An Implementation of J

Roger K.W. Hui
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

1. Interpreting a Sentence
   1.1 Word Formation
   1.2 Parsing
   1.3 Trains
   1.4 Name Resolution

2. Nouns
   2.1 Arrays
   2.2 Types
   2.3 Memory Management

3. Verbs
   3.1 Anatomy of a Verb
   3.2 Rank
   3.3 Atomic (Scalar) Verbs
   3.4 Obverses, Identities, and Variants
   3.5 Error Handling

4. Adverbs and Conjunctions

5. Representation
   5.1 Atomic Representation
   5.2 Display Representation
   5.3 String Representation
   5.4 Tree Representation

6. Display
   6.1 Numeric Display
   6.2 Boxed Display
   6.3 Formatted Display

7. Comparatives

8. Primitives

Appendices
   A. Incunabulum
   B. Program Files
   C. The LinkJ Interface
   D. Compiling
   E. Foreign Conjunction
   F. System Summary

Bibliography

Glossary and Index
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 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:

\[ \text{sum} = + \_6.95 \_i.3 \_4 \]

The source code for word formation is in file w.c. The process is controlled by the function \texttt{wordil} (word index and length) and the table \texttt{state}. Rows of \texttt{state} correspond to 10 states; columns to 9 character classes. Each table entry is a (new state, function) pair. Starting at state \texttt{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.

<table>
<thead>
<tr>
<th>NEW STATE/FUNCTION</th>
<th>STATES</th>
<th>CLASSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>b ? = a = N = a = 9 = ? = ? = ' =</td>
<td>b Blank</td>
<td>b Blank</td>
</tr>
<tr>
<td>b&gt; ? &gt; a &gt; N &gt; a &gt; 9 &gt; ?</td>
<td>?</td>
<td>? Other</td>
</tr>
<tr>
<td>b&gt; ? &gt; a a a a ?</td>
<td>?</td>
<td>a Name</td>
</tr>
<tr>
<td>b&gt; ? &gt; a a B a ?</td>
<td>?</td>
<td>N N</td>
</tr>
<tr>
<td>b&gt; ? &gt; a a a a C</td>
<td>?</td>
<td>B NB</td>
</tr>
<tr>
<td>z z z z z z</td>
<td>?</td>
<td>C NB,</td>
</tr>
<tr>
<td>b&gt; ? &gt; 9 9 9 9 9</td>
<td>9</td>
<td>Numbers</td>
</tr>
<tr>
<td>' ' ' ' ' ' '</td>
<td>'</td>
<td>Quote</td>
</tr>
<tr>
<td>b&gt; ? &gt; a &gt; N &gt; a &gt; 9 &gt; ?</td>
<td>?</td>
<td>' Adjacent Qtes</td>
</tr>
<tr>
<td>z z z z z z z</td>
<td>z</td>
<td>Comment</td>
</tr>
<tr>
<td>b ? a N B 9 .</td>
<td></td>
<td>FUNCTION</td>
</tr>
</tbody>
</table>

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

Chapter 1 Interpreting
\( \text{Emit}(j, i-1) \) 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), \text{Emit} combines their indices to form a single word. At the end of the string, \( \text{Emit}(j, i-1) \) is executed.

This process is applied to \( \text{sum} = +/_{-6.95} \times i.3 \ 4 \), 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 \( j \).

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

Chapter 1 3 Interpreting
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:

```c
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 (cASN); 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:

```c
#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, wpunc, 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 \(((i \# y) = i \sim y) \# y\) where \(y = 'abc'\). (\$ denotes the marker.)

<table>
<thead>
<tr>
<th>QUEUE</th>
<th>STACK</th>
<th>RULE/ ACTION</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$((i # y) = i \sim y) # y$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$((i # y) = i \sim y) # y$</td>
<td>'aba'</td>
<td>13 move</td>
<td></td>
</tr>
<tr>
<td>$((i # y) = i \sim y)$</td>
<td>#'aba'</td>
<td>13 move</td>
<td></td>
</tr>
<tr>
<td>$((i # y) = i \sim y)$</td>
<td>)#'aba'</td>
<td>13 move</td>
<td></td>
</tr>
<tr>
<td>$((i # y) = i \sim)$</td>
<td>'aba')#'aba'</td>
<td>13 move</td>
<td></td>
</tr>
<tr>
<td>$((i # y) = i \sim)$</td>
<td></td>
<td>'aba')#'aba'</td>
<td>13 move</td>
</tr>
</tbody>
</table>

---

$((i \# y) = i \sim 'aba')#'aba' | 13 move |

$((i \# y) = i \sim 'aba')#'aba' | 13 move |

$((i \# y) = i \sim 'aba')#'aba' | 13 move |

$((i \# y) = v0 'aba')#'aba' | 3 adv |

| $((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' |

---

$(110) = v0 'aba')#'aba' | 2 dyad | 110 -: 012=010 |

---

$110#'aba' | 12 punc |

---

Chapter 1  7  Interpreting
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:

<table>
<thead>
<tr>
<th>FORK</th>
<th>HOOK</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g$ \ / \ $g$</td>
<td>$g$ \ / \ $g$</td>
</tr>
<tr>
<td>$f$ \ // \ $h$</td>
<td>$y$ \ // \ $h$</td>
</tr>
<tr>
<td></td>
<td>$y$ \ // \</td>
</tr>
<tr>
<td>$y$ \ // \ $y$</td>
<td>$y$ \ // \ $y$</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>TRAIN</th>
<th>RESULT</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0 a1 a2</td>
<td>adverb</td>
<td>$x ; a0 ; a1 ; a2$</td>
</tr>
<tr>
<td>a0 a1 c2</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>a0 c1 a2</td>
<td>conjunction</td>
<td>$(x ; a0) ; c1 ; (y ; a2)$</td>
</tr>
<tr>
<td>a0 c1 c2</td>
<td>conjunction</td>
<td>$(x ; a0) ; c1 ; (x ; c2 ; y)$</td>
</tr>
<tr>
<td>c0 a1 a2</td>
<td>conjunction</td>
<td>$x ; c0 ; y ; a1 ; a2$</td>
</tr>
<tr>
<td>c0 a1 c2</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>c0 c1 a2</td>
<td>conjunction</td>
<td>$(x ; c0 ; y) ; c1 ; (x ; a2)$</td>
</tr>
<tr>
<td>c0 c1 c2</td>
<td>conjunction</td>
<td>$(x ; c0 ; y) ; c1 ; (x ; c2 ; y)$</td>
</tr>
<tr>
<td>a0 a1</td>
<td>adverb</td>
<td>$x ; a0 ; a1$</td>
</tr>
<tr>
<td>a0 c1</td>
<td>adverb</td>
<td>$x ; a0 ; c1 ; x$</td>
</tr>
<tr>
<td>c0 a1</td>
<td>conjunction</td>
<td>$x ; c0 ; y ; a1$</td>
</tr>
<tr>
<td>c0 c1</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

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`:

```c
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 1.3 4 are represented thus:

```
<table>
<thead>
<tr>
<th>t</th>
<th>c</th>
<th>n</th>
<th>r</th>
<th>s[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>17 Cogito, ergo sum</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>t</th>
<th>c</th>
<th>n</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>t</th>
<th>c</th>
<th>n</th>
<th>r</th>
<th>s[0]</th>
<th>s[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>1</td>
<td>12</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

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

<table>
<thead>
<tr>
<th>PART</th>
<th>MACRO</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>AT</td>
<td>type</td>
</tr>
<tr>
<td>c</td>
<td>AC</td>
<td>reference count</td>
</tr>
<tr>
<td>n</td>
<td>AN</td>
<td>number of atoms</td>
</tr>
<tr>
<td>r</td>
<td>AR</td>
<td>rank</td>
</tr>
<tr>
<td>s</td>
<td>AS</td>
<td>shape</td>
</tr>
<tr>
<td></td>
<td>AV</td>
<td>&quot;value&quot;, atoms in ravelled order</td>
</tr>
</tbody>
</table>

Chapter 2 12  Nouns
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

\[
\text{GA}(xyz,t,n,r,s);
\]

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:

```c
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;
```

Chapter 2 13 Nouns
The following utilities in file u.c are convenient for creating simple arrays:

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

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

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

\[
\begin{align*}
\text{zero} & \quad 0 \\
\text{one} & \quad 1 \\
\text{two} & \quad 2 \\
\text{neg1} & \quad -1 \\
\text{pie} & \quad \pi \text{ ("pi" conflicts with C usage)} \\
\text{a0j1} & \quad 0j1 \\
\text{mtv} & \quad \$0 \\
\text{jot} & \quad <$0 \\
\text{dash} & \quad ' - ' \\
\end{align*}
\]
2.2 Types

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

<table>
<thead>
<tr>
<th>( \texttt{AT} (\mathbf{x}) )</th>
<th>( \texttt{C DATA TYPE} )</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{BOOL}</td>
<td>\texttt{B}</td>
<td>Boolean</td>
</tr>
<tr>
<td>\texttt{CHAR}</td>
<td>\texttt{C}</td>
<td>literal</td>
</tr>
<tr>
<td>\texttt{INT}</td>
<td>\texttt{I}</td>
<td>integer</td>
</tr>
<tr>
<td>\texttt{FL}</td>
<td>\texttt{D}</td>
<td>floating point</td>
</tr>
<tr>
<td>\texttt{CMPX}</td>
<td>\texttt{Z}</td>
<td>complex</td>
</tr>
<tr>
<td>\texttt{BOX}</td>
<td>\texttt{A}</td>
<td>boxed</td>
</tr>
<tr>
<td>\texttt{VERB}</td>
<td>\texttt{V}</td>
<td>verb</td>
</tr>
<tr>
<td>\texttt{ADV}</td>
<td>\texttt{V}</td>
<td>adverb</td>
</tr>
<tr>
<td>\texttt{CONJ}</td>
<td>\texttt{V}</td>
<td>conjunction</td>
</tr>
<tr>
<td>\texttt{NAME}</td>
<td>\texttt{C}</td>
<td>name</td>
</tr>
<tr>
<td>\texttt{LPAR}</td>
<td>\texttt{I}</td>
<td>left parenthesis</td>
</tr>
<tr>
<td>\texttt{RPAR}</td>
<td>\texttt{I}</td>
<td>right parenthesis</td>
</tr>
<tr>
<td>\texttt{ASGN}</td>
<td>\texttt{I}</td>
<td>assignment</td>
</tr>
<tr>
<td>\texttt{MARK}</td>
<td>\texttt{I}</td>
<td>parser marker</td>
</tr>
<tr>
<td>\texttt{SYMB}</td>
<td>\texttt{SY}</td>
<td>symbol table</td>
</tr>
</tbody>
</table>

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

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

\[
\begin{align*}
\#define \texttt{NUMERIC} & \quad (\texttt{BOOL}+\texttt{INT}+\texttt{FL}+\texttt{CMPX}) \\
\#define \texttt{NOUN} & \quad (\texttt{NUMERIC}+\texttt{CHAR}+\texttt{BOX})
\end{align*}
\]

Then the phrase \texttt{NUMERIC\&AT} (\( \mathbf{x} \)) tests for numeric arrays, and the phrase \texttt{NOUN\&AT} (\( \mathbf{x} \)) 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:

\[
\begin{align*}
\text{cvt} (t, x) & \quad \text{Convert } x \text{ to type } t; \text{ signal error if not possible} \\
\text{pcvt} (t, x) & \quad \text{Convert } x \text{ to type } t; \text{ return } x \text{ if not possible} \\
\text{xcvt} (x) & \quad \text{Convert } x \text{ to the “lowest” type}
\end{align*}
\]

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

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

```c
#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 \( \text{AN, AT, AV, etc.} \)
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 tems used in reshape, over, or shape, because they are accounted for in the main loop.

On the other hand, a function may account for tems: On entry into the function, the current top-of-stack is recorded; on exit, tems are freed down to the recorded point. (These actions are mediated by the macros PROLOG and EPILOG.) Whether a function accounts for tems 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 \texttt{AF} and the macros \texttt{F1} and \texttt{F2} codify these properties:

\begin{verbatim}
typedef A(*AF)();

#define F1(f) A f(w)A w;
#define F2(f) A f(a,w)A a,w;
\end{verbatim}

\texttt{AF} is the data type of a function having these properties. \texttt{F1} and \texttt{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 \texttt{VERB}; the details of this representation are
deferred until the next chapter, 4 \textit{Adverbs and Conjunctions}.

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

1. Define and declare functions which implement the monad and dyad.
2. Associate \texttt{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 \texttt{j} are added to file
   \texttt{vm.c} (or to one of several \texttt{v*.c} files), and declarations are added to \texttt{je.h}:

   \begin{verbatim}
   FILE vm.c
   F1(jdot1)\{ R times(a0j1,w); \}
   F2(jdot2)\{ R plus(a,jdot1(w)); \}
   \end{verbatim}

   \begin{verbatim}
   FILE je.h
   extern A jdot1();
   extern A jdot2();
   \end{verbatim}
2. The association between j and jdot1 and jdot2 is established in the tables ps and ps PTR in file t.c. ps PTR[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[ps PTR[CJDOT]]. Entries in that locale are as follows:

/*199 E. CEBAR */ {VERB, 0,abar, 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 ps PTR 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 cannot be specified as static data structures. (ps and ps PTR 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 $&001).@$, 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:

\[
\begin{align*}
\text{rank} & \quad = \quad \#\$ \\
\text{rep} & \quad = \quad 3\&\&.1 \\
\text{cellax} & \quad = \quad 0\&>.@ (+\text{rank}) \langle . \text{rank} \rangle \& (0\&<: \emptyset []) \\
\text{enl} & \quad = \quad \} \langle @@ , \{ . : */@ \} . \} \rangle \& (\&@\&@[]) \\
\text{enc} & \quad = \quad -@\text{cellax} \ ((.\$) \$ \{(.$) \text{enl} , @}]) \} \\
\text{sfx} & \quad = \quad -@<.\&\text{rank} \\
\text{agree} & \quad = \quad (\text{sfx} \{ . \$@\} \& : (\text{sfx} \{ . \$@\}) \\
\text{frame} & \quad = \quad ("err'@+)\& (\$@([\} @. (<\&\text{rank}))) \& . \text{agree} \\
\text{r} & \quad = \quad \text{rep} \text{n} \\
\text{mcell} & \quad = \quad (0\{r\})\&\text{enc} \\
\text{lcell} & \quad = \quad (1\{r\})\&\text{enc}@[ \quad [. \text{lframe} \quad = \quad . \text{frame} (.\$) \} [ \\
\text{rcell} & \quad = \quad (2\{r\})\&\text{enc}@] \quad [. \text{rframe} \quad = \quad . \text{frame} (.\$) \} ] \\
\text{u"n y} & \quad \text{is} \quad \text{u@> mcell y} \\
\text{x u"n y} & \quad \text{is} \quad \text{x} \quad (\text{lcell} \quad (\text{lframe} \quad \text{u@> rframe}) \quad \text{rcell}) \quad \text{y}
\end{align*}
\]

The utility \text{rank} returns the rank of its argument, and \text{rep} replicates one or two ranks into three. \( r \text{cellax} \ y \) computes the number of cell axes for rank \( r \) and noun \( y \); \( s \text{enl} \ y \) boxes the first cell of \( y \) for cell shape \( s \); and \( r \text{enc} \ y \) boxes the cells of \( y \) for rank \( r \). \( l\text{cell} \) (\( r\text{cell} \)) builds an array of boxed left (right) argument cells; \( l\text{frame} \) and \( r\text{frame} \) 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, \( \text{u@>} \) 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 \text{rank1ex} and \text{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 \texttt{f}, rank effects can be achieved by invoking \texttt{ranklex} and \texttt{rank2ex}, mediated by the macros \texttt{F1RANK} and \texttt{F2RANK}:

\begin{verbatim}
A ranklex( 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);
\end{verbatim}

\texttt{a} and \texttt{w} are the left and right arguments of the verb; \texttt{f1} and \texttt{f2} are functions which implement the monad and dyad; \texttt{m, l, r} are ranks; and \texttt{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 \texttt{repeat}, which uses \texttt{F2RANK} as follows:

\begin{verbatim}
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)); ...  
}
\end{verbatim}

If the argument ranks are not greater than the verb ranks, then \texttt{F2RANK (F1RANK)} does nothing, and execution proceeds to the statement following the macro; if the argument ranks are greater, then \texttt{F2RANK (F1RANK)} invokes \texttt{rank2ex (ranklex)}, and on return therefrom exits \texttt{f} with the result obtained therefrom. In this scheme, \texttt{rank2ex (ranklex)} invokes \texttt{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 \texttt{from}, which eschews \texttt{rank2ex} on numeric left arguments when rank effects are rather simple. (\texttt{from} does use \texttt{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 \texttt{self}. The macros \texttt{PREF1} and \texttt{PREF2} are used in such cases, wherein \texttt{ranklex} and \texttt{rank2ex} are invoked with ranks extracted from \texttt{self}, and not with "hard-wired" numbers as in the use of \texttt{F1RANK} and \texttt{F2RANK} for primitive verbs.
3.3 Atomic (Scalar) Verbs

An atomic verb is a primitive verb of the form \( \mathbf{f}_\mathbf{g} \) (that is, a verb whose monad is \( \mathbf{f}_ \) and whose dyad is \( \mathbf{g} \)), where \( \mathbf{f} \) and \( \mathbf{g} \) 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 \texttt{SF1} or \texttt{SF2} (in file \texttt{v.h}). For example, the kernels for the dyad + are as follows (in file \texttt{ve.c}):

\begin{verbatim}
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))
\end{verbatim}

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

\begin{verbatim}
#define SF2(f,T_v,T_x,exp) \ 
B f(u,v,x) T_v*u,*v;T_x*x; {*x=(exp); R!jerr; }
\end{verbatim}
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.)

exp1 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:

\[
\begin{align*}
A \text{sex1} & (A w, I zt, SF f1) \\
A \text{sex2} & (A a, A w, I zt, SF f2)
\end{align*}
\]

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:

\[
\begin{align*}
F2(\text{plus}) & \{
\text{switch(coerce2(&a,&w,BOOL))}\{
\text{case BOOL: R sex2(a,w,INT ,bplus);}\\
\text{case INT: R pcvt(INT,sex2(a,w,FL,iplus));}\\
\text{case FL: R sex2(a,w,FL ,dplus);}\\
\text{case CMPX: R sex2(a,w,CMPX, jplus);}\\
\text{default: R 0;}
\}
\}
\]

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 \( \mathfrak{u} \) is an obverse (usually the inverse) of a verb \( \mathfrak{v} \) if \( \mathfrak{x} \mathfrak{u} \mathfrak{v} \mathfrak{x} \) for a significant subdomain of \( \mathfrak{v} \). The obverse is used in the conjunctions under (\( \mathfrak{s} \).) and power (\( \mathfrak{^}: \)). For example, exponential \( \mathfrak{^} \) and logarithm \( \mathfrak{^}: \) are obverses, and:

\[
\begin{align*}
3+&^:.4 &\text{is} &\mathfrak{^} \mathfrak{^}(\mathfrak{^}3)+^:.4 &\mathfrak{^}^:\_1 &\text{is} &\mathfrak{^} \\
3\star&^:.4 &\text{is} &\mathfrak{^} \mathfrak{^}(\mathfrak{^}3)\star^4 &\mathfrak{^}^:\_1 &\text{is} &\mathfrak{^}
\end{align*}
\]

Obverses are produced by the function \( \text{inv} \) in file ai.c. (\( \text{inv} \) implements \( \mathfrak{^}^:\_1 \).) 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 \( \text{ps} \) in file t.c. For example, the ID for \( \mathfrak{^} \) is \( \text{CEXP} \) and that for \( \mathfrak{^} \) is \( \text{CLOG} \); therefore \( \text{ps}[\text{CEXP}] \mathfrak{.inv} \) is \( \text{CLOG} \) and \( \text{ps}[\text{CLOG}] \mathfrak{.inv} \) is \( \text{CEXP} \). (\( \text{ps}[\_] \mathfrak{.inv} \) is zero otherwise.)

BONDED VERBS. Bonding (Currying) is fixing an argument of a dyad to derive a monad: \( \mathfrak{n} \mathfrak{v} \) or \( \mathfrak{v} \mathfrak{n} \). For example, \( 10\mathfrak{^}. \) is base-10 log and \( \mathfrak{^}60.5 \) is square root. The obverse of a bonded verb is computed by the subfunction \( \text{invamp} \) in file ai.c, invoked by \( \text{inv} \) 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 \text{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 OVERSE. 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 OVERSE. 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 (1.), 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 \texttt{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, \texttt{jerr} must be inspected.

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

The macro \texttt{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 \texttt{w} is a literal atom:

\begin{verbatim}
ASSERT(!AR(w), EVRANK);
ASSERT(CHARAT(w), EVDOMAIN);
\end{verbatim}

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

\begin{verbatim}
FL(iota){A z;I m,n,*v;
    FLRANK(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;
}
\end{verbatim}
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 \texttt{io\text{}ta} above:

\begin{verbatim}
RZ(z=reshape(mag(w),apv(ABS(m),0L,1L)));
\end{verbatim}

If \texttt{reshape} did not check for zero arguments, the statement would have to be elaborated:

\begin{verbatim}
RZ(t0=mag(w));
RZ(t1=apv(ABS(m),0L,1L));
RZ(z=reshape(t0,t1));
\end{verbatim}

A \textit{conventional function} is a function that follows the conventions described herein — return zero on zero arguments and on errors. The data type \texttt{AR} (j.t.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 \texttt{RZ} or \texttt{RE}, and the resulting programs are neater. For example, consider the functions \texttt{lamin1} and \texttt{lamin2} (vs.c), implementing \textit{laminate} (: :):

\begin{verbatim}
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));}
\end{verbatim}

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

Chapter 3 27 Verbs
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):

\[
\begin{align*}
m \& n & \text{ undefined} \\
m \& v & m \& v \ y \ is \ m \ v \ y \\
u \& n & u \& n \ y \ is \ y \ u \ n \\
u \& v & u \& v \ y \ is \ u \ v \ y; \ x \ u \& v \ y \ is \ (v \ x) \ u (v \ y)
\end{align*}
\]

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

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

f1 monad
f2 dyad
f left conj. or adverb argument
r right conj. or adverb argument
h auxiliary argument

If fn=.*&*, the internal array for fn is:

<table>
<thead>
<tr>
<th>t</th>
<th>c</th>
<th>n</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERB</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>on1</th>
<th>on2</th>
<th>%</th>
<th>!</th>
<th>0</th>
<th>_</th>
<th>_</th>
<th>_</th>
<th>&amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>f2</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>mr</td>
<td>lr</td>
<td>rr</td>
<td>id</td>
</tr>
</tbody>
</table>

Chapter 4

28     Adverbs and Conjs
Access to fields in \( \texttt{fn} \) is by name and by macros defined in \texttt{jt.h} and \texttt{a.h}, and \emph{never} by offsets and indices. Thus, \( \texttt{AV(fn)} \) points to the "value" of \( \texttt{fn} \); and if \( \texttt{v=(V*)AV(fn)} \), then \( \texttt{v->f1} \) is \texttt{on1}; \( \texttt{v->f2} \) is \texttt{on2}; \( \texttt{v->f} \) is the array for \( \&. \); \( \texttt{v->g} \) is the array for \( 1 \): (that is, \( \texttt{v->f} \) and \( \texttt{v->g} \) are arrays similar to \( \texttt{fn} \)); \( \texttt{v->mr} \) is \_ (indicating that \( \texttt{fn} \) has infinite monadic rank); and so on. The macro \( \texttt{VAV(f) = ((V*)AV(f))} \) is useful for working with adverbs and conjunctions.

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

\texttt{FILE c.c:}

\begin{verbatim}
static DF1(withl) {DECLFG; R g2(fs,w,gs);}  
static DF1(withr) {DECLFG; R f2(w,gs,fs);}  
static CS1(on1, f1(gl(w,gs),fs))
static CS2(on2, f2(gl(a,gs),gl(w,gs),fs))
\end{verbatim}

\texttt{F2(amp) {}}

\begin{verbatim}
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));
}
\end{verbatim}

\texttt{FILE je.h:}

\begin{verbatim}
extern A amp();
\end{verbatim}

Corresponding to the four possibilities, \texttt{amp} defines four cases, which either signal error or return a verb; the functions \texttt{withl}, \texttt{withr}, \texttt{on1}, and \texttt{on2} are invoked when a verb derived from \& is applied. For example, \( \&.1: m=.?4 4$100 \) first branches to the case \texttt{VV} in \texttt{amp}, and subsequently applies \texttt{on1} to \texttt{m}. Consider a partial macro expansion of \texttt{on1} and the values of its local variables for this example:

\begin{center}
\textbf{Chapter 4} \hspace{2cm} 29 \hspace{2cm} \textbf{Adverbs and Conjs}
\end{center}
MACRO EXPANSION:

```c
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)->g1:0, g2=gs?VAV(gs)->g2:0;
    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 `&`:

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

Chapter 4  30  Adverbs and Conjs
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:

\[
\begin{align*}
\text{A } f1(\text{A } w); \\
\text{A } f2(\text{A } a, \text{A } w);
\end{align*}
\]

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,

\[
\begin{align*}
df1(w, ds(CPOUND)) \\
df1( w, amp(ds(CPOUND), ds(COPE))) \\
df2(a, w, amp(ds(CPOUND), ds(COPE))) \\
df1(w, qq(ac1(f1), sc(1L)))
\end{align*}
\]

The phrase \( ds(CPOUND) \) is the verb \#, \( ds(COPE) \) is the verb \( \rangle \), and \( amp(ds(CPOUND), ds(COPE)) \) is \( \&\&\rangle \); the examples compute \#w, \#\&\rangle w, a\#\&\rangle 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 \textit{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:

\begin{tabular}{ll}
    noun & ID \\
    verb & ID \\
    adverb & ID \\
    conjunction & ID \\
\end{tabular}

The ID is a string computed by the function \texttt{spellout} in file \texttt{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:

\begin{itemize}
    \item `'0' noun
    \item `'2' hook
    \item `'3' fork
    \item `'4' bonded conjunction
    \item `'5' 2-element a-train or c-train
    \item `'6' 3-element a-train or c-train
\end{itemize}

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:
\[\text{ar} = 5!:1\]
\[\text{plus} = +\]
\[\text{sum} = +/\]
\[\text{mean} = +/ % #\]
\[\text{ar} < '\text{plus}'\]
\[\text{ar} < '\text{sum}'\]
\[\text{ar} < '\text{mean}'\]

\[+ \quad (+/) \quad (+/ % #)\]

\[\text{a} = .5\]
\[\text{ar} < 'a'\]

\[\text{xenos} = .!:\]
\[\text{ar} < '\text{xenos}'\]

\[\text{ar} < 'ar'\]

\[\text{Ingamma} = . ^ . @ ! @ <:\]
\[\text{ar} < '\text{Ingamma}'\]

Chapter 5 33 Representation
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
dpdl =. {.root (dpx&.@{. , dp &.@{.)@oarg
dpgr =. {.root (dp &.@{. , dpx&.@{.)@oarg
dpg =. dpgr`dpdl`dpx @. (i.&(<,'`')@oarg)
dptil =. dpx`oarg@.@{.@oarg) @. (((<,'0')&.@{.@@{.@oarg
dpcase =. oarg`dpdl`dpgl`dpgr`dptil`dpx @.
                        ((=:00.`:4~')&i.@{.)

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

```
display <'display'
```

Chapter 5  
34  
Representation
The model is divided into groups of verbs. The first group are utilities:

\begin{itemize}
  \item \texttt{ar} \quad \text{atomic representation}
  \item \texttt{type} \quad \text{type}
  \item \texttt{boxed} \quad 1 \text{ if boxed}
  \item \texttt{oarg} \quad \text{open the second element of the list argument}
\end{itemize}

\texttt{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 \textit{fork}; see 5.1 \textit{Atomic Representation}), and the result of \texttt{root} is \( a \); otherwise, \( r \ \texttt{root} \ a \) produces:

\[
\begin{align*}
a, r & \quad \text{one argument} \\
(\{.a\}, r, (\}.a) & \quad \text{two arguments} \\
r & \quad \text{no arguments (primitive)}
\end{align*}
\]

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

\[
\begin{array}{ll}
\text{ID} & \text{AGENDA} \\
0 & \texttt{oarg} \quad \text{noun (leaf)} \\
8. & \texttt{dpGL} \quad \text{gerundial left subtree} \\
`:` & \texttt{dpGL} \quad \text{gerundial left subtree} \\
4 & \texttt{dpG} \quad \text{bonded conjunction; gerundial left or right subtree} \\
`~` & \texttt{dpTIL} \quad \text{possible instance of \textit{evoke}} \\
\text{other} & \texttt{dpX} \quad \text{none of the above}
\end{array}
\]

display is a model of \( \texttt{5!2} \).
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.

```plaintext
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 ergol1c- 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:

\[
\begin{align*}
ar &=. 5!:1 \\
type &=. 3!:0 \\
boxed &=. 32&=\@type \\
nc &=. 4!:0 \\
t &=. >@({}&(;::'n c n n n xb')\@(1 2 4 8 16 32&i.))@type \\
r s &=. (' '&,)@(@, '& ')@"": (@($@$, $) \\
elem &=. ("": @, )\@(@('-'&sn&."1)@, )@boxed \\
 sn &=. (., ~": @#) \@ (nt@], >@[, (rs, elem)@]) \\
st &=. ('x'&, @((& vac')@nc)\"(nt". @>)@. (2&=\@nc) \\
val &=. (>@ar)\" ~/. &>)@. (2&=\@nc) \\
upfx &=. ).~ >:@(<./)@(i. &' cnb') \\
sr &=. (., ~": @#) \@ (st , ] upfx@sn val)
\end{align*}
\]

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.
\texttt{rs} computes \textit{rank} and \textit{shape}; \texttt{rs y} is `'\,(':\,($\!\$\!y)\,,$\!y)\,,'\,'\,`, the formatted result of the rank and shape, surrounded by blanks.

\texttt{elem} computes \textit{elements}. If the argument \texttt{y} is open, this is simply `'\,\!y\,`, the format of the ravel of \texttt{y}; if boxed, it is the catenation of the representations of the boxed elements.

\texttt{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: \texttt{length}, `'\,:@#\,`; \texttt{type}, \texttt{nt}; \texttt{name}, `:@[`; \texttt{rank} and \textit{shape}, \texttt{rs}; and \textit{elements}, \texttt{elem}.

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

\texttt{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.

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

\texttt{sr} is a model of `5\,:3`.

---

Chapter 5 38 Representation
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.

```plaintext
5!:4 <'tree'

```

![Diagram of a tree representing 5!:4](image)

```
connect =. char @ create @ char @ (root:) @ (tr @ ar)
5!:4 <'tree'

- tree =. @ @ @ @ @ @ @ @ @ @

root =. @ @ @ @ @ @ @ @ @ @

- tr =. @ @ @ @ @ @ @ @ @ @

- ar =. @ @ @ @ @ @ @ @ @ @

@ =. 5!:1

@ =. 3!:0

@ =. 32&=@type

@ =. 0&.=@$e.

@ >@ (1&{})

| =. |. !."'

| =. 1&(|. !."')

@ =. 1& ((1 1&.)@(_1 _1&.)@":@<

@ =. 9!:6 ''

@ =. 10{boxc

@ =. (+./ \ * +./ \) @ (' '&~:) @: ("1)

@ =. 1& |.@$ 1&~: }. (10 6 0{boxc) & @ ($ & (9{boxc))

@ =. -@ (i. 1& @ |. #@ [ |. limb1@]

@ =. (limb +/-) @ extent, . ]

@ =. [+ |. . dash& (= (':1') @ | ' &,:@ ($ & dash) @ (-& !: $)

@ =. PAD ((i. 1, ' ') @ [ |. @. (mt@))

@ =. # <@ {."1 ; >./ @: ("@)@>

@ =. (0{boxc) &@=sh1@ [ |. !: ' '&~: @]
```

Chapter 5
kernb = . (6{boxc} & @) * . ' &~: @sh1[@
 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`0{ . )
 center = . ((i & 1) - @+ < . @- : @ (+ / ) ) @ ] . # @ ] [. [
 root = . label @ ( center extent @ @ [ . @ ]

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

 trx = . @ (root ; ) graft @ : (trx @ ) @ 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} ) * . sh1 "1@ ( e . & dash )
 left = . (3 {boxc} rep ( e . & (9 {boxc} ) * . sh1 "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

Chapter 5 40 Representation
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:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>@</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>@</td>
</tr>
<tr>
<td></td>
<td></td>
<td>@</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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:
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 \( \bot \) (bot), \( \mid \) (left), \( \uparrow \) (cross), and \( \uparrow \) (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 asign 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 thorn1 (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 \texttt{y} is a literal array having the same rank as \texttt{y} (but at least one), such that the shape of \texttt{:.y} matches the shape of \texttt{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:

\begin{verbatim}
sprintf =. "",
type =. 3!:0
real =. {.@+
imag =. {:@+

minus =. $&'_@('@'-'&=@(.)
umbar =. >@(@(e.&(<._1' _ _ _ _.'))@('iInN'&i.@{.)
afte =. minus , (i.&O@(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@:(#&>)
ths =. (-@width ;@:({.&}>"1 ]) @ fmt
\end{verbatim}

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

\begin{itemize}
\item \texttt{sprintf} a function in the C library
\item \texttt{type} type
\item \texttt{real} the real part of a complex number
\item \texttt{imag} the imaginary part of a complex number
\end{itemize}
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 $\mathbf{b}$ is a literal array $\mathbf{a} = .".\mathbf{b}$ such that:
- The rank of $\mathbf{a}$ is the greater of 2 or the rank of $\mathbf{b}$.
- Excluding the last two axes, the shape of $\mathbf{a}$ matches the shape of $\mathbf{b}$.
- The frame (formed by $\text{TIFO}_s$) is the same in all the planes.

Boxed display can be modelled as follows:

```plaintext
type   = . 3!:0
boxed  = . 32&= @ type
mt     = . 0&€.@$
boxc   = . 9!:6 ''
tcorn   = . 2 0{boxc
tint   = . 1 10{boxc
bcorn   = . 8 6{boxc
bint   = . 7 10{boxc
sh     = . (*/@) ; , (:) @(1&,)@$ $ ,
rows   = . */\@)@$
bl     = . )@(&0)@(+/)@(!=)@)(/ i.@.(@(&1))
mask   = . 1&,. #&,, .&0@>:@i.@#
mat    = . mask@bl@rows { ' '&,@sh
edge   = . ,@(1&,.@[ )# +:@@@[ $ ]
left   = . edges(3 9{boxc)@>(0&)@[ , "0 1 ]
right  = . edge&(5 9{boxc)@>(0&)@[ ,"0 1 ]
top    = . 1&|.@((tcorn&,)@((edge&int)@>(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&.>
```

Chapter 6  47  Display
The model is divided into groups of definitions (which are verbs unless indicated otherwise). The first group are utilities:

- **type**
- **boxed** 1 if boxed
- **mt** 1 if empty
- **boxc** (noun) box drawing characters
- **tcorn** (noun) the characters \( \Gamma \Gamma \)
- **tint** (noun) the characters \( \Upsilon - \)
- **bcorn** (noun) the characters \( \Upsilon \Lambda \)
- **bint** (noun) the characters \( \Upsilon \Lambda \)

**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 \( \Gamma \Gamma \Upsilon - \) (top), \( \Upsilon \Lambda \Upsilon - \) (bot), \( \Upsilon \Upsilon \) (left), or \( \Upsilon \Upsilon \) (right) at appropriate positions on the perimeter of each plane.

**topleft** catenates the characters \( \Upsilon \Lambda \Upsilon - \) 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:

\[
\text{y = .(i.2 3);'abc';(i.4 1);(<2 2$'ussr');12;<+.&>.i.2 2 3}
\]

\[
y = . 2 3$y
\]

\[
x = .\text{mat@th&.}>y
\]
\[
\begin{array}{cccc}
& & \text{rc } x & \{ \text{rc } x \} \\
2 & 5 & 1 & 3 & 4 & 1 \\
4 & 4 & 1 & 2 & 11 & 9 \\
4 & 5 & 4 & 3 & 4 & 9 \\
11 & 5 & 11 & 3 & 11 & 9 \\
\end{array}
\]

\[a = \{2 \ 3 \ 4\}'abcdefghi jklmnopqrstuvwx'\]

\[
\begin{array}{lll}
a & \text{mat } a & \$a \\
abcd & abcd & 2 \ 3 \ 4 \\
efgh & efgh & \\
ijkl & ikl & 7 \ 4 \\
mnop & mnop & \\
qrst & qrs & \\
uvw & uvw & \\
\end{array}
\]

\[\text{topleft } 3 \ 4\}'a' \quad (2 \ 3; 4 \ 5) \ \text{perim } 6 \ 10\}'a'\]

\[t=\{\{\text{rc } x\}\}\text{inside@:}(\text{take@>})\}'2 \ x \quad (\text{rc } x) \ \text{perim } t\]

\[
\begin{array}{ccc|ccc|ccc}
\hline
0 & 1 & 2 & abc & 0 & 1 & 2 & 3 & \hline
0 & 1 & 2 & us & 0 & 1 & 2 & \hline
1 & 2 & 3 & 12 & 0 & 1 & 2 & 3 & 4 & 5 & \hline
2 & 3 & 5 & sr & 6 & 7 & 8 & 9 & 10 & 11 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

Chapter 6

49

Display
6.3 Formatted Display

\( x^n \cdot y \) 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:

```plaintext
fmtexp = '++\'@* , _3@('00',)@":@ |
cexp = :@i.'e') (\. , fmtexp".\.\.) ]
cminus = '-\'((\'e,'_ ' # i.@#)\])
larg = (+_20@\@0=)@-=@1\@\/@@"\.(.-\' %e')
nsprintf = larg@ cexp@cminus@": ]
pssprintf = ".''(.-\' %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:

```plaintext
Chapter 6 50 Display
```
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 1b on left argument.
7. Comparatives

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

\[ x = y \quad \text{if} \quad (|x-y| < :! .0 \text{ qct} * (|x|>.|y|) \]

(\(< :! .0 \) 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 \(\text{qct}\) times the magnitude of the larger. \(\text{qct}\), 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 \(\text{fit}\) conjunction (!.).

Tolerant relations can be modelled as follows:

\[
\begin{align*}
teq &= \text{!.|@- < :! .0 \text{ qct}&@>.&|} & \text{in file ut.c} \\
tlt &= . < ! .0 * . -. & \text{ut.c} \\
tle &= . < : ! .0 + . = & \text{ut.c} \\
tfloor &= . < : ! .0 ([ - -.@tle]) & \text{ut.c} \\
tceil &= . < : ! .0 ([ + tlt]) & \text{ut.c} \\
dsignum &= . \text{qct}&<@| * 0&< - 0& > & \text{ve.c} \\
jsignum &= . \text{qct}&<@| * (\%)) & \text{ve.c}
\end{align*}
\]

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 =. ]` (l&=) @> (feq l&=) 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**: \(\pi, 3.14159265358979\ldots\)
- **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 \(2 \_ \_\_\_\)
= 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 @ / @ : + 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 + . =

Chapter 8 55 Primitives
> monad v.c ope

mrk =. >./@:(rank@>)
crk =. mrk (-@[{.$&i@[,$@])&.>

 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 g@ =. -.@<:

>. 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 ve.c rect =. 9 116o."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 126o."0"_
*. dyad ve.c lcm See 3.3 Atomic Verbs.
*: monad ve.c square =. *~
*: dyad vb.c rand =. -.@*.
- 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 =. qt 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)

Chapter 8 58 Primitives
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 =. 1: % @. (1&>:@#)

  minv =. (i. @$: ($,) (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.@]-]-|~)#)1])@.(*@rank@))"0 _

|: monad vs.c cant1 =. i.@-@rank |: ]

|: dyad vs.c cant2

mask =. =/ i.@>:@ (>./)

vec =. >@((@:(i.&.@)>)((<./ .)+ _&@@-.)

ind =. vec +/- .* (#. |:)

Chapter 8 60 Primitives
canta =. (_GPS ind mask@[]) [ ,_0
en =. - #_;
ci =. (/:@pf {) [ 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

Chapter 8 61 Primitives
,: 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)


;: 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

Chapter 8 62 Primitives
dyad ve.c base2

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

monad ve.c abasel

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

dyad ve.c abase2 =. ([ | ([%-]-1)/\.@).@,)"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 =. (!#) (=@[`] (")) \n
  osub =. >@[` (>@[ >@[ {}] ) @. (}*#))

  oind =. (+/@&. </.&. i.)@(2&(@(, &1 1)@$)

  oblique =. ( @(osub"0 1)) (oind`) (`=,(@(<"_2))) \n
  sldot =. id (oblique : key)

/: monad vg.c gradel

  qsort =. NB. a function in the C library

  arg =. <"_1 ,. ]&.@i.@#

  gradel =. >@[:"1 @ qsort @ arg

/: dyad vg.c grade2 =. (~ /:

\ adverb ap.c bslash

  base =. 1&>.@-@[ * i.@em

  iind =. base ,. [@[ <. en - base

  seg =. (i.+i.)/@[ [ ]"1 _

  infix =. (/@seg) (iind `) (")) \ ("0 _

  prefix =. (@[ [ >:@, @i.@#`) (")) \ 

  bslash =. id (prefix : infix)

\. adverb ap.c bsdot

  en =. #]

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

Chapter 8  64  Primitives
kay =. en'em @. (0&<@[])
omask =. (em,en) $ ($&0@[ , $&1@kay)
outfix =. (@#) (omask') (\()) \ ("0 _)
suffix =. (i.@) ~ (\(, .@i. @#)) \ 
bsdot =. id (suffix : outfix)

\: monad  vg.c  dgradel

qsort  NB. a function in the C library
darg =. <"_1 , . -&.>@i. @#
dgradel =. |.@- @: (>@["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

\{ dyad vs.c from

ifrom =. (#@) pind [] >@{ "@_0
afi =. pind\'(i.@[-(pind)) \@ (boxed@))
afrom =. (@@].@:afi\') >@{ i: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 rtrace

m} m"_

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


\. monad vs.c behead =. 1&).
. dyad vs.c drop
   pi =. 0<@[ * 0<@. @-
   ni =. 0>:@[ * 0>@@+ y
   di =. ( {. ~ rank) (pi + ni) $@[ u
   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

Chapter 8 67 Primitives
@: 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 =. (\.(^:_))@&

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

? monad v.c roll

tick =. [ <.@%~ (* 'qrl=:(<:+^31)|(7^5)*qrl':'' )@]

roll =. (<:^31)&tick"0

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

? dyad v.c deal

tick =. [ <.@%~ (* 'qrl=:(<:+^31)|(7^5)*qrl':'' )@]

step =. <@~.@((+ (2^31)&tick)/\)@[ C. ]

arg =. <@i.@[@[ [ ,.& "-- ]

deal =. ([ {. >@(step&.>/)@arg)"0

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

)

CX.C label

a. noun j.c aip The 256-letter ASCII alphabet.

A. monad vp.c adotl

ord =. >:@(>./) base =. >:@i.@-@#
rfd =. +/@({.>}.)/.
dfr =. /:^:2@/,/ adotl =. (base #. rfd)@((ord pfll ])`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@]) [] { []) " _

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

e. dyad      vb.c  eps    =. i.&< #@[n]

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

Chapter 8  69  Primitives
i. **dyad** \( \text{vh.c indexof} \)

If \( x \) and \( y \) are literal lists, then \( x \text{i. } y \) is \( x \text{ciof } y \):

\[
\text{map} = ' (i.\text{¬y.}) \ (a.i.|.y.) \) 256$\#y. ' : '
\[
\text{ciof} = a. \& i. \theta \} \{ \text{map}@[\]

Otherwise, if \( x \) and \( y \) are not floating point or complex numbers, or if the comparison tolerance \( \text{qct} \) is zero, a straightforward hashed algorithm is used.

Otherwise, if \( x \) or \( y \) are floating point numbers and \( \text{qct} \) is nonzero, an algorithm due to Arthur Whitney is used:

\[
\text{bit } x \quad \text{Convert a floating point } x \text{ into a Boolean vector}
\]

\[
\text{tib } b \quad \text{Convert a Boolean vector } b \text{ into a floating point number}
\]

\[
\text{hash } b \quad \text{Hash function on a Boolean vector } b
\]

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

\[
\text{hl} = \text{.hash } \text{mask*.bit } yj*1-qct
\]

\[
\text{hr} = \text{.hash } \text{mask*.bit } yj*1+qct
\]

Look for \( \text{hl} \) and \( \text{hr} \) in the list of hashed \( xi \)'s. In other words, if \( \text{hash} \) were a perfect hash, then for \( \text{th} = \text{.hash@} (\text{mask*.})@ \text{bit"0}, \)

\[
x \text{i. } y \text{ is } ((\text{th } x)\text{i. th } y*1-qct) ((\text{th } x)\text{i. th } y*1+qct).
\]

---

**Chapter 8**

---

**j. monad** \( \text{vm.c jdot1 } = . \text{0j1}* \)

**j. dyad** \( \text{vm.c jdot2 } = . (+ j.)"0 \)

**NB.** \( \text{w.c wordil } \) See 1.1 Word Formation.

**o. monad** \( \text{vm.c pix } = . \text{pi}* \)
o. dyad \texttt{vm.c circle}

\begin{verbatim}
sin  =. 1&o. \quad \text{NB. a function in the C library}
cos  =. 2&o. \quad \text{NB. a function in the C library}
sinh =. 5&o. \quad \text{NB. a function in the C library}
cosh =. 6&o. \quad \text{NB. a function in the C library}

cir0 =. 1&+  %:@* 1&-
zp4 =. -60j1 %:@*  +&0j1
zp8 =. 0j1&+ %:@* 0j1&-
zm4 =. +&1 * -&1 %:@% +&1
real =. \ioms{-0}(+-)
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
ztan =. zatanh&. j.

zasin =. zasinh&. j.
zacosh =. ]`j.@|@imag)@.(0>&@real) @ (^.&+ zm4)
ztanh =. 1&+ -@^.&% 1&-

\begin{verbatim}
cirp =. (cir0@[)`(zsin@)`(zcosh@)`(ztan@)`(zp4@)`
   `(zsinh@)`(zcosh@)`(ztanh@)`(zp8@)`)
   `(real@`)`(|@`)`(imag@`)`(zarc@`) @. [
cirm =. (cir0@[)`(zasin@)`(zacos@)`(zatan@)`(zm4@)`
   `(zasin@)`(zacos@)`(zatan@)`(zm4@)`
   `(zsinh@)`(zcosh@)`(ztanh@)`(-@zp8@)`)
   `(|@`)`(j.@)`(r.@)` @. [1@]
circle =. cirp`cirm @. (0>&@[]) " 0
\end{verbatim}

See \textit{Handbook of Mathematical Functions}, Chapter 4 (Abramowitz and Stegun [1964]).
<table>
<thead>
<tr>
<th>p. monad</th>
<th>vm.c</th>
<th>poly1</th>
<th>Not yet available.</th>
</tr>
</thead>
<tbody>
<tr>
<td>p. dyad</td>
<td>vm.c</td>
<td>poly2</td>
<td>Not yet available.</td>
</tr>
<tr>
<td>r. monad</td>
<td>vm.c</td>
<td>rdot1</td>
<td>=. ^@j.</td>
</tr>
<tr>
<td>r. dyad</td>
<td>vm.c</td>
<td>rdot2</td>
<td>=. (* r.)&quot;0</td>
</tr>
<tr>
<td>x.</td>
<td>cx.c</td>
<td>xd</td>
<td></td>
</tr>
<tr>
<td>y.</td>
<td>cx.c</td>
<td>xd</td>
<td></td>
</tr>
<tr>
<td>0: monad</td>
<td>v.c</td>
<td>zero1</td>
<td>=. 0&quot;_</td>
</tr>
<tr>
<td>0: dyad</td>
<td>v.c</td>
<td>zero2</td>
<td>=. 0&quot;_</td>
</tr>
<tr>
<td>1: monad</td>
<td>v.c</td>
<td>one1</td>
<td>=. 1&quot;_</td>
</tr>
<tr>
<td>1: dyad</td>
<td>v.c</td>
<td>one2</td>
<td>=. 1&quot;_</td>
</tr>
</tbody>
</table>
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) {d, *s; {DO(n, d[i]=s[i]);}}
tr(r, d) {I z=1; DO(r, 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(reh) {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:n);
 if(n==wn) mv(z->p+wn, z->p, n); R z;}
V1(shar) {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,resh,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)):c(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

cc.c      conjunctions
cc.c      conjunctions — cuts
cf.c      conjunctions — fit
cg.c      conjunctions — gerunds
cp.c      conjunctions — power
cr.c      conjunctions — rank
cr.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:

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

```c
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:

```c
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
C jc(I k, AF*fl, AF*f2) {
    switch(k) {
        /* k: index */
        /* fl: 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 \( jc(k, &fl, &f2) \), wherein (presumably depending on \( k \)) \*fl 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 \*fl and \*f2 are invoked with those arguments.

File main.c contains an example of using LinkJ. It has a \texttt{main} function which repeats the following steps, \textit{ad infinitum}:

```c
Appendix C 79 LinkJ
```
• 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:

```c
#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


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

left argument
typedef array
typedef translat. action
monad *:
dyad #:
absolute value
A reference count
parser action header
monad from C function
dyad from C function
derive verb from adverb
monad A.
dyad A.
parser action
type
derive an adv from a conj
dyad = A subcase
typedef APL function
standard index
bonded conjunction
A no. of header words
no. of atoms in an item
1 if all ones
A
$&1 _1$#
&
A number of atoms
fne A subcase
A composite type
application file handler
appf subfunction
arithmetic progression
A rank
atomic representation
atomic rep. opened
monad 5!:1
A shape
last op was assignment
A type
argument validation
verb-verb case of conj
adverb-derived monad
AS2 a.h 4 adverb-derived dyad
AT j.t.h 2.1 A type
atan2 C
atco c.c 8 @:
atop c.c 8 @
AV j.t.h 2.1 A value
a0j1 j.c 2.1 0j1

B j.t.h 2.2 type def boolean
band ve.c dyad * B subcase
base l ve.c 8 monad #.
base2 ve.c 8 dyad #.
bdiv ve.c dyad % B subcase
behead vs.c 8 monad 1.
beq vb.c dyad = B subcase
BfromD k.c 7 convert: B from D
BfromI k.c convert: B from I
BfromZ k.c convert: B from Z
bin vm.c dyad ! subfunction
binD vm.c dyad ! subfunction
binI vm.c dyad ! subfunction
ble vb.c dyad <: B subcase
blt vb.c dyad < B subcase
bminus ve.c dyad - B subcase
bool a.c 8 b.
BOOL j.t.h 2.2 A type
booltab a.c 8 function values for b.
bool1 a.c monad m b.
bool2 a.c dyad m b.
btor ve.c dyad + B subcase
box vs.c 8 monad <
BOX j.t.h 2.2 A type
boxq x.c 6.2 monad 9!6
boxes x.c 6.2 monad 9!7
bp u.c 2.2 bytes per atom
bplus ve.c dyad + B subcase
break C
breaker io.c check for user break
brem ve.c dyad | B subcase
bsdot ap.c 8 \
bslash ap.c 8 \
BxD k.c convert: B from D case
BxI k.c convert: B from I case
BxZ k.c convert: B from Z case
bytes m.c bytes in use
C j.t.h 2.2 type def byte
C* j.c.h 1.1 character ID codes
c a m.c copy array
cA j.t.h 1.1 char type: letter
cadv pc.c 1.2 :12 translator action
canta vs.c 8 : subfunction
cantm vs.c : on tables
cant1 vs.c 8 monad 1:
cant2 vs.c 8 dyad 1:
car m.c copy array recursively
case C
cases p.c 1.2 parse table
casel cg.c 8 monad m@.v
case2 cg.c 8 dyad m@.v
catalog vs.c 8 monad {
catsp xw.c ,&
CB j.t.h 1.1 char type: B
CC j.t.h 1.1 char type: colon
ccong pc.c 1.2 :12 translator action
ccurry pc.c 1.2 :12 translator action
ccvt k.c convert: conditional
CD j.t.h 1.1 char type: dot
CDERIV a.h 4 derive verb from conj
cdot1 vp.c 8 monad c.
cdot2 vp.c 8 dyad c.
cdyd pc.c 1.2 :12 translator action
ceiling1 ve.c 8 monad >
center rt.c 5.4 5!4 subfunction
ceq vb.c dyad = C subcase
cfd vp.c 8 cycle from direct
cforqv pc.c 1.2 :12 translator action
cformo pc.c 1.2 :12 translator action
cg aq x.c PC monad 8!0
cgas x.c PC monad 8!1
cgav x.c PC 8!0 setting
char C
char CHAR j.t.h 2.2 A type
chookv pc.c 1.2 :12 translator action
ciof vh.c 8 dyad i. subfunction
circle vm.c 8 dyad o.
clock C
clock-related j.h 1.1 clock-related
cmonad pc.c 1.2 :12 translator action
CMFX j.t.h 2.2 A type

Glossary 87

Index
do  C  do n times under index i
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
dx  x.c  5.2  monad 5!|2
ds au.c  3.1  define symbol
dsibling 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
eigl vm.c  8  monad c.
eig2 vm.c  8  dyad c.
eelse 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  $ handle
eo  c.c  u . v and u : : v
EPilogue j.h  2.3  temps clean-up
eps vb.c  8  dyad e.
eq vb.c  8  dyad =
ersee j.c  3.5  if display event msgs
EV* j.h  3.5  event codes
even cc.c  8  .
every a.c  "each" operator variant
ever c.g  3 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
eexec v.c  8  monad "."
exec2 v.c  8  dyad "."
expn1 vm.c  8  monad ^
expn2 vm.c  8  dyad ^
extrm C
fa m.c  free array
fabs C
facf vm.c  monad ! subfunction
fact vm.c  8  monad !
factpl 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
fdef au.c  derive verb/adv/conj
ferr C
fgetc C
fgets C
fh vh.c  8  dyad i. hasher
fi u.c  string to integer
fibon C
cp.c  8  dyad u:^n
FILL j.t.h  fill value
filler u.c  fill value
FINDC vh.c  8  dyad i. find in hash
fit cf.c  3.4  !.
fitctl cf.c  3.4  monad u!:n
fitct2 cf.c  3.4  dyad u!:n
fitpp1 cf.c  3.4  monad "::!n
fill a.c  8  f.
fix a.c  u f. subfunction
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 j.t.h  2.2  A type
floor C
floor1 ve.c  8  monad <.
Glossary 90 Index
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nand</td>
<td>vb.c</td>
<td>dyad *:</td>
</tr>
<tr>
<td>nc</td>
<td>s.c</td>
<td>4!0 subfunction</td>
</tr>
<tr>
<td>ncases</td>
<td>p.c</td>
<td>1.2 no. of rows in cases</td>
</tr>
<tr>
<td>nx</td>
<td>x.c</td>
<td>monad 4!0</td>
</tr>
<tr>
<td>NDEPTH</td>
<td>p.c</td>
<td>max depth of fn calls</td>
</tr>
<tr>
<td>ndig</td>
<td>w.c</td>
<td>AMIGA LatticeC kludge</td>
</tr>
<tr>
<td>ne</td>
<td>vbc</td>
<td>dyad -:</td>
</tr>
<tr>
<td>negate</td>
<td>vcc</td>
<td>monad -</td>
</tr>
<tr>
<td>neg1</td>
<td>jc</td>
<td>2.1 _1</td>
</tr>
<tr>
<td>seq</td>
<td>vbc</td>
<td>dyad - subfunction</td>
</tr>
<tr>
<td>NEVM</td>
<td>jh</td>
<td>3.5 no. of event messages</td>
</tr>
<tr>
<td>NKFD</td>
<td>io.h</td>
<td>size of fn key defn buffer</td>
</tr>
<tr>
<td>NINPUT</td>
<td>jh</td>
<td>max length of input line</td>
</tr>
<tr>
<td>nla</td>
<td>sc</td>
<td>4!1 initial interest</td>
</tr>
<tr>
<td>nline</td>
<td>cx.c</td>
<td>m:n number of lines</td>
</tr>
<tr>
<td>nlmask</td>
<td>sc</td>
<td>4!1 numbers to type</td>
</tr>
<tr>
<td>NLOG</td>
<td>io.h</td>
<td>size of session log</td>
</tr>
<tr>
<td>nls</td>
<td>sc</td>
<td>4!1 subfunction</td>
</tr>
<tr>
<td>nlk</td>
<td>sc</td>
<td>4!1 subfunction</td>
</tr>
<tr>
<td>nil</td>
<td>xc</td>
<td>monad 4!1</td>
</tr>
<tr>
<td>nl2</td>
<td>xc</td>
<td>dyad 4!1</td>
</tr>
<tr>
<td>NMEM</td>
<td>mc</td>
<td>max size for malloc</td>
</tr>
<tr>
<td>NN</td>
<td>a.h</td>
<td>4 noun-noun case of conj</td>
</tr>
<tr>
<td>NOBUF</td>
<td>jh</td>
<td>length of obuf</td>
</tr>
<tr>
<td>nor</td>
<td>vbc</td>
<td>8 dyad +:</td>
</tr>
<tr>
<td>norm</td>
<td>vi.c</td>
<td>%:0 (+/ . * )</td>
</tr>
<tr>
<td>not</td>
<td>vcc</td>
<td>8 monad -</td>
</tr>
<tr>
<td>NOTCONJ</td>
<td>jth</td>
<td>1.2 A composite type</td>
</tr>
<tr>
<td>NOUN</td>
<td>jth</td>
<td>2.2 A composite type</td>
</tr>
<tr>
<td>NPP</td>
<td>jh</td>
<td>max value for qpp</td>
</tr>
<tr>
<td>NPPROMPT</td>
<td>jh</td>
<td>max length of prompt</td>
</tr>
<tr>
<td>NTH2</td>
<td>fc</td>
<td>6.3 dyad &quot;::&quot; max width</td>
</tr>
<tr>
<td>NTSSTACK</td>
<td>jh</td>
<td>temps: stack frame size</td>
</tr>
<tr>
<td>nu</td>
<td>w.c</td>
<td>1.1 national use alternatives</td>
</tr>
<tr>
<td>nub</td>
<td>vcc</td>
<td>8 monad -</td>
</tr>
<tr>
<td>nubsieve</td>
<td>vbc</td>
<td>8 monad -</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>jth</td>
<td>2.2 A composite type</td>
</tr>
<tr>
<td>NV</td>
<td>a.h</td>
<td>4 noun-verb case of conj</td>
</tr>
<tr>
<td>NW</td>
<td>xwc</td>
<td>WS length of header</td>
</tr>
<tr>
<td>NWFX</td>
<td>xwc</td>
<td>WS length of prefix</td>
</tr>
<tr>
<td>NWPTR</td>
<td>xwc</td>
<td>WS length of pointer</td>
</tr>
<tr>
<td>NXIL</td>
<td>xwc</td>
<td>WS block size in wcp</td>
</tr>
<tr>
<td>oblique</td>
<td>ap.c</td>
<td>8 monad u/</td>
</tr>
<tr>
<td>obuf</td>
<td>jc</td>
<td>buffer for short output</td>
</tr>
<tr>
<td>obverse</td>
<td>c.c</td>
<td>3.3 :</td>
</tr>
<tr>
<td>obv1</td>
<td>c.c</td>
<td>monad u : v</td>
</tr>
<tr>
<td>obv2</td>
<td>c.c</td>
<td>dyad u : v</td>
</tr>
<tr>
<td>odd</td>
<td>c.c</td>
<td>8 :</td>
</tr>
<tr>
<td>oind</td>
<td>ap.c</td>
<td>8 monad u/. subfunction</td>
</tr>
<tr>
<td>oldout</td>
<td>xs.c</td>
<td>old outfile value</td>
</tr>
<tr>
<td>omask</td>
<td>ap.c</td>
<td>8 dyad u/ subfunction</td>
</tr>
<tr>
<td>one</td>
<td>jc</td>
<td>2.1 i</td>
</tr>
<tr>
<td>onel</td>
<td>v.c</td>
<td>8 monad 1</td>
</tr>
<tr>
<td>one2</td>
<td>v.c</td>
<td>8 dyad 1</td>
</tr>
<tr>
<td>on2</td>
<td>c.c</td>
<td>4 monad u\v and u\v</td>
</tr>
<tr>
<td>ope</td>
<td>vs.c</td>
<td>8 monad &gt;</td>
</tr>
<tr>
<td>oprod</td>
<td>ac</td>
<td>dyad u/</td>
</tr>
<tr>
<td>ord</td>
<td>vpc</td>
<td>8 order of a permutation</td>
</tr>
<tr>
<td>overs</td>
<td>ap.c</td>
<td>8 monad u/. subfunction</td>
</tr>
<tr>
<td>outfile</td>
<td>jc</td>
<td>output file handle</td>
</tr>
<tr>
<td>output</td>
<td>vpc</td>
<td>8 dyad u/</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>jh</td>
<td>large d value</td>
</tr>
<tr>
<td>overr</td>
<td>vs.c</td>
<td>8 dyad ,</td>
</tr>
</tbody>
</table>

**Glossary**

93

**Index**
SCALARFN  a.h  1 if scalar function
scalar4  u.c  2.1 scalar 4-byte object
csc  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 01:2 and 01:3
scpt2  xs.c  dyad 01:2 and 01:3
script1  xs.c  monad 11:2
script2  xs.c  dyad 11:2
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 $:
sem  j.c  1 if using session mgr
semsexit  j.c  session manager: epilog
semsinit  j.c  session manager: prolog
sex  u.c  symbol table expunge
sex1  cr.c  5.3 monad scalar executor
sex2  cr.c  5.3 dyad scalar executor
SF  jl.h  5.3 typedef scalar function
SF1  vl.h  5.3 scalar monad header
SF2  vl.h  5.3 scalar dyad header
SGN  j.h  signum
shape  vs.c  8 monad $
shift1  cf.c  5.4 monad 1..n
shift2  cf.c  5.4 dyad 1..n
sh1  ai.c  1$ (1..w)
short  C
shr  ai.c  1..w
sigpipe  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
sil_ ai.c  u/"1"
SN  wc  1.1 word formation state
SNB  wc  1.1 word formation state
SNZ  wc  1.1 word formation state
sp  xc  monad 71:0
spell  wc  1.1 spelling table
spellin  wc  1.1 ASCII string to ID
spellout  wc  1.1 ID to string
split  xc  monad 71:2
sprintf  C
spz  xc  monad 71:1
SQ  wc  1.1 word formation state
SQO  wc  1.1 word formation state
sqroots  vm.c  8 monad %:
sqrt  C
square  ve.c  8 monad *:
sread  x.h  script opcode
srdb  sc  symbol table read
srdlg  sc  srd local or global
sreduce  xc  monad f/ for scalar f
srep  rc  5.3 string representation
srl  rc  5.3 srep subfunction
srz  xc  5.3 monad 51:3
SS  wc  1.1 word formation state
sscript1  xs.c  monad 11:3
sscript2  xs.c  dyad 11:3
ST  wc  1.1 typedef rhetic states
state  wc  1.1 rhetic state table
struc  C
strcat  C
strchr  C
strmp  C
strncpy  C
strlcn  C
strspn  C
strtod  C
struct  C
stype  xc  2.2 monad 31:0
sum  ve.c  monad +
suffix  ap.c  8 monad u/
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>8</td>
<td>( \text{typedef}) unsigned byte ( \text{WB} )</td>
</tr>
<tr>
<td>under</td>
<td>1</td>
<td>small ( D ) value ( \text{WFD} )</td>
</tr>
<tr>
<td>under1</td>
<td>1</td>
<td>monad ( u_{#} v ) ( \text{WFD} )</td>
</tr>
<tr>
<td>under2</td>
<td>1</td>
<td>dyad ( u_{#} v ) ( \text{WFD} )</td>
</tr>
<tr>
<td>unlink</td>
<td>C</td>
<td>( \text{while} ) ( \text{C} )</td>
</tr>
<tr>
<td>ung</td>
<td>xw.c</td>
<td>WS remove given names ( \text{WF1} )</td>
</tr>
<tr>
<td>unquote</td>
<td>p.c</td>
<td>monad or dyad ( m # ) ( \text{WF2} )</td>
</tr>
<tr>
<td>unquo1</td>
<td>a.c</td>
<td>dyad ( m # ) ( \text{while1} )</td>
</tr>
<tr>
<td>unquo2</td>
<td>a.c</td>
<td>dyad ( u_{#} v ) ( \text{while2} )</td>
</tr>
<tr>
<td>unsr</td>
<td>r.c</td>
<td>( 5 : 3 ) inverse ( \text{WI} )</td>
</tr>
<tr>
<td>until</td>
<td>C</td>
<td>( \text{with} ) ( \text{C} )</td>
</tr>
<tr>
<td>unw</td>
<td>r.c</td>
<td>( \text{unsr} ) subfunction ( \text{withr} )</td>
</tr>
<tr>
<td>upon2</td>
<td>c.c</td>
<td>dyad ( u_{#} v ) ( \text{wnc} )</td>
</tr>
<tr>
<td>v</td>
<td>jth</td>
<td>( \text{typedef}) verb ( \text{wdyad} )</td>
</tr>
<tr>
<td>vadv</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>VAV</td>
<td>jth</td>
<td>( \text{AV for verb/adverb/conj} ) ( \text{wc})</td>
</tr>
<tr>
<td>vconj</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vcurrent</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vdyad</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>VERB</td>
<td>jth</td>
<td>( \text{22}) a type ( \text{vconj} )</td>
</tr>
<tr>
<td>vfile</td>
<td>xfc</td>
<td>validate file name ( \text{vconj} )</td>
</tr>
<tr>
<td>vforkv</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vformo</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vhookv</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vi</td>
<td>uc</td>
<td>validate integer ( \text{vconj} )</td>
</tr>
<tr>
<td>vib</td>
<td>uc</td>
<td>validate integer, bounded ( \text{vconj} )</td>
</tr>
<tr>
<td>vis</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vmonad</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vmove</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{vconj} )</td>
</tr>
<tr>
<td>vn</td>
<td>uc</td>
<td>validate noun ( \text{wgpu} )</td>
</tr>
<tr>
<td>VN</td>
<td>ah</td>
<td>( 4) verb-noun case of conj ( \text{wr}) xw.c WS prefix\</td>
</tr>
<tr>
<td>vnm</td>
<td>sc</td>
<td>validate name ( \text{wgpu} )</td>
</tr>
<tr>
<td>void</td>
<td>C</td>
<td>( \text{wgpu})</td>
</tr>
<tr>
<td>vpunc</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator action ( \text{wgpu} )</td>
</tr>
<tr>
<td>vs</td>
<td>uc</td>
<td>validate string ( \text{wgpu} )</td>
</tr>
<tr>
<td>vtrans</td>
<td>pv.c</td>
<td>( \text{12 : 11}) translator ( \text{wgpu} )</td>
</tr>
<tr>
<td>VV</td>
<td>ah</td>
<td>( 4) verb-verb case of conj ( \text{wgpu} )</td>
</tr>
<tr>
<td>v2</td>
<td>uc</td>
<td>integer pair ( \text{wgpu} )</td>
</tr>
<tr>
<td>w</td>
<td>*</td>
<td>right argument ( \text{wgpu} )</td>
</tr>
<tr>
<td>wa</td>
<td>xfc</td>
<td>file write or append ( \text{wgpu} )</td>
</tr>
<tr>
<td>WATERLOO</td>
<td>jh</td>
<td>1 if Waterloo libraries ( \text{wgpu} )</td>
</tr>
</tbody>
</table>
xigcd  ve.c  dyad +. I subcase subfn
xil   xw.c  WS dir, index/length
XINF  j.h    _internal representation
xlrem  ve.c  dyad I I subcase subfn
XNAN  j.h    _internal representation
xn1  cx.c  monad m : y
xn2  cx.c  monad u : y
xv1  cx.c  dyad x : n
xv2  cx.c  dyad x : v

Z

result
Z

typedef complex
zacos vz.c 8 complex: _2&0.
zacosh vz.c 8 complex: _6&0.
zarc vz.c 8 complex: 12&0.
zasin vz.c 8 complex: _1&0.
zasinh vz.c 8 complex: _5&0.
ZASSERT vz.c complex: arg validation
zatan vz.c 8 complex: _3&0.
zatanh vz.c 8 complex: _7&0.
zceiling vz.c complex: monad >.
zcir vz.c 8 complex: dyad o.
zconjug vz.c complex: monad +
zcos vz.c 8 complex: 2&0.
zcosh vz.c 8 complex: 6&0.
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
zerol v.c 8 monad 0;
zero2 v.c 8 dyad 0;
zexp vz.c 8 complex: monad ^
ZEE 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
zlc vz.c complex: monad ^.

glossary 98  index
Appendix E. Foreign Conjunction

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

0!:0  host xf *
0!:1  hostxe 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 jiframe xf *
1!:12 jiframe xf
1!:55 jferase xf *
2!:0  • wnc xw
2!:1  wnl xw *
2!:2  savel xw • save2 xw
2!:3  psavel 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  n1l s • n12 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  cgq 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  prtscreen 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  ss script1 xs * ss script2 xs
0!:55  joff u *
1!:0  jfdir xf *
1!:1  jfread xf *
1!:2  * jfwrite xf
1!:3  * jffappend xf
1!:4  jfsime 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  psavel 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  r1q x *
9!:1  rls x *
9!:4  promptq x *
9!:5  prompts x *
9!:6  boxq x *
9!:7  boxes 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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sclass</code></td>
<td>vb * eq vb</td>
</tr>
<tr>
<td><code>box</code></td>
<td>v * lt vb</td>
</tr>
<tr>
<td><code>ope</code></td>
<td>v * gt vb</td>
</tr>
<tr>
<td><code>coninf</code></td>
<td>w</td>
</tr>
<tr>
<td><code>+</code></td>
<td>conjuc v * plus ve</td>
</tr>
<tr>
<td><code>*</code></td>
<td>signum v * times ve</td>
</tr>
<tr>
<td><code>-</code></td>
<td>negate v * minus ve</td>
</tr>
<tr>
<td><code>%</code></td>
<td>recip v * divide ve</td>
</tr>
<tr>
<td><code>^</code></td>
<td>expn1 vm * expn2 vm</td>
</tr>
<tr>
<td><code>$</code></td>
<td>shape vs * reitem vs</td>
</tr>
<tr>
<td><code>~</code></td>
<td>swap a</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>.</code></td>
<td>* dot c</td>
</tr>
<tr>
<td><code>:</code></td>
<td>* colon cx</td>
</tr>
<tr>
<td><code>,</code></td>
<td>ravel vs * over vs</td>
</tr>
<tr>
<td><code>;</code></td>
<td>raze v * link v</td>
</tr>
<tr>
<td><code>#</code></td>
<td>tally vs * repeat vs</td>
</tr>
<tr>
<td><code>!</code></td>
<td>fact vm * out of vm</td>
</tr>
<tr>
<td><code>/</code></td>
<td>slash a *</td>
</tr>
<tr>
<td><code>\</code></td>
<td>bslash ap *</td>
</tr>
<tr>
<td><code>[</code></td>
<td>left1 v * left2 v</td>
</tr>
<tr>
<td><code>]</code></td>
<td>right1 v * right2 v</td>
</tr>
<tr>
<td><code>{</code></td>
<td>catalog vs * from vs</td>
</tr>
<tr>
<td><code>}</code></td>
<td>rbrace a *</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>* qq cr</td>
</tr>
<tr>
<td><code>\</code></td>
<td>* tie cg</td>
</tr>
<tr>
<td><code>@</code></td>
<td>* atop c</td>
</tr>
<tr>
<td><code>#</code></td>
<td>* amp c</td>
</tr>
<tr>
<td><code>?</code></td>
<td>roll v * deal v</td>
</tr>
<tr>
<td><code>)</code></td>
<td>label cx</td>
</tr>
<tr>
<td><code>b</code></td>
<td>bool a *</td>
</tr>
<tr>
<td><code>e</code></td>
<td>razein vb * eps vb</td>
</tr>
<tr>
<td><code>i</code></td>
<td>iota v * indexof vb</td>
</tr>
<tr>
<td><code>o</code></td>
<td>pix vm * circle vm</td>
</tr>
<tr>
<td><code>x</code></td>
<td>xd cx</td>
</tr>
<tr>
<td><code>isl</code></td>
<td>p</td>
</tr>
<tr>
<td><code>isg</code></td>
<td>p</td>
</tr>
<tr>
<td><code>floor1</code></td>
<td>ve * minimum ve</td>
</tr>
<tr>
<td><code>ceil1</code></td>
<td>ve * maximum ve</td>
</tr>
<tr>
<td><code>coninf</code></td>
<td>w</td>
</tr>
<tr>
<td><code>rect</code></td>
<td>vm * gcd ve</td>
</tr>
<tr>
<td><code>polar</code></td>
<td>vm * lcm ve</td>
</tr>
<tr>
<td><code>not</code></td>
<td>ve * less v</td>
</tr>
<tr>
<td><code>minv</code></td>
<td>vi * mdiv vi</td>
</tr>
<tr>
<td><code>logari</code></td>
<td>vm * logar2 vm</td>
</tr>
<tr>
<td><code>ensuite</code></td>
<td>cx</td>
</tr>
<tr>
<td><code>nub</code></td>
<td>v *</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>vs * rotate vs</td>
</tr>
<tr>
<td><code>even</code></td>
<td>c</td>
</tr>
<tr>
<td><code>obverse</code></td>
<td>c</td>
</tr>
<tr>
<td><code>table</code></td>
<td>vs * over vs</td>
</tr>
<tr>
<td><code>cut</code></td>
<td>cc</td>
</tr>
<tr>
<td><code>base1</code></td>
<td>ve * base2 ve</td>
</tr>
<tr>
<td><code>fit</code></td>
<td>cf</td>
</tr>
<tr>
<td><code>sldot</code></td>
<td>ap *</td>
</tr>
<tr>
<td><code>bstdot</code></td>
<td>sp *</td>
</tr>
<tr>
<td><code>head</code></td>
<td>vs * take vs</td>
</tr>
<tr>
<td><code>behead</code></td>
<td>vs * drop vs</td>
</tr>
<tr>
<td><code>exec1</code></td>
<td>v * exec2 v</td>
</tr>
<tr>
<td><code>agency</code></td>
<td>cg</td>
</tr>
<tr>
<td><code>under</code></td>
<td>c</td>
</tr>
<tr>
<td><code>a. alp</code></td>
<td>j</td>
</tr>
<tr>
<td><code>c. eigl</code></td>
<td>vm * eig2 vm</td>
</tr>
<tr>
<td><code>E. ebar</code></td>
<td>vb</td>
</tr>
<tr>
<td><code>jdot1</code></td>
<td>vm * jdot2 vm</td>
</tr>
<tr>
<td><code>p. poly1</code></td>
<td>vm * poly2 vm</td>
</tr>
<tr>
<td><code>y. xd</code></td>
<td>cx</td>
</tr>
<tr>
<td><code>decrem</code></td>
<td>ve * le vb</td>
</tr>
<tr>
<td><code>increment</code></td>
<td>ve * ge vb</td>
</tr>
<tr>
<td><code>inf1</code></td>
<td>v * inf2 v</td>
</tr>
<tr>
<td><code>dublee</code></td>
<td>ve * nor vb</td>
</tr>
<tr>
<td><code>square</code></td>
<td>ve * nand vb</td>
</tr>
<tr>
<td><code>halve</code></td>
<td>ve * match vb</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>root vm</td>
</tr>
<tr>
<td><code>* powop * cp</code></td>
<td></td>
</tr>
<tr>
<td><code>self1</code></td>
<td>p * self2 p</td>
</tr>
<tr>
<td><code>nusieve</code></td>
<td>vb * ne vb</td>
</tr>
<tr>
<td><code>cant</code></td>
<td>1 vs * cant2 vs</td>
</tr>
<tr>
<td><code>lamin</code></td>
<td>1 vs * lamin2 vs</td>
</tr>
<tr>
<td><code>words</code></td>
<td>w *</td>
</tr>
<tr>
<td><code>abasel</code></td>
<td>ve * abase2 ve</td>
</tr>
<tr>
<td><code>foreign</code></td>
<td>x</td>
</tr>
<tr>
<td><code>grad1</code></td>
<td>vg * grade2 vg</td>
</tr>
<tr>
<td><code>dggrad1</code></td>
<td>vg * dgrade2 vg</td>
</tr>
</tbody>
</table>
| `thorn1` | / thorn2 /
| `evger` | cg |
| `atco` | c |
| `ampco` | c |
| `label` | cx |
| `bool` | a |
| `razein` | vb * eps vb |
| `iota` | v * indexof vb |
| `pix` | vm * circle vm |
| `xd` | cx |
| `A. adot1` | vp * adot2 vp |
| `C. cdot1` | vp * cdot2 vp |
| `f. fix` | a |
| `NB. wordil` | w |
| `zdot1` | vm * zdot2 vm |
| `zero1` | v * zero2 v |
| `one1` | v * one2 v |