h		size of fn key defn buffer
NINB	io.h		size of typeahead buffer
NINPUT	j.h		max length of input line
nla	s.c		4!:1 initials interest
nline	cx.c		m : n number of lines
nlmask	s.c		4!:1 numbers to type
NLOG	io.h		size of session log
nls	s.c		4!:1 subfunction
nlx	s.c		4!:1 subfunction
nl1	x.c		monad 4!:1
nl2	x.c		dyad 4!:1
NMEM	m.c		max size for malloc
NN	a.h	4	noun-noun case of conj
NOBUF	j.h		length of obuf
nor	vb.c	8	dyad +:
norm	vi.c		%:@(+/ . * +)
not	ve.c	8	monad -.
NOTCONJ	jt.h	1.2	A composite type
NOUN	jt.h	2.2	A composite type
NPP	j.h		max value for qpp
NPROMPT	j.h		max length of prompt
NTH2	f.c	6.3	dyad ": max width
NTSTACK	j.h		temps: stack frame size
nu	w.c	1.1	national use alternatives
nub	v.c	8	monad ~.
nubsieve	vb.c	8	monad ~:
NUMERIC	jt.h	2.2	A composite type
NV	a.h	4	noun-verb case of conj
NW	xw.c		WS length of header
NWPFX	xw.c		WS length of prefix
NWPTR	xw.c		WS length of pointer
NXIL	xw.c		WS block size in wcp
oblique	ap.c	8	monad u/.

obuf	j.c		buffer for short output
obverse	c.c	3.3	:.
obv1	c.c		monad u :. v
obv2	c.c		dyad u :. v
odd	c.c	8	.:
oind	ap.c	8	monad u/. subfunction
oldout	xs.c		old outfile value
omask	ap.c	8	dyad u\ subfunction
one	j.c	2.1	1
one1	v.c	8	monad 1:
one2	v.c	8	dyad 1:
onm	s.c		opened name
on1	c.c	4	monad u&v and u@v
on2	c.c	4	dyad u&v
ope	vs.c	8	monad >
oprod	a.c		dyad u/
ord	vp.c	8	order of a permutation
osub	ap.c	8	monad u/. subfunction
outfile	j.c		output file handle
outfix	ap.c	8	dyad u\.
outof	vm.c	8	dyad !
over	vs.c	8	dyad ,
OVERFLOW	j.h		large D value
overr	vs.c	8	dyad ,.
P	jt.h		typedef primitives
pad	rt.c	5.4	5!:4 subfunction
parse	p.c	1.2	interpret tokenized line
pcopy1	xw.c		monad 2!:5
pcopy1f	xw.c		pcopy1 subfunction
pcopy2	xw.c		dyad 2!:5
pcopy2f	xw.c		pcopy2 subfunction
pcvt	k.c	2.2	convert if possible
pdt	vi.c		dyad +/ . *
pfill	vp.c	8	permutation fill
PI	j.h		3.14159265358 ...
pie	j.c	2.1	3.14159265358 ...
pind	vp.c	8	positive indices
pinv	vp.c		permutation inverse
pix	vm.c	8	monad o.
plus	ve.c	8	dyad +
ply	cp.c		monad u^:n
polar	vm.c	8	monad *.
poly1	vm.c	8	monad p.
poly2	vm.c	8	dyad p.

`povtake`	v.c		monad `>` subfunction
`powop`	cp.c	8	`^:`
`prefix`	ap.c	8	monad `u/`
`PREF1`	a.h	3.2	monad rank handler
`PREF2`	a.h	3.2	dyad rank handler
`preparse`	cx.c		tokenize lines in `m : n`
`probe`	s.c		symbol table: ref or set
`prod`	u.c		`*/` on integer list
`PROLOG`	j.h	2.3	temps: checkpoint
`prompt`	io.c	0	display user input prompt
`promptq`	x.c		monad `9!:4`
`prompts`	x.c		monad `9!:5`
`prtscr`	x.c		MAC monad `8!:19`
`ps`	t.c	3.1	table of primitives
`psave1`	xw.c		monad `2!:3`
`psave1f`	xw.c		`psave1` subfunction
`psave2`	xw.c		dyad `2!:3`
`psave2f`	xw.c		`psave2` subfunction
`psptr`	t.c	3.1	index in `ps` for each ID
`pstr`	f.c	6.3	dyad `":` `printf` string
`PT`	p.h	1.2	`typedef` parse table
`Ptr`	jt.h		`typedef` pointer
`punc`	p.c	1.2	parser action
`qbx`	j.c	6.2	box drawing characters
`qct`	cf.c	7	comparison tolerance
`qevm`	j.c	3.5	event messages
`qfuzz`	j.c	7	fuzz
`qpp`	cf.c		print precision
`qprompt`	j.c		input prompt
`qq`	c.c	8	"
`qr`	vi.c	8	QR decomposition
`qrl`	j.c	8	random link
`qsort`	C		
`R`	j.h		`return`
`ra`	m.c		reference array
`rank`	vs.c	8	monad `#@$`
`rankle`	u.c		``,`]@.(*@#@$)``
`rank1`	cr.c		monad `u"n`
`rank1ex`	cr.c		monad rank executor
`rank2`	cr.c		dyad `u"n`
`rank2ex`	cr.c		dyad rank executor
`ravel`	vs.c	8	monad `,`
`raze`	v.c	8	monad `;`

`razein`	vb.c	8	monad `e.`
`rbrace`	a.c	8	`}`
`rc`	f.c	6.2	boxed display subfn
`rd`	xf.c		file read
`rdot1`	vm.c	8	monad `r.`
`rdot2`	vm.c	8	dyad `r.`
`RE`	j.h	3.5	return if error
`recip`	ve.c	8	monad `%`
`rect`	vm.c	8	monad `+.`
`reduce`	a.c		monad `u/`
`refresh`	x.c		PC monad `8!:7`
`reitem`	vs.c	8	dyad `$`
`repeat`	vs.c	8	dyad `#`
`reshape`	vs.c		dyad `$,`
`residue`	ve.c	8	dyad `|`
`return`	C		
`reverse`	vs.c	8	monad `|.`
`rewind`	C		
`rfd`	vp.c	8	reduced from direct
`RHS`	jt.h	1.2	`A` composite type
`ri`	x.c		monad `3!:2`
`right1`	v.c	8	monad `]`
`right2`	v.c	8	dyad `]`
`rinv`	vi.c	8	monad `128!:0`
`rlq`	x.c		monad `9!:0`
`rls`	x.c		monad `9!:1`
`RMAX`	j.h		max rank
`RMAXL`	j.h		max rank as `long`
`roll`	v.c	8	monad `?`
`root`	vm.c	8	dyad `%:`
`ropen`	xs.c		open script
`rotate`	vs.c	8	dyad `|.`
`round`	vi.c		`<.@(0.5&+)`
`RPAR`	jt.h	2.2	`A` type
`rr`	u.c		right rank
`RZ`	j.h	3.5	`return` if 0
`S`	jt.h		`typedef` short integer
`SA`	w.c	1.1	word formation state
`SAPPEND`	x.h		script opcode
`save1`	xw.c		monad `2!:2`
`save1f`	xw.c		`save1` subfunction
`save2`	xw.c		dyad `2!:2`
`save2f`	xw.c		`save2` subfunction
`sc`	u.c	2.1	scalar integer

SCALARFN	a.h		1 if scalar function
scalar4	u.c	2.1	scalar 4-byte object
scc	u.c	2.1	scalar character
scf	u.c	2.1	scalar floating point
sclass	vb.c	8	monad =
scnm	u.c		scalar name
scpt	xs.c		scpt1/scpt2 subfn
scpt1	xs.c		monad 0!:2 and 0!:3
scpt2	xs.c		dyad 0!:2 and 0!:3
script1	xs.c		monad 1!:2
script2	xs.c		dyad 1!:2
SEEK_CUR	x.h		fseek opcode
SEEK_END	x.h		fseek opcode
SEEK_SET	x.h		fseek opcode
seg	ap.c	8	monad u\ subfunction
self	*	4	A array for current verb
selfv	p.c		old $: value
self1	p.c		monad $:
self2	p.c		dyad $:
sesm	j.c		1 if using session mgr
sesmexit	io.c		session manager: epilog
sesminit	io.c		session manager: prolog
sex	s.c		symbol table expunge
sex1	cr.c	3.3	monad scalar executor
sex2	cr.c	3.3	dyad scalar executor
SF	jt.h	3.3	typedef scalar function
SF1	v.h	3.3	scalar monad header
SF2	v.h	3.3	scalar dyad header
SGN	j.h		signum
shape	vs.c	8	monad $
shift1	cf.c	3.4	monad |.!.n
shift2	cf.c	3.4	dyad |.!.n
shl	ai.c		1&(|.!.w)
short	C		
shr	ai.c		|.!.w
sigflpe	i.c		floating point exception
sigint	i.c		user interrupt
signal	C		
signum	ve.c	8	monad *
sin	C		
sinh	C		
size_t	C		
sizeof	C		
slash	a.c	8	/
sldot	ap.c	8	/.

sleep	C		
sll_	ai.c		u/"1"_
SN	w.c	1.1	word formation state
SNB	w.c	1.1	word formation state
SNZ	w.c	1.1	word formation state
sp	x.c		monad 7!:0
spell	w.c	1.1	spelling table
spellin	w.c	1.1	ASCII string to ID
spellout	w.c	1.1	ID to string
spit	x.c		monad 7!:2
sprintf	C		
sps	x.c		monad 7!:1
SQ	w.c	1.1	word formation state
SQQ	w.c	1.1	word formation state
sqroot	vm.c	8	monad %:
sqrt	C		
square	ve.c	8	monad *:
SREAD	x.h		script opcode
srd	s.c		symbol table read
srdlg	s.c		srd local or global
sreduce	x.c		monad f/ for scalar f
srep	r.c	5.3	string representation
sr1	r.c	5.3	srep subfunction
srx	x.c	5.3	monad 5!:3
SS	w.c	1.1	word formation state
sscript1	xs.c		monad 1!:3
sscript2	xs.c		dyad 1!:3
ST	w.c	1.1	typedef rhematic states
state	w.c	1.1	rhematic state table
static	C		
stdin	C		
stdnm	s.c		standardize name
stdout	C		
str	u.c		string of length n
strcat	C		
strchr	C		
strcmp	C		
strcpy	C		
strlen	C		
strspn	C		
strtod	C		
struct	C		
stype	x.c	2.2	monad 3!:0
sum	ve.c		monad +/
suffix	ap.c	8	monad u\

swap	a.c	8	~
swap1	a.c		monad u~
swap2	a.c		dyad u~
switch	C		
SX	w.c	1.1	word formation state
SY	jt.h	2.2	typedef symb tab entry
SYMB	jt.h	2.2	A type
symbis	s.c		symbol table set
symbrd	s.c		symbol table read
SYS	j.h	D	system ID
SYS_*	j.h	D	system IDs and masks
system	C		
SZ	w.c	1.1	word formation state
S9	w.c	1.1	word formation state
taa	ct.c	1.3	trains: adv adv
TAA	ct.c	1.3	trains: adv adv case
taaa	ct.c	1.3	trains: adv adv adv
TAAA	ct.c	1.3	trains: adv adv adv case
TAAC	ct.c	1.3	trains: adv adv conj case
table	vs.c	8	monad ,.
tac	ct.c	1.3	trains: adv conj
TAC	ct.c	1.3	trains: adv conj case
taca	ct.c	1.3	trains: adv conj adv
TACA	ct.c	1.3	trains: adv conj adv case
tacc	ct.c	1.3	trains: adv conj conj
TACC	ct.c	1.3	trains: adv conj conj case
TACT	p.h	1.2	translator action header
tail	vs.c	8	monad {:
take	vs.c	8	dyad {.
tally	vs.c	8	monad #
tbase	m.c		temps: base
tca	ct.c	1.3	trains: conj adv
TCA	ct.c	1.3	trains: conj adv
tcaa	ct.c	1.3	trains: conj adv adv
TCAA	ct.c	1.3	trains: conj adv adv
TCAC	ct.c	1.3	trains: conj adv conj
TCC	ct.c	1.3	trains: conj conj
tcca	ct.c	1.3	trains: conj conj adv
TCCA	ct.c	1.3	trains: conj conj adv
tccc	ct.c	1.3	trains: conj conj conj
TCCC	ct.c	1.3	trains: conj conj conj
tceil	ut.c	7	tolerant >.
TDECL	ct.c		trains: declarations
teq	ut.c	7	tolerant =

tf	m.c		temps: free old frame
tfail	pv.c		translator: 1 if failed
tfloor	ut.c	7	tolerant <.
tg	m.c		temps: get new frame
th	f.c	6.1	monad ": num subcase
thbox	f.c	6.2	monad ": A subcase
thn	vh.c	8	dyad i. hash bytes
thorn1	f.c	6	monad ":
thorn2	f.c	6.3	dyad ":
ths	vh.c	8	dyad i. start of hash
tie	cg.c	8	`
time	C		
time_t	C		
tle	ut.c	7	tolerant <:
tleaf	rt.c	5.4	5!:4 subfunction
tlt	ut.c	7	tolerant <
tmpnam	C		
tname	pv.c		translator: names
tokens	w.c	1.1	build parse stack
tostdout	j.c		1 if output to stdout
totbytes	m.c		bytes used in session
tparse	pv.c		translator: :11 and :12
tpop	m.c	0	temps: purge
tpush	m.c		temps: insert new entry
traverse	m.c		apply f to each leaf
treach	rt.c	5.4	apply trr to each atom
trep	rt.c	5.4	tree representation
troot	rt.c	5.4	5!:4 subfunction
trr	rt.c	5.4	5!:4 subfunction
trx	x.c		monad 5!:4
ts	x.c		monad 6!:0
tsit	x.c		monad 6!:2
tss	x.c		monad 6!:1
tssbase	j.c		time base
tstack	m.c		temps (A*)AV(tstacka)
tstacka	m.c		temps current stackframe
ttokens	pv.c		translator: tokenize
ttop	m.c	2.3	temps top
tval	pv.c		translator: values
twprimes	s.c		sizes for symbol tables
two	j.c	2.1	2
tymes	ve.c	8	dyad *
typedef	C		
U	jt.h		unsigned

UC	jt.h		typedef unsigned byte
under	c.c	8	&.
UNDERFLOW	j.h		small D value
under1	c.c		monad u&.v
under2	c.c		dyad u&.v
unlink	C		
ung	xw.c		WS remove given names
unquote	p.c		monad or dyad m~
unquo1	a.c		monad m~
unquo2	a.c		dyad m~
unsigned	C		
unsr	r.c		5!:3 inverse
until	C		
unw	r.c		unsr subfunction
upon2	c.c		dyad u@v
V	jt.h	4	typedef verb
vadv	pv.c	1.2	:11 translator action
VAV	jt.h	2.1	AV for verb/adverb/conj
vconj	pv.c	1.2	:11 translator action
vcurry	pv.c	1.2	:11 translator action
vdyad	pv.c	1.2	:11 translator action
VERB	jt.h	2.2	A type
vfin	xf.c		validate file name
vforkv	pv.c	1.2	:11 translator action
vformo	pv.c	1.2	:11 translator action
vhookv	pv.c	1.2	:11 translator action
vi	u.c		validate integer
vib	u.c		validate integer, bounded
vis	pv.c	1.2	:11 translator action
vmonad	pv.c	1.2	:11 translator action
vmove	pv.c	1.2	:11 translator action
vn	u.c		validate noun
VN	a.h	4	verb-noun case of conj
vnm	s.c		validate name
void	C		
vpunc	pv.c	1.2	:11 translator action
vs	u.c		validate string
vtrans	pv.c	1.2	:11 translator
VV	a.h	4	verb-verb case of conj
v2	u.c		integer pair
w	*		right argument
wa	xf.c		file write or append
WATERLOO	j.h	D	1 if Waterloo libraries

WB	f.c	6.1	monad ": B max width
wcp	xw.c		WS copy
WD	f.c	6.1	monad ": D max width
wend	xw.c		WS offset to directory
wex	xw.c		WS expunge
wexf	xw.c		WS wex subfunction
WF1	xw.c		WS monad header
WF2	xw.c		WS dyad header
while	C		
while1	cp.c		monad u^:v
while2	cp.c		dyad u^:v
WI	f.c	6.1	monad ": I max width
withl	c.c	4	monad m&v
withr	c.c	4	monad u&n
wnc	xw.c		WS name class
wncf	xw.c		WS wnc subfunction
wnl	xw.c		WS name list
wnlf	xw.c		WS wnl subfunction
wopen	xw.c		WS open
wopr1	xw.c		WS monad executor
wopr2	xw.c		WS dyad executor
wordil	w.c	1.1	word index and length
words	w.c	1.1	monad ;:
wp	xw.c		WS dir names
WP	jt.h		words per array
wpfx	xw.c		WS prefix
wptr	xw.c		WS wp and wq location
wq	xw.c		WS dir index/length/type
wr	io.c		write to screen
wrdir	xw.c		WS write directory
WREAD	x.h		WS opcode
wtype	t.c		ctype clone for wordil
WUPDATE	x.h		WS opcode
WWRITE	x.h		WS opcode
WZ	f.c	6.1	monad ": Z max width
xadv	cx.c		x m : 1
XC	x.h		!: argument encoding
xconj	cx.c		x m : 2 y
xcvt	k.c	2.2	convert to "lowest" type
xcvta	k.c		xcvt subfunction
xd	cx.c		monad or dyad m : n
xdash	rt.c	5.4	"dash" box drawing char
xdgcd	ve.c		dyad +. D subcase subfn
xdrem	ve.c		dyad | D subcase subfn

xigcd	ve.c		dyad +. I subcase subfn
xil	xw.c		WS dir, index/length
XINF	j.h		_ internal representation
xirem	ve.c		dyad | I subcase subfn
XNAN	j.h		_. internal representation
xn1	cx.c		monad m : y
xn2	cx.c		dyad x : n
xv1	cx.c		monad u : y
xv2	cx.c		dyad x : v
z	*		result
Z	jt.h	2.2	typedef complex
zacos	vz.c	8	complex: _2&o.
zacosh	vz.c	8	complex: _6&o.
zarc	vz.c	8	complex: 12&o.
zasin	vz.c	8	complex: _1&o.
zasinh	vz.c	8	complex: _5&o.
ZASSERT	vz.c		complex: arg validation
zatan	vz.c	8	complex: _3&o.
zatanh	vz.c	8	complex: _7&o.
zceiling	vz.c		complex: monad >.
zcir	vz.c	8	complex: dyad o.
zconjug	vz.c		complex: monad +
zcos	vz.c	8	complex: 2&o.
zcosh	vz.c	8	complex: 6&o.
zdiv	vz.c		complex: dyad %
ZEPILOG	vz.c		complex: standard exit
zeq	vz.c		complex: dyad =
ZEQ	vz.c		complex: 1 if equal
zero	j.c	2.1	0
zeroZ	j.c		8-byte zero
zero1	v.c	8	monad 0:
zero2	v.c	8	dyad 0:
zexp	vz.c	8	complex: monad ^
ZEZ	vz.c		complex: 1 if zero
zfloor	vz.c	8	complex: monad <.
ZF1	vz.c		complex: monad header
ZF1DECL	vz.c		complex: declarations
ZF2	vz.c		complex: dyad header
ZF2DECL	vz.c		complex: declarations
zgcd	vz.c		complex: dyad +.
zgcd1	vz.c		complex: zgcd subfn
zj	vz.c		complex: 0j1
zlcm	vz.c		complex: dyad *.
zlog	vz.c	8	complex: monad ^.
zmag	vz.c	8	complex: monad |
zminus	vz.c		complex: dyad -
zmj	vz.c		complex: 0j_1
zm4	vz.c	8	complex: _4&o.
znegate	vz.c		complex: monad -
znonce1	vz.c		complex: nonce error
znonce2	vz.c		complex: nonce error
ZNZ	vz.c		complex: 1 if nonzero
ZOV	vz.c		complex: 1 if overflow
zplus	vz.c		complex: dyad +
zpow	vz.c		complex: dyad ^
zp4	vz.c	8	complex: 4&o.
zp8	vz.c	8	complex: 8&o.
zrem	vz.c		complex: dyad |
zsin	vz.c	8	complex: 1&o.
zsinh	vz.c	8	complex: 5&o.
zsqrt	vz.c		complex: monad %:
ZS1	vz.c		complex: monad shell
ZS2	vz.c		complex: dyad shell
ztan	vz.c	8	complex: 3&o.
ztanh	vz.c	8	complex: 7&o.
ztrend	vz.c	8	complex: monad *
ztymes	vz.c		complex: dyad *
ZUN	vz.c		complex: 1 if underflow
ZxB	k.c		convert: Z from B case
ZxD	k.c		convert: Z from D case
ZxI	k.c		convert: Z from I case
z1	vz.c		complex: 1j0

Appendix E. Foreign Conjunction

x, xf, xs, etc. are names of C program files

0!:0	host xf •
0!:1	hostne xf •
0!:2	script1 xs • script2 xs
0!:3	sscript1 xs • sscript2 xs
0!:55	joff u •
1!:0	jfdir xf •
1!:1	jfread xf •
1!:2	• jfwrite xf
1!:3	• jfappend xf
1!:4	jfsize xf •
1!:11	jiread xf •
1!:12	• jiwrite xf
1!:55	jferase xf •
2!:0	• wnc xw
2!:1	wnl xw •
2!:2	save1 xw • save2 xw
2!:3	psave1 xw • psave2 xw
2!:4	copy1 xw • copy2 xw
2!:5	pcopy1 xw • pcopy2 xw
2!:55	• wex xw
3!:0	stype x •
3!:1	ir x •
3!:2	ri x •
4!:0	ncx s •
4!:1	nl1 s • nl2 s
4!:55	ex s •
5!:0	fxx x •
5!:1	arx x •
5!:2	drx x •
5!:3	srx x •
5!:4	trx x •

6!:0	ts x •
6!:1	tss x •
6!:2	tsit x •
6!:3	dl x •
7!:0	sp x •
7!:1	sps x •
7!:2	spit x •
8!:0	cgaq x •
8!:1	cgas x •
8!:4	colorq x •
8!:5	colors x •
8!:7	refresh x •
8!:9	edit x •
8!:16	fontq x •
8!:17	fonts x •
8!:19	prtscr x •
9!:0	rlq x •
9!:1	rls x •
9!:4	promptq x •
9!:5	prompts x •
9!:6	boxq x •
9!:7	boxs x •
9!:8	evmq x •
9!:9	evms x •
10!:	jc x •
128!:0	qr vi •
128!:1	rinv vi •

Appendix E. Foreign Conjunction

x, xf, xs, etc. are names of C program files

0!:0	host xf •
0!:1	hostne xf •
0!:2	script1 xs • script2 xs
0!:3	sscript1 xs • sscript2 xs
0!:55	joff u •
1!:0	jfdir xf •
1!:1	jfread xf •
1!:2	• jfwrite xf
1!:3	• jfappend xf
1!:4	jfsize xf •
1!:11	jiread xf •
1!:12	• jiwrite xf
1!:55	jferase xf •
2!:0	• wnc xw
2!:1	wnl xw •
2!:2	save1 xw • save2 xw
2!:3	psave1 xw • psave2 xw
2!:4	copy1 xw • copy2 xw
2!:5	pcopy1 xw • pcopy2 xw
2!:55	• wex xw
3!:0	stype x •
3!:1	ir x •
3!:2	ri x •
4!:0	ncx s •
4!:1	nl1 s • nl2 s
4!:55	ex s •
5!:0	fxx x •
5!:1	arx x •
5!:2	drx x •
5!:3	srx x •
5!:4	trx x •
6!:0	ts x •
6!:1	tss x •
6!:2	tsit x •
6!:3	dl x •
7!:0	sp x •
7!:1	sps x •
7!:2	spit x •
8!:0	cgaq x •
8!:1	cgas x •
8!:4	colorq x •
8!:5	colors x •
8!:7	refresh x •
8!:9	edit x •
8!:16	fontq x •
8!:17	fonts x •
8!:19	prtscr x •
9!:0	rlq x •
9!:1	rls x •
9!:4	promptq x •
9!:5	prompts x •
9!:6	boxq x •
9!:7	boxs x •
9!:8	evmq x •
9!:9	evms x •
10!:	jc x •
128!:0	qr vi •
128!:1	rinv vi •

Appendix F. System Summary

vb, p, v, etc. are names of C program files

=	sclass vb • eq vb	isl p	isg p
<	box v • lt vb	floor1 ve • minimum ve	decrem ve • le vb
>	ope v • gt vb	ceil1 ve • maximum ve	increm ve • ge vb
_	coninf w	coninf w	inf1 v • inf2 v
+	conjug ve • plus ve	rect vm • gcd ve	duble ve • nor vb
*	signum ve • tymes ve	polar vm • lcm ve	square ve • nand vb
-	negate ve • minus ve	not ve • less v	halve ve • match vb
%	recip ve • divide ve	minv vi • mdiv vi	sqroot vm • root vm
^	expn1 vm • expn2 vm	logar1 vm • logar2 vm	• powop cp
$	shape vs • reitem vs	ensuite cx	self1 p • self2 p
~	swap a •	nub v •	nubsieve vb • ne vb
|	mag ve • residue ve	reverse vs • rotate vs	cant1 vs • cant2 vs
.	• dot c	• even c	• odd c
:	• colon cx	• obverse c	
,	ravel vs • over vs	table vs • overr vs	lamin1 vs • lamin2 vs
;	raze v • link v	• cut cc	words w •
#	tally vs • repeat vs	base1 ve • base2 ve	abase1 ve • abase2 ve
!	fact vm • outof vm	• fit cf	• foreign x
/	slash a •	sldot ap •	grade1 vg • grade2 vg
\	bslash ap •	bsdot ap •	dgrade1 vg • dgrade2 vg
[	left1 v • left2 v	• lev c	
]	right1 v • right2 v	• dex c	
{	catalog vs • from vs	head vs • take vs	tail vs •
}	rbrace a •	behead vs • drop vs	curtail vs •
"	• qq cr	exec1 v • exec2 v	thorn1 f • thorn2 f
`	• tie cg		• evger cg
@	• atop c	• agenda cg	• atco c
&	• amp c	• under c	• ampco c
?	roll v • deal v		

)	label cx	a.	alp j	A.	adot1 vp • adot2 vp
b.	bool a •	c.	eig1 vm • eig2 vm	C.	cdot1 vp • cdot2 vp
e.	razein vb • eps vb	E.	• ebar vb	f.	fix a •
i.	iota v • indexof vh	j.	jdot1 vm • jdot2 vm	NB.	wordil w
o.	pix vm • circle vm	p.	poly1 vm • poly2 vm	r.	rdot1 vm • rdot2 vm
x.	xd cx	y.	xd cx	0:	zero1 v • zero2 v
				1:	one1 v • one2 v